UNIVERSIDAD NACIONAL AUTÓNOMA

DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ARAGÓN

DESARROLLO DE SOFTWARE SEGURO CON UMLSEC Y JAVA.

T E S I S

QUE PARA OBTENER EL TITULO DE:

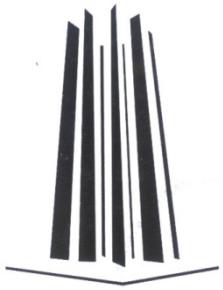
INGENIERO EN COMPUTACIÓN

PRESENTA:

SERGIO LUIS CHAVARRIA CISNEROS.

ASESOR DE TESIS:

M. EN C. LEOBARDO HERNÁNDEZ AUDELO.



MARZO, 2010.

Agradecimientos.

A mi familia, especialmente a mis padres por todo el apoyo que no han dejado de darme, y sin el cual, difícilmente habría conseguido esta meta, que también es suya. Muchas gracias papá, muchas gracias mamá.

A la universidad, porque se convirtió por muchos años en mi segundo hogar, y por enseñarme, además de los conocimientos profesionales, inigualables valores que llevaré a lo largo de la vida.

A todos mis maestros, porque con sus enseñanzas, ejemplos y regaños, han contribuido en el desarrollo personal que me ayudará a lo largo de mi vida profesional.

A todos mis compañeros por su ayuda y apoyo, pero principalmente por su valiosa amistad.

Índice.

Lista de Figuras.

Índice de Tablas

Capítulo 1: Introducción.

Resumen: La información proporciona conocimiento y el conocimiento da poder. Toda información requiere conservar características especiales de acceso, integridad y confidencialidad. A lo largo de la historia de la humanidad se han desarrollado diversos mecanismos para proteger estas características y en la actualidad, con el amplio uso de sistemas computarizados de información, existe la necesidad de adaptar nuevos mecanismos para garantizar su protección. En este capítulo se presenta información básica general, indispensable en el desarrollo seguro de sistemas de información; se abordan las problemáticas actuales en el desarrollo de software, la necesidad de considerar requerimientos de seguridad en las etapas iniciales de desarrollo, así como conceptos básicos de seguridad informática que permitan comprender mejor el objeto de este trabajo. Por otro lado, una de las herramientas de modelado de sistemas más utilizada es el lenguaje UML. En este capítulo, se describen cada uno de los diagramas que conforman este lenguaje y su uso en el modelado de sistemas. Comprender mejor su significado e importancia ayudará a considerar el uso del lenguaje para describir requerimientos de seguridad en las distintas etapas del desarrollo de sistemas. Por último, se mencionan los lenguajes de programación más utilizados en la actualidad así como sus principales características con el objeto de conocer las ventajas y desventajas de cada uno de ellos.

1.1 Situación actual del desarrollo de software.

En la sociedad moderna, el uso de herramientas de cómputo se ha convertido en un recurso esencial para el desarrollo de diversas áreas de conocimiento ya sea educación, salud, finanzas, economía, entretenimiento, música, entre otras.

El desarrollo de sistemas de software y el amplio uso de redes de computadoras permiten disponer de información detallada en el momento preciso, sin embargo, con el nacimiento de esta nueva forma de compartir información, también se han desarrollado nuevas formas de espionaje y robo de información también conocido como 'espionaje informático'. Existen diferentes razones por las que una persona u organización desea competir por tal información, desde intereses académicos, diversión pero, sobre todo, intereses económicos. Existen herramientas de software y técnicas específicas que detectan y explotan errores o '*bugs*' en los sistemas para de esta manera ganar acceso a la información. Los errores de software existentes son diversos y van desde problemas de configuración hasta errores de diseño o implementación. A lo largo de los últimos

años, este problema ha crecido debido al uso masivo de Internet, ya que como es bien sabido, posibilita el acceso a información desde cualquier parte del mundo, así como también la posibilidad de ataques y accesos indebidos, elevando la probabilidad de tener pérdidas de distintas índoles tales como monetarias, de imagen, entre otras.

Para evitar el acceso indebido y el robo de información se requieren la implementación de diversas técnicas informáticas de control de acceso, autenticación y cifrado, las cuales reducen la probabilidad de estos robos de información. De estas necesidades surge el concepto de seguridad informática o seguridad de la información.

En la actualidad existe un gran número de aplicaciones en uso, ya sea de propósito general o específico. Actualmente se desarrollan gran cantidad de sistemas en áreas donde anteriormente no se requería de software alguno. De todo esto nace la necesidad del desarrollo de sistemas seguros que permitan proteger la información de manera adecuada, pero ¿cuál es la situación actual del desarrollo de software en el ámbito de la seguridad?

El desarrollo de sistemas también tiene sus propios problemas. Todas las compañías dedicadas al desarrollo de software reciben una fuerte presión del mercado para entregar tecnología nueva y competente en tiempos muy cortos. Hacer las cosas cuidadosamente es tardado y costoso, por lo que es muy común realizar software de prisa y sin demasiada inversión de tiempo para la realización de pruebas. Esta manera de desarrollar sistemas ha dado como resultado una red global de miles de millones de sistemas vulnerables. La tendencia lógica es que esta situación crezca en los próximos años. Existen múltiples factores que provocan este incremento, pero básicamente se reconocen tres, que son determinantes e inherentes al crecimiento tecnológico, los cuales son (HogLung & McGraw, 2004):

- **Complejidad**: Cada día los sistemas son más complejos y esto conlleva a un incremento en la cantidad de errores de programación. Se estima que el número de errores por miles de líneas de código (KLOC – 1000 lines of code) está entre 5 y 50. Un sistema sujeto a pruebas rigorosas de calidad contiene alrededor de 5 errores por KLOC. Un sistema al que no se le aplican pruebas de seguridad, como es el caso de gran parte de software comercial, tendrá alrededor de 50 errores por KLOC. Con la tendencia en la complejidad de los sistemas, este número irá en aumento ya que la cantidad de errores de programación es directamente proporcional a la cantidad de líneas de código.

El amplio uso de lenguajes de programación de bajo nivel como son C o C++, que no ofrecen protección contra ataques comunes como son los *'buffer overflows*' complican más el escenario. En teoría, podemos decir que un programa pequeño se puede probar y analizar para que esté libre de problemas de seguridad, pero para los sistemas complejos, que es el caso de la mayoría de los sistemas actuales, se vuelve prácticamente imposible. Por otro lado, crear software seguro tampoco es fácil y muchos desarrolladores que emplean mecanismos de seguridad, no tienen un conocimiento a fondo de la materia. En la gran mayoría de casos, los problemas de seguridad no comienzan rompiendo mecanismos o protocolos criptográficos, sino explotando la manera en cómo se implementan. No basta con utilizar mecanismos de seguridad e insertarlos en un sistema; en la planeación, diseño e implementación se debe tomar en cuenta estos aspectos de una manera coherente y constante. En el desarrollo de sistemas, generalmente se analiza cuidadosamente todos los aspectos funcionales del mismo, dejando los requerimientos de seguridad en un segundo plano. Cabe resaltar que, hasta el día de hoy, no existe una metodología completa y coherente en la construcción de sistemas grandes y complejos donde se tomen en cuenta aspectos de seguridad. No olvidemos que un sistema es tan seguro como su parte más débil.

- **Extensibilidad**: Los sistemas extensibles son aquellos que permiten la ejecución de código móvil. Hacen uso de máquinas virtuales que mantienen la seguridad de los tipos de datos y se encargan de verificar la seguridad de los accesos en tiempo real, permitiendo ejecutar código móvil no confiable únicamente bajo ciertos permisos. La dificultad de estos sistemas radica precisamente en su arquitectura extensible, ya que se vuelve más complicado analizar la seguridad de un sistema si el código móvil que lo extiende no se conoce y en algunos casos tampoco se puede predecir.

- **Conectividad**: La creciente conectividad de computadoras a través de redes de comunicación ha incrementado el número de posibles ataques así como la facilidad con la que se pueden realizar. Cada vez resulta menos complicado disponer de información que explica detalladamente la manera con la que se puede explotar alguna vulnerabilidad, además de que es posible programar ataques que no requieren de intervención humana alguna. Debemos de considerar la interacción del sistema con adversarios motivados que actúan de manera independiente. Como ha sucedido con muchos '*worms*' bien conocidos, la vulnerabilidad de un servicio se puede propagar a través de la misma red a todas las máquinas gracias a la conectividad; una pequeña falla

de un sistema se convierta en un problema a gran escala que a su vez puede originar grandes pérdidas económicas. Si a todo esto agregamos toda la variedad de nuevos dispositivos especializados que requieren de la misma interconexión, el panorama futuro no resulta muy alentador.

Actualmente podemos mencionar básicamente dos métodos utilizados para la creación de software seguro. La estrategia tradicional es mediante la penetración y la aplicación posterior de parches, dando por hecho que un sistema contiene vulnerabilidades. Esto se hace contratando equipos especializados en la penetración de sistemas de cómputo, también conocidos como equipos tigre. Para muchos sistemas, este método no es el ideal, ya que cada penetración que logra aprovechar una nueva vulnerabilidad puede causar un daño significativo; esto sin mencionar que los administradores de sistemas generalmente dudan en aplicar parches debido a que muchas veces aplicarlos ocasiona interrupción de los servicios, además de que los mismos parches pueden contener amenazas de seguridad. Por otro lado, los parches resultan costosos para el equipo de desarrollo y provocan desconfianza en el usuario final.

El otro método utilizado para el desarrollo de software seguro es mediante su formalización a través de modelos matemáticos. Este método es muy utilizado en el diseño de componentes de seguridad crítica como es el caso de los protocolos de seguridad. El uso de éstos modelos ayuda a establecer los requerimientos de seguridad a nivel de especificación y mediante el uso de una buena formalización, podemos programar y aplicar diferentes pruebas al modelo. Obviamente, los resultados satisfactorios de un diseño basado en este modelo no garantizan que un sistema sea absolutamente seguro ya que los resultados solo corresponden a pruebas realizadas sobre una abstracción de la realidad, y en el mundo físico los atacantes siempre intentarán ir más allá de estos modelos. De cualquier forma, un análisis de seguridad basado en modelos siempre es útil ya que nos da la oportunidad de contemplar y evitar ciertos ataques y de hacer que el esfuerzo requerido para lograrlos, aumente.

Ninguno de los dos métodos mencionados es adecuado para aplicarse en el desarrollo de sistemas industriales y comerciales. El primero resulta muy costoso y no tiene manera de garantizar un buen nivel de seguridad ya que los parches se aplican a una base ya diseñada que seguramente nunca contempló requerimientos de seguridad. Por su parte el segundo método puede funcionar mejor, pero requiere de una alta especialización en la materia, cosa que generalmente no se contempla en el desarrollo de software comercial.

De acuerdo con todo lo anterior, para el desarrollo seguro de sistemas se necesita de un método adicional que integre análisis de requerimientos de seguridad con métodos de desarrollo estándar, que sea fácil de aprender y utilizar, y en el que sea posible contemplar los aspectos de seguridad en cada etapa del desarrollo de software, desde el diseño, implementación, prueba y puesta en funcionamiento. También se requiere de una notación particular precisa para el análisis e implementación de especificaciones ya que esto permitiría la creación de herramientas automatizadas de análisis de seguridad. Por si todo esto no fuera poco, la falta de recursos en tiempo y capacitación de los desarrolladores obliga a llenar estos vacíos ocupando herramientas ya creadas y utilizadas en el desarrollo de software. También es necesario examinar cuidadosamente los límites de los componentes especificados que no forman parte del sistema pero que mantienen interacción con él, haciendo ciertas suposiciones en el contexto del mismo. Por último, y no por ello menos importante, existe un problema conocido como problema de refinamiento. Se dice que una implementación es un refinamiento de su especificación, y una implementación de una especificación segura no necesariamente será segura, lo cual no es deseable, por lo que se requiere que las formalizaciones de requerimientos de seguridad se conserven por refinamiento (Jürgens, 2005).

Con el objetivo de solucionar los problemas mencionados, Jan Jürgens propone un nuevo método de desarrollo de sistemas basado en un modelo utilizando el lenguaje UML. La idea de un desarrollo basado en modelos comienza construyendo precisamente el modelo del sistema. Éste se crea utilizando la intuición humana y generalmente corresponde a un modelo abstracto. En un segundo paso, se deriva la implementación del sistema, ya sea de manera automática utilizando generación de código, o manual, en la que de cualquier forma uno puede generar secuencias de prueba para verificar que el código siga las especificaciones establecidas en el modelo. El objetivo de esta propuesta es el de incrementar la calidad del código de sistema manteniendo el costo de implementación y los tiempos de entrega. Para el caso de sistemas con seguridad crítica, este método permite considerar requerimientos de seguridad desde las etapas iniciales y durante el ciclo de vida del desarrollo de sistema; de esta manera es posible analizar la seguridad del sistema en las etapas de diseño y posteriormente verificar que el código implementado es seguro generando secuencias de prueba creadas a partir del modelo.

A continuación se abordan conceptos básicos de seguridad informática, ya que son necesarios para comprender gran parte de éste trabajo. Debido a que el campo de la seguridad es muy amplio y complejo, sólo se abordan aquellos conceptos que se consideran indispensables para comprender en lo general este trabajo.

1.2 Conceptos Básicos de Seguridad Informática.

La seguridad informática se refiere a la protección de la información en contra del acceso, transferencia, modificación o destrucción no autorizada, ya sea de manera accidental o intencional. De acuerdo al ISO7498-2[1], la seguridad informática se basa en 5 servicios que son los siguientes:

- **Autenticación:** Permite determinar si una entidad es quien dice ser. Su implementación generalmente se basa en algo que se sabe, se es o se tiene. Se aplica tanto a las entidades como a la información, por lo que se divide en autenticación de entidades y autenticación de origen de datos.
- **Control de acceso:** Este servicio proporciona protección en contra del uso no autorizado a los recursos, especificando niveles de acceso a cada entidad autenticada. Este servicio también se le conoce como servicio de Autorización.
- **Confidencialidad:** Servicio que garantiza el acceso y comprensión semántica de la información sólo a las partes autorizadas.
- **Integridad de datos:** Servicio que evita la alteración no autorizada de datos. No puede garantizar que no se altere la información, pero si detectar un cambio y reportarlo a una entidad apropiada. El servicio de integridad de datos intentará corregir o recuperar los cambios.
- **No-repudio:** Servicio encargado de evitar la negación de acciones realizadas por alguna entidad. Proporciona evidencia que se puede almacenar para posteriormente presentarse a una tercera parte para resolver alguna disputa originada electrónicamente.

La criptografía es la disciplina que abarca los principios, medios y métodos para la transformación de datos, de tal manera que sea posible implementar la mayoría de los servicios de seguridad antes descritos. Para implementar estos servicios existen diferentes mecanismos de seguridad. Gran parte estos mecanismos basan su implementación mediante el uso de algoritmos criptográficos.

Es importante aclarar que no existe un mecanismo de seguridad capaz de proveer todos los servicios de seguridad, por lo que generalmente se combinan al momento de implementar sistemas de seguridad integrales. Los mecanismos definidos en el ISO 7498-2 son los siguientes:

[1] Information processing systems – Open Systems Interconnection – Basic Reference Model

- **Mecanismos de cifrado:** Proporciona el servicio de confidencialidad y también se puede utilizar para implementar servicios de autenticación y de verificación de integridad, así como complemento en otros mecanismos de seguridad. En su aplicación más común, garantiza que la información no es inteligible para los usuarios, entidades o procesos no autorizados.

El proceso de cifrado consiste en la transformación de texto claro a texto cifrado. Hace uso de una llave o clave que permite variar la función de transformación. Al proceso de transformación de texto cifrado a texto claro se le conoce como descifrar. Basado en el tipo de llave, existen dos tipos de cifrado que son: el cifrado simétrico y el cifrado asimétrico.

En el cifrado simétrico o criptografía de llave privada, la llave utilizada para cifrar la información es la misma requerida para descifrar. Los mecanismos de cifrado simétrico generalmente se dividen en cifrado por flujo y cifrado por bloque. Los cifrados por flujo se encargan de realizar la transformación bit a bit, mientras que los cifrados por bloque toman como entrada a cada iteración del algoritmo bloques de n-bits. Debido a que la seguridad de este tipo de cifrado se basa en el ocultamiento de la llave privada, generalmente requiere de un proceso de administración, así como un proceso de acuerdo e intercambio de llave, lo cual, en muchas ocasiones complica su implementación.

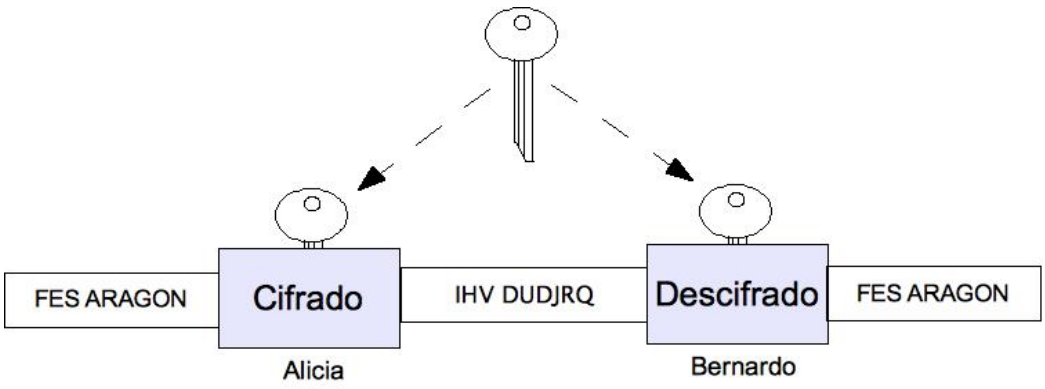


Figura 1: Cifrado/Descifrado utilizando criptografía de llave privada o cifrado simétrico

En el cifrado asimétrico o criptografía de llave pública, se hace uso de un par de llaves distintas entre sí, conocidas como llave privada y llave pública. Ambas llaves se relacionan de tal forma que si se utiliza una de ellas para cifrar un mensaje, la otra servirá para descifrarlo y viceversa. Como su nombre lo indica, la llave privada se deberá conservar en secreto mientras que la llave pública estará disponible para todo el mundo. Su seguridad se basa en el hecho de que el conocimiento de una de las llaves no puede derivar en el conocimiento de la otra. La mayor ventaja que ofrece este tipo de cifrado es que no requiere de un mecanismo de acuerdo de llaves, lo que lo hace más sencillo, sin embargo no resulta en algoritmos tan rápidos como los algoritmos de cifrado simétrico.

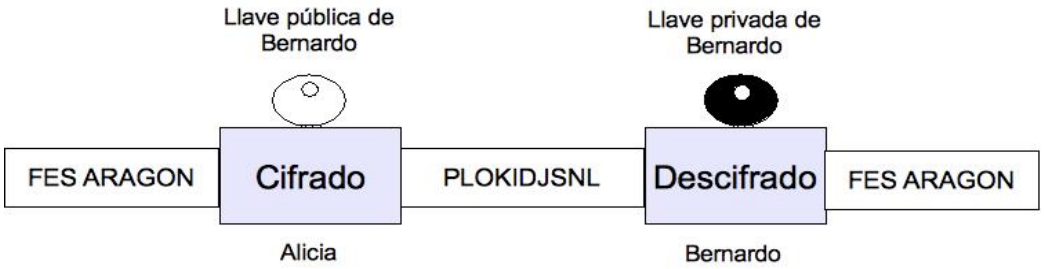


Figura 2: Diagrama de funcionamiento del cifrado de llave pública o cifrado asimétrico

- **Mecanismos de integridad de datos:** Estos mecanismos deben considerar la integridad de una unidad de datos única y la integridad de un flujo de unidades de datos. La integridad de una unidad de datos incluye dos procesos, uno para la entidad transmisora y otro para la entidad receptora. El emisor agrega una cantidad que está en función de los datos. El receptor genera una cantidad correspondiente y la compara con la recibida para determinar si los datos se modificaron en el camino. El mecanismo por sí mismo no garantiza la integridad de los datos, sólo se encarga de verificarla. Una manera de proteger la integridad de datos es utilizando el mecanismo de cifrado, de esta manera se garantiza la integridad de los datos y la confidencialidad. Otra opción es mediante el uso de firma digital, en cuyo caso se garantiza la integridad y el no-repudio. Si solo se desea integridad de datos sin los servicios de confidencialidad y no-repudio, se hace uso de las funciones criptográficas hash como son MD5 y SHA-1.

- **Mecanismos de firma digital:** La firma digital o electrónica sigue el concepto de la firma autógrafa. Define los procedimientos para firmar una unidad de datos y para verificar la unidad de datos firmada. Generalmente se implementa mediante cifrado asimétrico. Los esquemas de firma digital normalmente utilizan dos algoritmos, uno para la firma de datos, utilizando la llave privada del firmante, y uno para verificar la validez de las firmas en el que se hace uso de la llave pública del usuario. La firma digital se envía junto con los datos ordinarios.

  Las firmas digitales proporcionan los servicios de autenticación de datos, no repudio y, al combinar la firma digital con funciones hash, también proporcionan la verificación de integridad de datos. Para la utilización de firmas digitales se requiere tener confianza sobre las llaves públicas utilizadas y para ello se necesita crear y utilizar una infraestructura de llave pública (PKI, por las siglas en inglés de *Public Key Infrastructure*) para que sea posible emitir los Certificados Digitales de llave pública y su correspondiente administración de llaves.

  El proceso general de firma digital es como sigue: teniendo un mensaje que deseamos firmar, obtenemos el valor hash del mensaje y lo ciframos utilizando la llave privada del firmante. El proceso de verificación consiste en calcular el valor hash del mensaje y utilizar la llave pública del firmante para descifrar la firma digital. Si el valor hash del mensaje es igual al valor de la firma digital descifrada, podemos asegurar que el documento ha sido firmado por la entidad correspondiente, además de que se garantiza el servicio de integridad de datos y no repudio. El siguiente diagrama muestra claramente ambos procedimientos, el de firma y el de verificación.

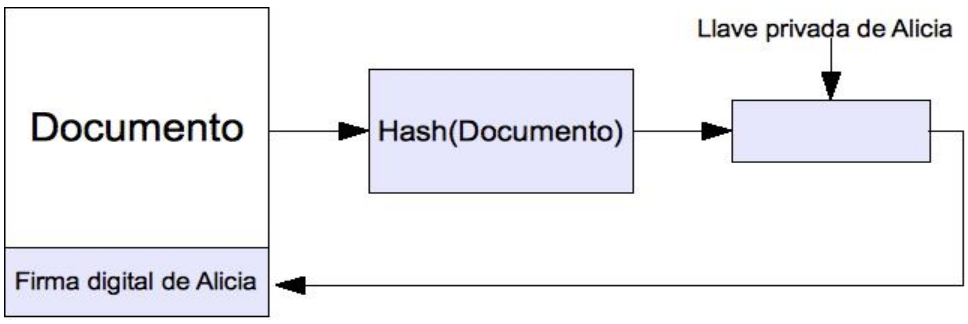


Figura 3: Procedimiento de la firma digital.

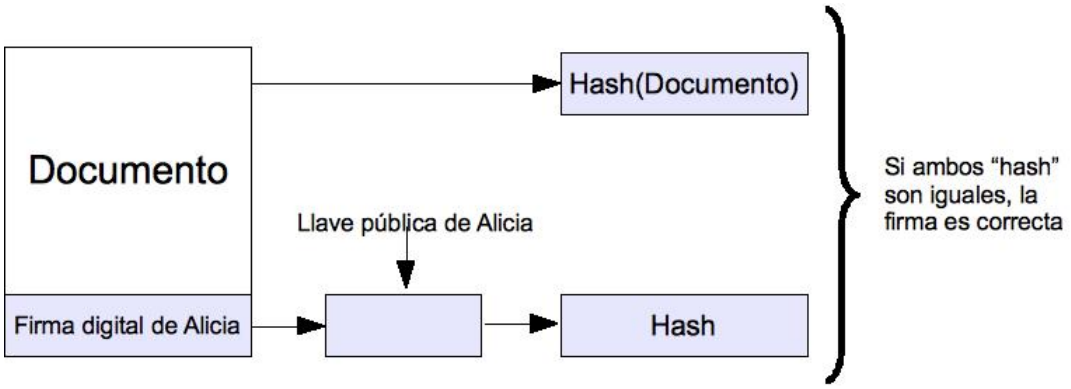


Figura 4: Proceso de verificación de firma digital.

- **Mecanismos de control de acceso:** Mediante este mecanismo se otorga o prohíbe el acceso a algún dato o acción, previa autenticación. Estos mecanismos garantizan que todos los accesos a los recursos son autorizados, es decir, solo es posible acceder a los objetos por entidades autorizadas y con un nivel de acceso definido.

  La efectividad de los controles de acceso se basa en dos premisas:
  - La correcta identificación o autenticación de entidades, donde debe resultar imposible adquirir los derechos de acceso de otro usuario.
  - La protección contra modificación no autorizada de la información que especifica los derechos de acceso de cada entidad como es el caso de las listas de control de acceso.

- **Mecanismos de intercambio de autenticación:** Proporciona autenticación de la entidad par. Si el mecanismo no logra autenticar la entidad de forma correcta, se procederá rechazando o terminando la conexión. Existen diversos mecanismos de intercambio de autenticación como es el caso del sistema Kerberos.

- **Mecanismos para rellenado de tráfico:** Se utiliza para proporcionar varios niveles de protección en contra del análisis de tráfico. Estos mecanismos se encargan de mantener el tráfico constante de tal manera que nadie puede obtener información observándolo. Este mecanismo solo puede ser efectivo si el rellenado de tráfico se protege mediante un servicio de confidencialidad.

- **Mecanismos de control de ruteo:** Estos mecanismos hacen posible elegir una ruta específica para el envío de datos a través de la red, evitando el paso por nodos indeseables.

- **Mecanismos de certificación notarial:** Las propiedades acerca de los datos transmitidos entre dos entidades deben asegurarse mediante un mecanismo de certificación notarial. Éste es proporcionado por una tercera parte, la cual es una entidad confiable para ambas partes y capaz de proporcionar la información de manera verificable.

  Un certificado digital es un documento que identifica de manera única, información sobre cierta entidad. Contiene la llave pública de la entidad así como información de identificación adicional que es firmada digitalmente por una tercera parte también conocida como Autoridad Certificadora (CA). Un certificado digital debe implementar en su estructura de datos el estándar X.509[2]. También se utiliza para resolver problemas de administración de llaves. La figura 5 muestra un ejemplo del contenido básico de un certificado digital, así como una representación del algoritmo utilizado para verificar la autenticidad del mismo. En el ejemplo de la figura, Carlos actúa como Autoridad Certificadora, es decir, si nosotros no conocemos de manera directa a Alicia, confiamos en que 'Carlos' la conoce y ha verificado que la información y llave pública presentadas en el certificado, realmente corresponden a ella.

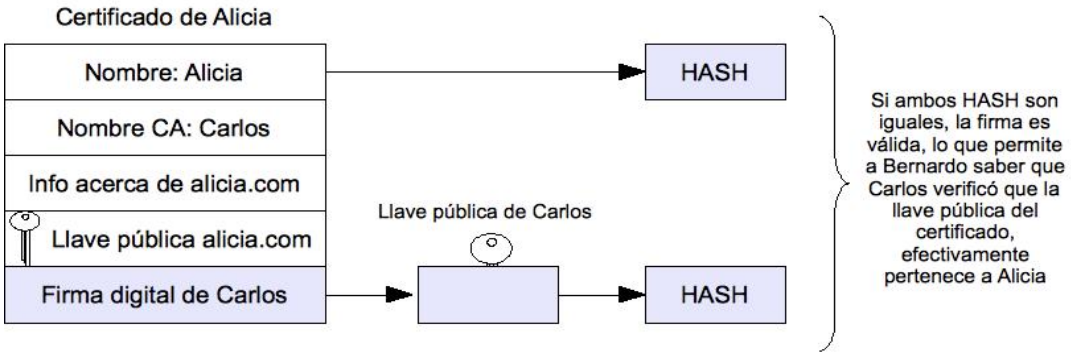


Figura 5: Verificación de una entidad, mediante el uso de un certificado digital.

[2] Estándar de la ITU-T (International Telecommunication Union) para una infraestructura de llave pública (PKI). Especifica formatos estándar de certificados de llave pública, listas de revocación de certificados y un algoritmo para la validación de la ruta de certificación.

En la Figura 5, confiamos en la Autoridad Certificadora (Carlos) debido a que lo conocemos y sabemos que se trata de una entidad confiable, sin embargo, si la Autoridad Certificadora de cierto certificado no existe en la lista de autoridades confiables, es posible crear una cadena de certificados hasta llegar a un certificado raíz que corresponde a una Autoridad Certificadora confiable para nosotros. Debido a que este concepto puede presentar cierta complejidad, se ilustra de manera detallada en la Figura 6.

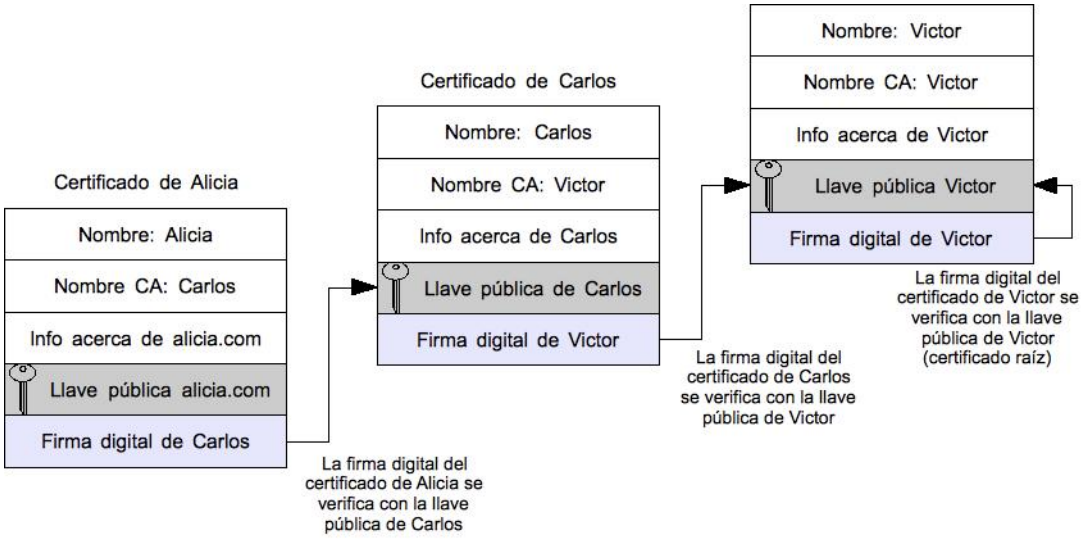


Figura 6: Cadena de certificados.

Los certificados digitales son muy utilizados en los servicios de Internet que requieran de una conexión segura, normalmente usados con el protocolo SSL.

Las Autoridades Certificadoras también son responsables de revocar aquellos certificados que no han sido verificados correctamente o que no cumplen con los requerimientos y políticas especificadas por la Autoridad Certificadora. Este proceso de revocación incluye el mantenimiento de una lista de revocación de certificados o CRL que incluyen el certificado, así como un número de serie.

Una vez descritos los conceptos de seguridad necesarios para la compresión de un desarrollo seguro de software, necesitamos abordar otro tema que parece totalmente distinto, sin embargo es indispensable para el desarrollo de sistemas en la actualidad. Todo desarrollo de software pasa por distintas etapas que van desde el análisis hasta su implementación. Debido a la complejidad que se presenta en estas etapas y a la necesidad de adoptar un medio de

comunicación común que permita describir el sistema a distintos niveles, se ha adoptado el uso de una herramienta conocida como UML (por las siglas en inglés de *Unified Modelling Language*). Ya que en este trabajo se pretende introducir un proceso de desarrollo seguro de sistemas basado en una extensión de este lenguaje, resulta necesario entender su uso, qué describen los diagramas y porqué se ha decidido adoptar esta herramienta como un medio de abstracción más en el desarrollo seguro de sistemas. La sección siguiente está dedicada completamente al lenguaje de modelado UML.

1.3 El Lenguaje de Modelado Unificado (UML).

UML es un lenguaje visual para la captura de patrones y diseño de software. Se utiliza para expresar relaciones, comportamientos e ideas de alto nivel en una notación fácil de aprender y eficiente al escribir. Basa sus orígenes en tres métodos de modelado distintos: el modelo de Booch propuesto por Grady Booch, el Object Modeling Technique cuyo autor es James Rumbauht y Objectory de Ivar Jacobson. Los tres autores comenzaron a trabajar en la primera versión de UML en 1994 hasta que en el año de 1997 se liberó la versión 1.1 del lenguaje, aceptada por el grupo OMG[3]. Desde ésta primera liberación ha tenido varias revisiones llegando hasta la versión 2.1 actual.

UML es un lenguaje y como tal está formado por una sintaxis y una semántica, por lo que existen reglas que rigen la organización de sus elementos y su significado. Un modelo UML está formado por uno o más diagramas. Cada diagrama representa entidades y sus relaciones entre sí. Es común representar una entidad con múltiples diagramas, cada uno de los cuales representa una vista o interés en particular.

La versión 2.0 de UML divide los diagramas en dos categorías que son los diagramas estructurales y los diagramas de comportamiento. Los diagramas estructurales se utilizan para capturar la organización física del sistema, mientras que los diagramas de comportamiento se enfocan en el funcionamiento de los elementos en el sistema.

[3] OMG (Object Management Group): Es un consorcio encargado de la producción y mantenimiento de especificaciones de la industria de la computación para el desarrollo de aplicaciones portables.

A continuación se describen brevemente cada uno de los diagramas que forman parte de la versión 2.1, sin embargo, es importante mencionar que el lenguaje solo es una herramienta gráfica utilizada para el modelado de sistemas, lo más importante en el desarrollo de sistemas es el proceso de desarrollo. El proceso actual más utilizado es el proceso de desarrollo unificado también conocido como RUP. En el diseño real de sistemas no se hace uso de todos los diagramas mencionados, si no que solo se recomienda utilizar aquellos que proporcionan información útil a cada uno de los '*stakeholders*'[4]. El objetivo del desarrollo de sistemas no es la utilización de todos y cada uno de los diagramas UML, sino sólo aquellos que nos proporcionan información importante acerca de nuestro sistema.

Comenzamos describiendo los diagramas estructurales:

Diagramas de clases.

Es uno de los diagramas más utilizados en el modelo conceptual y en el modelo de dominio[5]. Se utilizan para mostrar las clases, interfaces y las relaciones estáticas entre sí. Las clases se representan mediante rectángulos divididos básicamente en 3 compartimentos donde se especifican el nombre de la clase, sus atributos y sus operaciones.

Las relaciones entre las clases son distintas y debido a esto existen distintas maneras de representarlas, En UML, las relaciones entre clases son 5 y son: dependencia, asociación, agregación, composición y generalización (Guide).

- **Relación de dependencia:** Es aquella en la que un elemento (clase cliente) utiliza o depende de otro elemento (clase proveedora). Es una relación temporal en la que la clase cliente puede: utilizar la clase proveedora como parámetro de una de sus operaciones como variable local o envía mensajes a la clase proveedora.
- **Relación de asociación:** Una relación de asociación en términos de programación orientada a objetos es una relación "has a" o "tiene un" y se representa por un miembro instancia de la clase.
- **Relación de agregación:** En muchas ocasiones se considera un tipo especial de relación de asociación y se utiliza cuando un clasificador (clase) "es parte de" otro clasificador (clase). Está relacionada con el término de composición. En java se representa como un miembro de instancia aunque la relación entre las clases es más fuerte que la relación de asociación ya que indica que un objeto es parte de otro.

[4] Stakeholder (Interesado): Se dice de todas aquellas personas o entidades que afectan o son afectadas por el proyecto, ya sea de forma positiva o negativa.
[5] Modelo que captura los tipos de objetos más importantes en el contexto del sistema a desarrollar.

- **Relación de composición:** Es una relación muy similar a la de agrupación sin embargo el tiempo de vida de ambos objetos tiene mucha similitud, es decir, un objeto no puede existir si no existe el otro. En el lenguaje de programación Java también se representa mediante un miembro instancia de la clase.
- **Relación de generalización:** Es un tipo de relación del tipo padre/hijo, es decir, el elemento hijo se basa en el elemento padre. En términos de programación orientada a objetos se representa por una relación "es un" o "is a" y tiene que ver con la relación de herencia.

En la Figura 7 se muestra un diagrama de clases y las posibles relaciones existentes entre ellas.

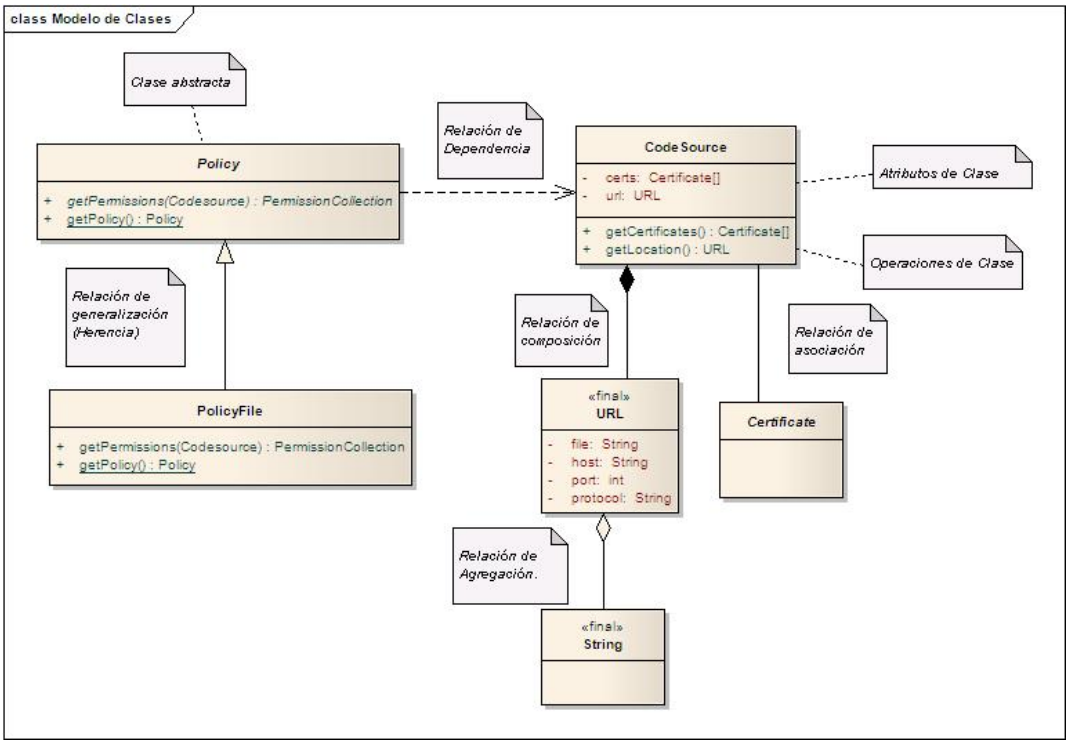


Figura 7: Representación de clases y sus relaciones en UML.

Otro de los elementos importantes en un diagrama de clases es la interfaz. Una interfaz es un clasificador UML con declaraciones de propiedades y operaciones pero sin su implementación. La diferencia entre una interfaz y una clase abstracta radica en que anteriormente, cuando no existían las interfaces y se pretendía encapsular cierta funcionalidad en una clase abstracta, era indispensable seguir reglas estrictas de herencia aún cuando por cuestiones de encapsulación no era necesario. El objetivo de una interfaz es encapsular funcionalidad independientemente de las diferencias que existen entre las clases que representan entidades en un sistema.

En la Figura 8 se muestran las dos maneras de representar una interfaz.

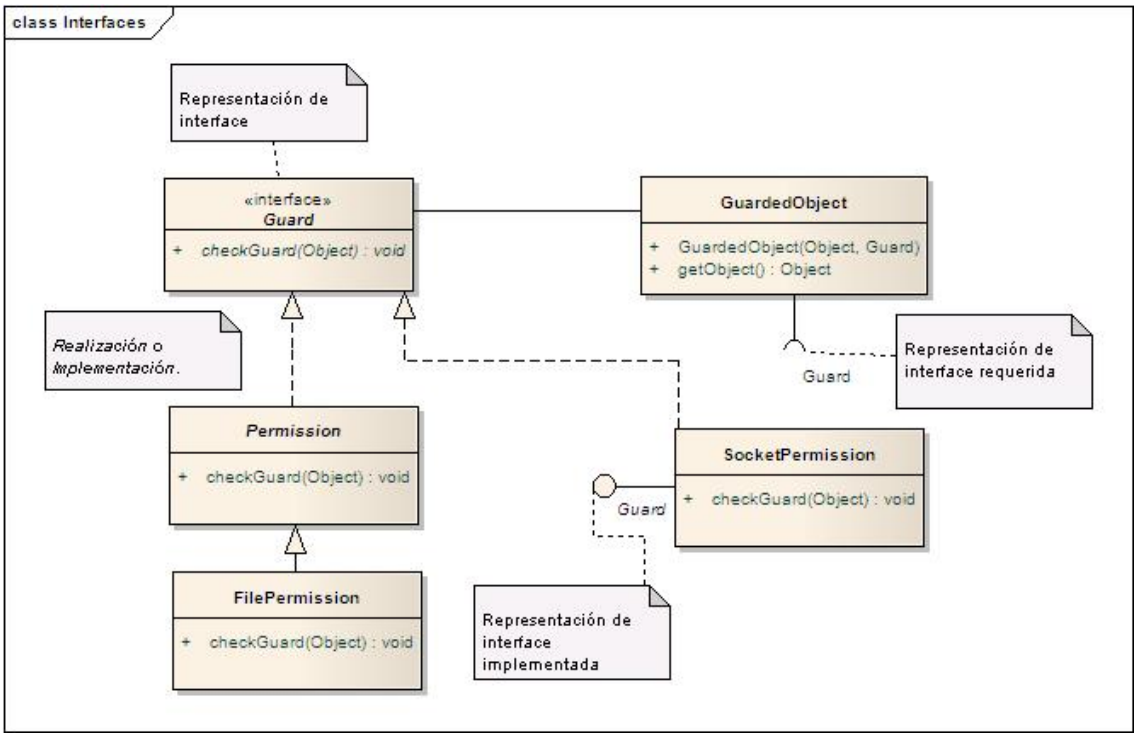


Figura 8: Representación de interfaces en UML.

Diagrama de componentes.

Son muy útiles en sistemas grandes, donde lo más conveniente es utilizar sub-sistemas más pequeños y administrables llamados componentes. Estos diagramas nos permiten modelar un sistema a un nivel más alto que los diagramas de clases, definiendo a su vez cada una de las interfaces proporcionadas y requeridas. Usualmente un componente se implementa mediante una o más clases u objetos en tiempo de ejecución. En la práctica, los diagramas de componentes son similares a los diagramas de paquete ya que definen límites y se utilizan para agrupar elementos en estructuras lógicas. La diferencia es que los diagramas de componentes ofrecen un mejor mecanismo de agrupación semántica. En los diagramas de componentes todos los elementos del modelo son privados, mientras que en los diagramas de paquetes solo se despliegan los elementos públicos. Un diagrama de componentes ayuda a representar las dependencias entre componentes software, incluyendo componentes fuente, componentes de código binario y componentes ejecutables. Existen componentes en tiempo de compilación, en tiempo de enlace o en tiempo de ejecución.

Los diagramas de componentes tienen sólo una versión con descriptores, no tienen versión con instancias. Para mostrar la versión con instancias se utiliza un diagrama de implementación o despliegue. Un diagrama de componentes se muestra en la siguiente figura (Figura 9).

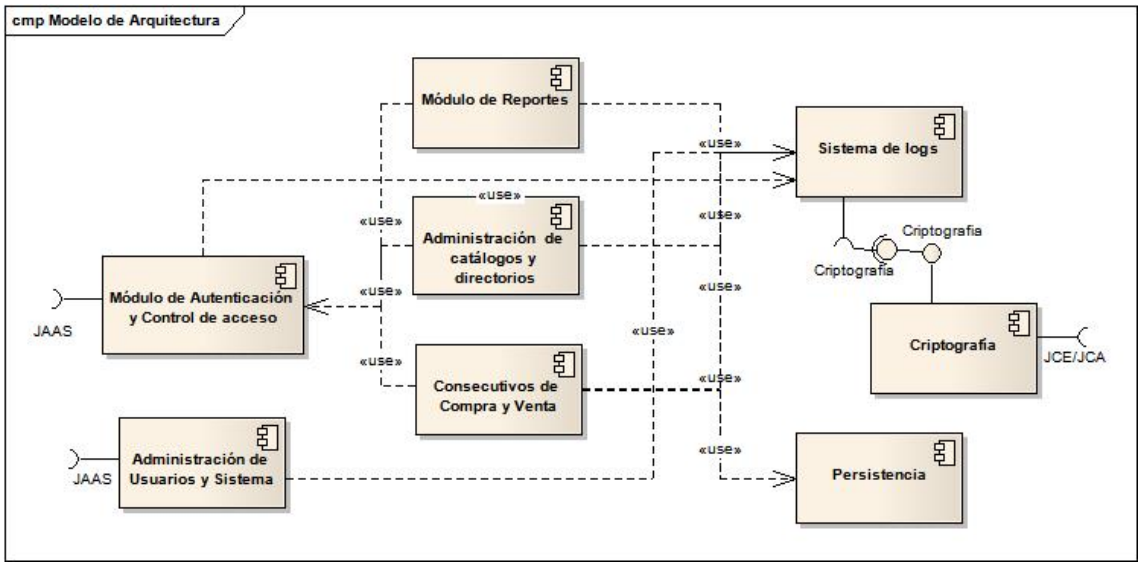


Figura 9: Diagrama de componentes

Diagrama de estructura compuesta.

Estos diagramas son útiles para representar la estructura interna de un clasificador[6] estructurado incluyendo sus puntos de interacción con otras partes del sistema. Una estructura es un conjunto de elementos interconectados entre sí, que existen en tiempo de ejecución y que de manera conjunta proporcionan alguna funcionalidad. Se utilizan para mostrar los detalles internos de un clasificador y así describir los objetos y roles que logran proporcionar la funcionalidad del clasificador contenido.

Este diagrama es semejante a los diagramas de clases solo que en vez de mostrar clases completas representan partes individuales. Es posible agregar conectores para enlazar dos o más partes en una relación de dependencia o asociación.

[6] Un clasificador UML se refiere a una categoría de elementos UML con características comúnes como atributos o métodos. Algunos tipos de clasificadores UML son la clase, componente, interface, nodo, caso de uso, subsistema, etc.

Los puertos definen el punto de interacción entre un clasificador y su ambiente, o entre un clasificador y sus partes internas. Permiten especificar los servicios que el clasificador proporciona o los servicios que requiere de su ambiente. También es posible modelar colaboraciones y ocurrencias de colaboración. Una colaboración describe los roles y atributos que definen una conducta específica de un clasificador. Una ocurrencia de colaboración representa un uso particular de la colaboración para explicar las relaciones entre las propiedades de un clasificador.

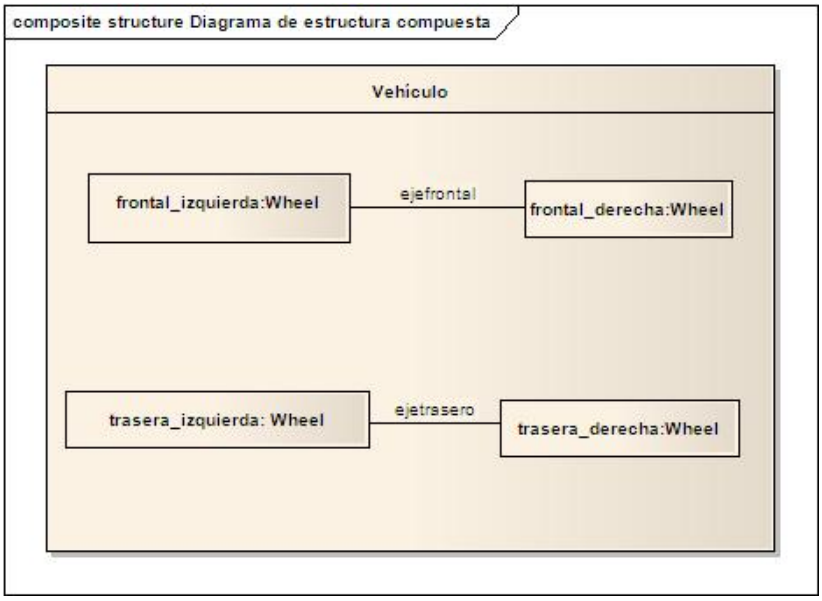


Figura 10: Ejemplo de un diagrama de estructura compuesta de una clase Vehículo.

Uno de los usos más comunes de los diagramas de estructura compuesta es para representar el resultado de la llamada a una operación, generalmente un constructor como se muestra en la figura siguiente:

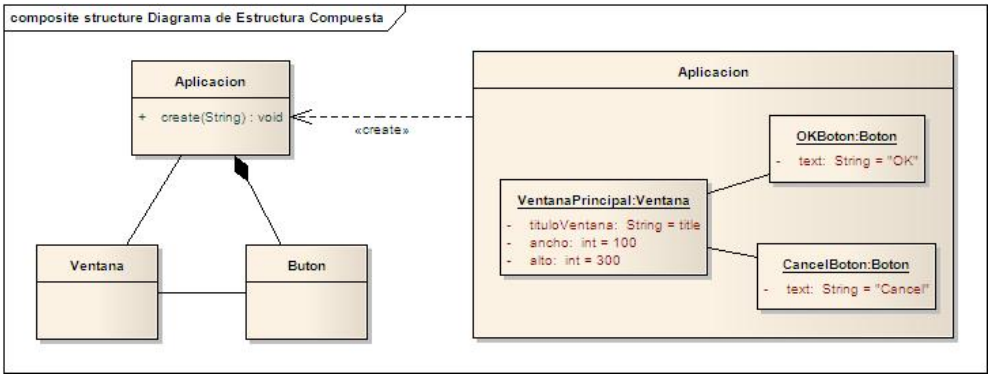


Figura 11: Creación de un diagrama de estructura compuesta que muestra el resultado del constructor de la clase Aplicación.

Diagrama de implementación o ejecución.

Ayuda a representar la vista estática de un sistema, mostrando la configuración en tiempo de ejecución de sus nodos de procesamiento así como sus componentes. Muestran el hardware del sistema y el software instalado. Son muy útiles para mostrar la arquitectura de los sistemas. Los componentes de un diagrama de implementación son los artefactos, nodos, ambientes de ejecución, dispositivos y especificaciones de ejecución.

Los artefactos representan piezas físicas de información relacionada con el proceso de desarrollo de software. Generalmente se utilizan para representar la versión compilada de un componente. En un diagrama se identifica mediante un rectángulo con un símbolo de papel con la esquina doblada colocado en la parte superior derecha del rectángulo. Los artefactos pueden tener propiedades y operaciones que lo manipulan, sobre todo cuando hablamos de un artefacto que representa un conjunto de opciones configurables.

Un nodo es una entidad física que se encarga de la ejecución de los artefactos. Muestran el sitio físico donde una pieza de código se ejecuta así como la manera en la que las diferentes partes del sistema se comunican en tiempo de ejecución. Generalmente se representa mediante una caja 3D con el nombre del nodo escrito dentro de la caja.

Un ambiente de ejecución es un nodo especializado que representa una configuración de software que a su vez contiene tipos específicos de artefactos. Un dispositivo es la especialización de un nodo y representa una máquina física capaz de realizar cierto tipo de cálculo. Un dispositivo se representa mediante un nodo con el estereotipo “*device*”.

La parte más importante de un diagrama de implementación es la relación existente entre los artefactos y los nodos donde se ejecutan. UML tiene diversas maneras de mostrar la implementación. Es posible representarse con solo dibujar el(los) artefacto(s) dentro del nodo; asociarse ambos, artefacto y nodo, mediante una flecha que va desde el artefacto al nodo con el estereotipo “*deploy*”, o simplemente listando todos los artefactos en un compartimento en el clasificador de implementación destino.

La especificación de una puesta en ejecución (deployment specification) es una colección de propiedades que especifican como un artefacto se colocará en el destino de ejecución. Estas especificaciones se representan mediante un rectángulo con el estereotipo “deployment spec” cuyos atributos se refieren a la información de puesta en ejecución requerida.

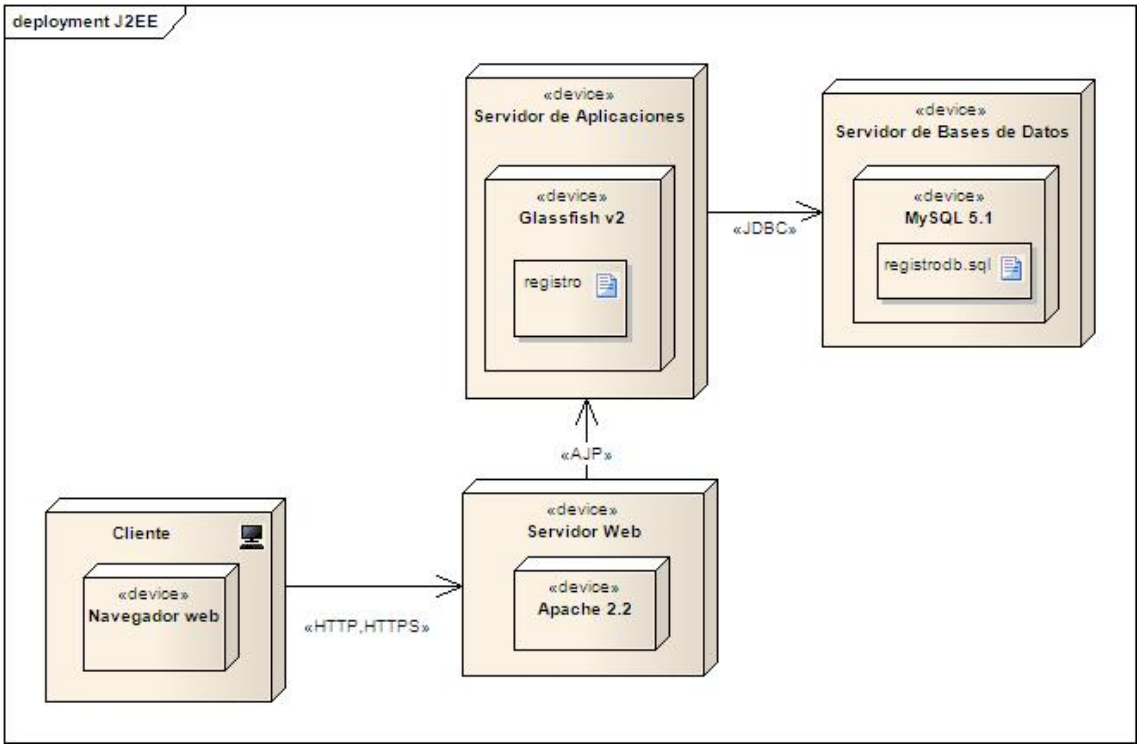


Figura 12: Diagrama de implementación.

Diagrama de paquetes.

Estos diagramas ayudan a mostrar las dependencias entre las partes de un sistema y en ocasiones se utilizan para buscar problemas o determinar el orden de compilación. Los usos más comunes son para organizar diagramas de casos de uso y diagramas de clase.

Un paquete se representa mediante un rectángulo con una pestaña en la parte superior izquierda. Los elementos del paquete se pueden dibujar dentro del paquete, en cuyo caso el nombre del mismo se colocará dentro de la pestaña. La segunda representación utiliza una línea sólida que va del paquete a cada uno de los elementos que contiene. Se coloca un circulo con un signo + dentro, en el extremo correspondiente a la representación del paquete. El nombre del paquete se escribe en el centro del rectángulo que representa al paquete. Esta última notación es muy utilizada cuando se desea mostrar información más detallada de los elementos del paquete.

En UML, la visibilidad de los elementos de un paquete puede ser ’pública’ o ’privada’. Los elementos públicos son aquellos que se pueden utilizar fuera del paquete y se representa mediante un signo de ’+’. Por su parte, los elementos privados son aquellos que solo se pueden utilizar dentro del mismo paquete y se representan con un signo ’-’. Para mostrar la importación de un

paquete se dibuja una línea punteada con una flecha abierta que va del paquete que importa al importado. Esta línea llevará la palabra clave “import”. Los elementos importados tiene una visibilidad pública por default en el paquete que lo importa, sin embargo, si deseamos que un elemento que importamos tenga una visibilidad privada en el paquete que lo importa, utilizamos la palabra clave “*access*” en vez de “*import*”.

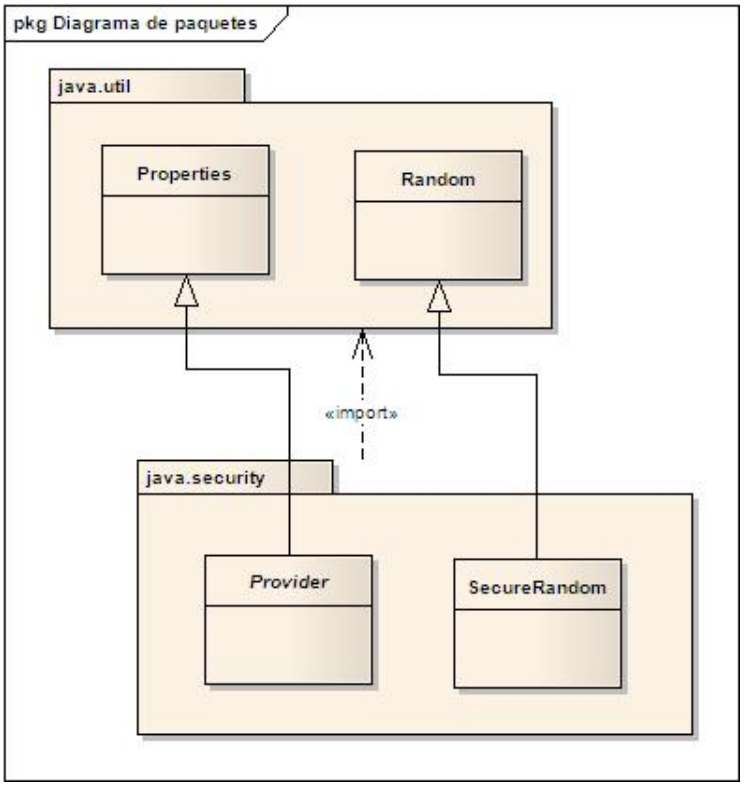


Figura 13: Ejemplo de un diagrama **de paquetes.**

Diagrama de objetos.

Los diagramas de objetos utilizan la misma sintaxis que los diagramas de clases y muestran como se relacionan las instancias de clases en un tiempo determinado.

En seguida, se describen los diagramas de comportamiento.

Diagramas de Actividad.

Son los diagramas UML más utilizados fuera del modelado de software Su función es mostrar el flujo de trabajo desde el punto de inicio hasta el punto final detallando rutas de las decisiones existentes en el progreso de eventos de la actividad. Estos diagramas generalmente se utilizan en el modelado de procesos de negocio, para mostrar la lógica de un caso de uso o de un escenario en particular, o en su caso, para modelar la lógica detallada de una regla de negocio.

Una actividad está formada por acciones y cada acción se refiere a un paso dentro de la actividad donde se realiza cierta manipulación de datos o procesamiento. En UML se representa gráficamente mediante un rectángulo con esquinas curveadas, especificando el nombre de la actividad en la esquina superior izquierda. Es posible mostrar los detalles de cada actividad dentro del rectángulo ya sea utilizando un diagrama de las acciones que la conforman o directamente mediante pseudo-código. Para especificar pre-condiciones y post-condiciones de actividades y acciones utilizamos las palabras clave *precondition*, *postcondition* y *localPrecondition* y *localPostcondition* respectivamente y se colocan dentro del rectángulo de la actividad y/o acción según corresponda.

UML 2.0 define varios tipos de nodos de actividad para modelar diferentes tipos de flujo de información. Hay nodos que representan parámetros de la actividad, nodos objeto que representan datos complejos y nodos de control que se encargan de dirigir el flujo a través de un diagrama de actividad.

En ocasiones resulta útil indicar quién o qué es responsable de realizar un conjunto de acciones dentro de los diagramas de actividad. En estos casos, es posible dividir los diagramas mediante "*particiones de actividad*", que consisten en líneas horizontales o verticales con el nombre de la partición colocada en algún extremo de la misma.

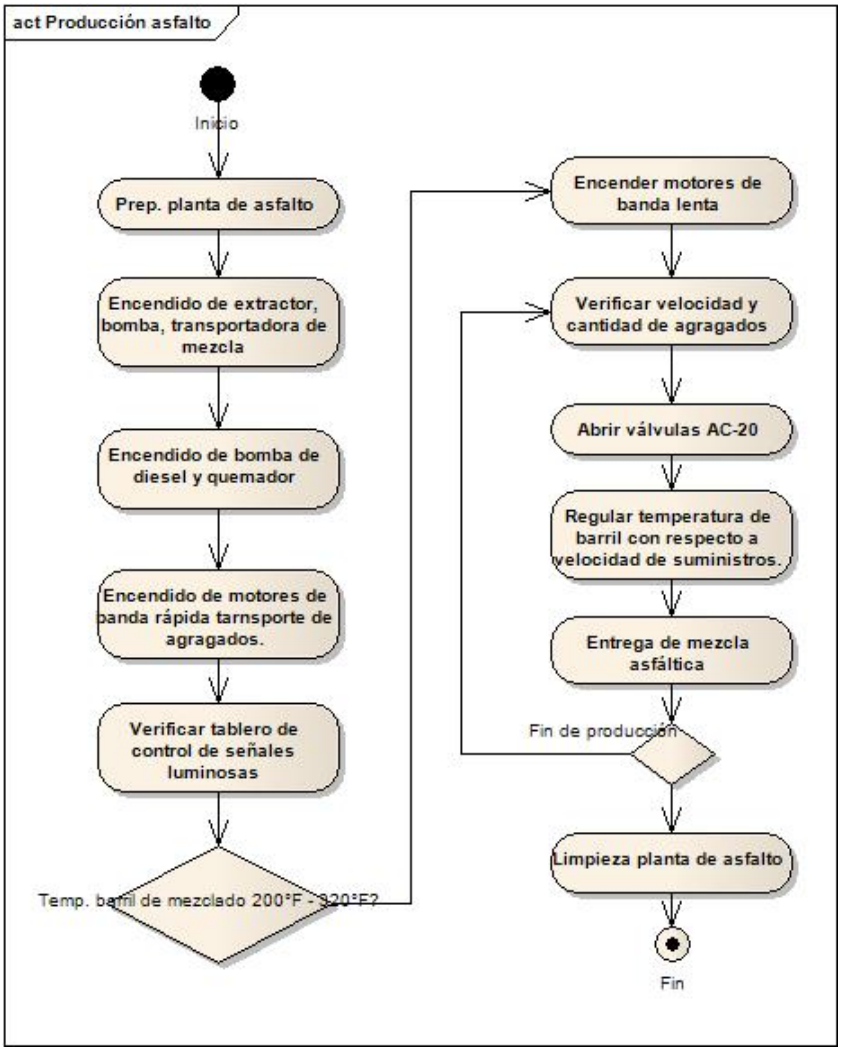


Figura 14: Ejemplo de Diagrama de Actividad.

Diagramas de Interacción (Diagramas de Secuencia).

Los diagramas de interacción corresponden a un conjunto de diagramas encargados de capturar la comunicación existente entre objetos. Se enfocan en representar el paso de mensajes entre objetos y la manera en cómo se conjuntan para realizar alguna funcionalidad. Existen muchas notaciones de los diagramas de interacción, sin embargo los diagramas de secuencia son los más utilizados.

El símbolo básico de estos diagramas es un rectángulo con la palabra clave "*sd*" y el nombre de la interacción colocados en un pentágono en la esquina superior izquierda del rectángulo.

Cada diagrama de interacción tiene participantes y éstos se muestran mediante un rectángulo y una línea por debajo, que en conjunto se le conoce como "*lifeline*" o línea de vida. Esta línea punteada que va del rectángulo hacia la parte baja del diagrama representa el tiempo de vida de cada participante u objeto.

Estos diagramas son ideales para mostrar que objetos se comunican con que otros y que mensajes disparan esas comunicaciones.

La comunicación entre los diferentes "*lifelines*" puede tomar diversas formas como son llamadas a métodos, envíos de señal, creación de una instancia, destrucción de objetos, etc., a lo que en general se le conocen como mensajes. El uso más común de los mensajes es para representar llamadas a métodos y en la sintaxis es posible especificar el paso de parámetros a cada uno.

En los diagramas de secuencia, los mensajes se representan mediante una línea sólida que va desde el transmisor al receptor. Si se trata de un mensaje asíncrono se coloca una punta de flecha abierta del lado del receptor y si es un mensaje síncrono se coloca una punta de flecha sólida cerrada.

Cuando un objeto está involucrado en algún tipo de acción durante un tiempo determinado se representa mediante una ocurrencia de ejecución. Éstas se muestran mediante un rectángulo gris o blanco colocado sobre cada "lifeline".

Es posible colocar notas en un "lifeline" que permitan especificar condiciones que deban cumplirse para que el resto de la interacción sea válida. Estas condiciones se conocen como "invariantes de estado" y típicamente corresponden a expresiones booleanas aunque pueden ser estados UML completos. Un estado booleano se representa mediante la expresión encerrada entre llaves y un estado UML se muestra mediante un rectángulo con esquinas redondeadas.

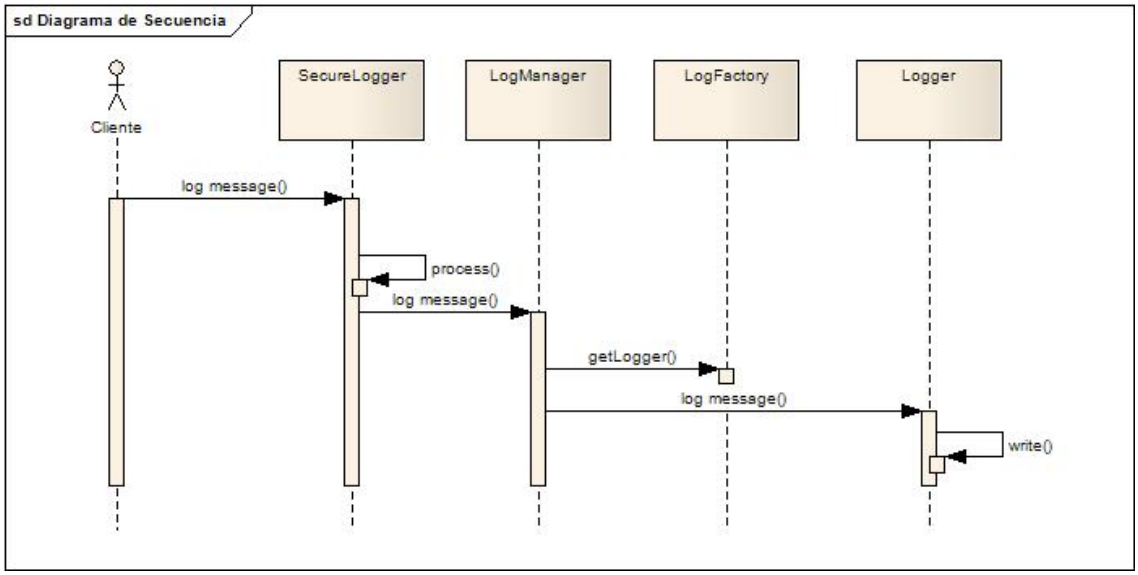


Figura 15: Diagrama de Secuencia (Patrón de desarrollo SecureLogger)

Además de los diagramas de secuencia, existen tipos adicionales de diagramas de interacción con propósitos específicos. Estos diagramas se listan a continuación:

Diagramas de Comunicación.

Son un tipo de diagramas de interacción que se enfocan más en los objetos involucrados en cierto comportamiento que en la naturaleza de los mensajes. También se conocen como diagramas de colaboración. La mayoría de las herramientas UML pueden convertir de manera automática un diagrama de secuencia a un diagrama de comunicación.

Se puede pensar que los diagramas de comunicación son una combinación de los diagramas de clases y los diagramas de secuencia, En un diagrama de comunicación, los objetos se representan mediante un rectángulo (como un diagrama de clases) y las conexiones entre objetos mediante una línea sólida (como un diagrama de secuencia). Cada mensaje tiene un número de secuencia y una flecha pequeña indicando la dirección del mensaje.

A continuación se muestra la interacción de los objetos del diagrama de secuencia de la en un diagrama de comunicación.

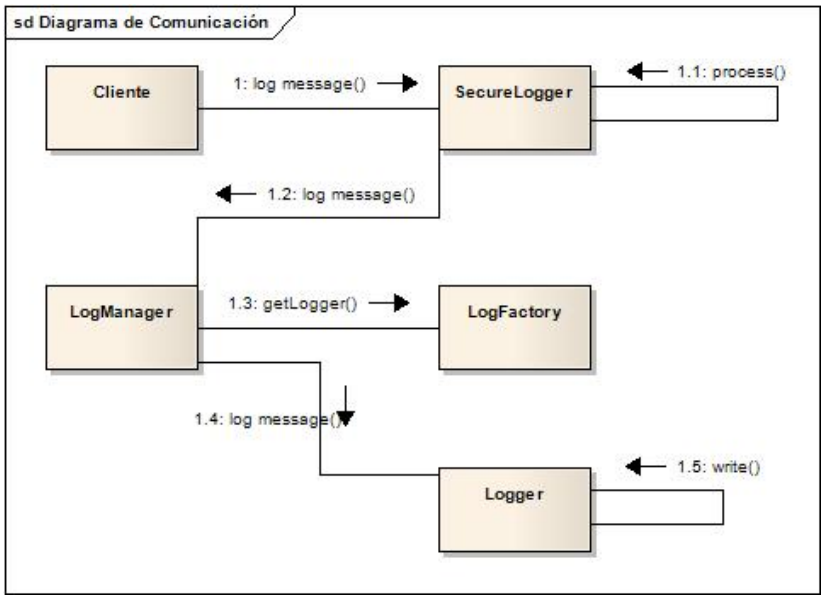


Figura 16: Ejemplo de un diagrama de comunicación simple.

Diagramas de descripción de la Interacción.

Estos diagramas representan interacciones utilizando una simplificación de la notación de los diagramas de actividad. A diferencia de los diagramas de actividad, en un diagrama de descripción de interacción los nodos del diagrama representan algún tipo de diagrama de interacción como pueden ser diagramas de secuencia, diagramas de comunicación, diagramas de tiempo y diagramas de descripción de la interacción. Ayudan a visualizar el flujo de control total a través del diagrama. No muestran información de mensajes detallada. Pueden agregarse ocurrencias de interacción para mostrar los detalles de los mensajes en una sección del sistema.

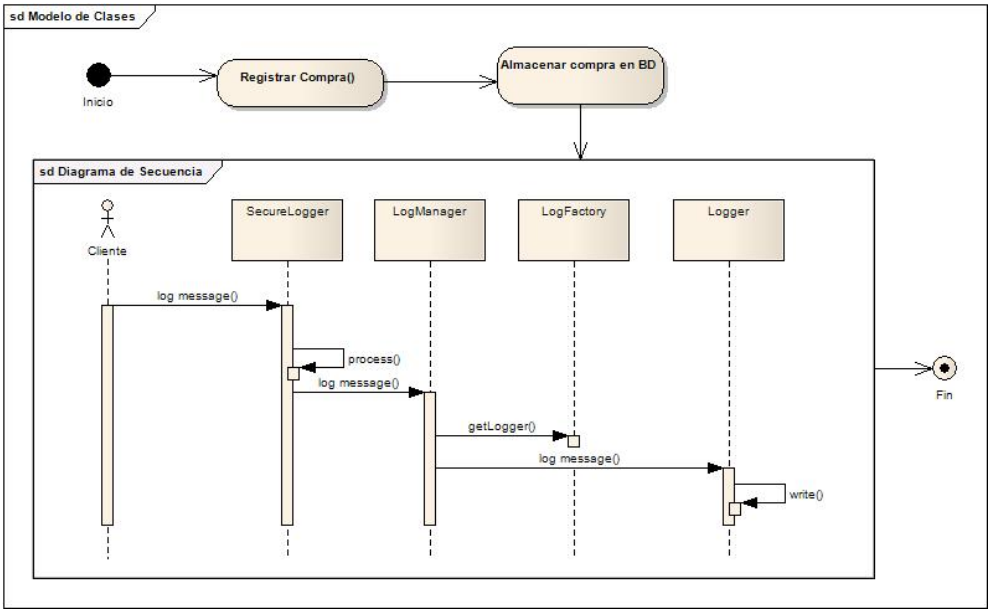


Figura 17: Ejemplo sencillo de un diagrama de descripción de interacción.

Diagramas de tiempo.

Se utilizan para mostrar el cambio en el estado o valor de uno o más elementos con respecto al tiempo. Estos diagramas son muy utilizados en los sistemas de tiempo real ó embebidos. A diferencia de los diagramas de secuencia, los diagramas temporales se leen de izquierda a derecha, en vez de arriba hacia abajo. Un lifeline se coloca en la parte izquierda del diagrama, seguidos de una lista de estados del lifeline y una representación gráfica de la transición entre los diferentes estados. Es posible indicar los eventos que esta transición entre estados.

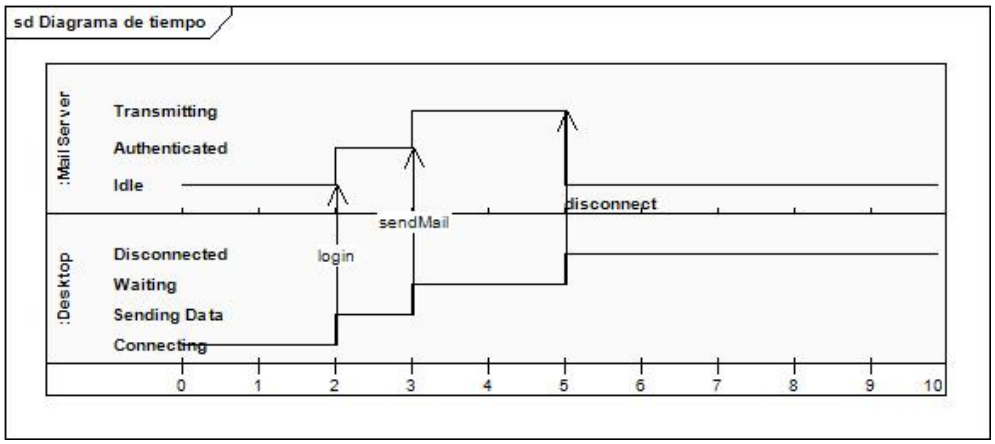


Figura 18: Ejemplo de Diagrama de Tiempo

Diagramas de Estado.

Los diagramas de estado se encargan de capturar el comportamiento de un sistema, ya sea a nivel de clase, subsistema o aplicación. También son muy utilizados para modelar la comunicación que existe entre entidades externas mediante un protocolo o un sistema basado en eventos. En ocasiones puede confundirse el uso de los diagramas de estado con los diagramas de actividad, sin embargo, el propósito de un diagrama de estado es el de mostrar los diferentes estados por los que pasa un objeto durante su ciclo de vida dentro del sistema, así como las transiciones entre los estados del objeto. Estas transiciones representan las actividades que provocan los cambios de estado del objeto. Por su parte, los diagramas de actividad se centran en representar las transiciones y actividades que provocan cambios en los objetos.

Debido a su doble propósito, UML define dos tipos de máquinas de estado: las máquinas de estado conductuales y las máquinas de estado de protocolo.

- Máquinas de estado conductuales: Representan el comportamiento de una pieza del sistema mediante una notación gráfica. Una máquina de estado se muestra utilizando una notación de rectángulo básica, con el nombre de la máquina en la parte superior del compartimento.
  El comportamiento de un clasificador se modela utilizando estados, pseudo-estados, actividades y transiciones. Un estado es un momento específico en el comportamiento del clasificador y se representa mediante un rectángulo con esquinas redondeadas y el nombre del mismo escrito dentro del rectángulo. Existen diferentes tipos de estados, los cuales son los estados simples, estados compuestos y estados de sub-máquina.

- Máquinas de estado de protocolo: Capturan el comportamiento de un protocolo. Especifican los cambios de estado y eventos asociados a una comunicación basada en protocolo. A diferencia de las máquinas de estado anteriores, los estados representan situaciones estables donde el clasificador no realiza ninguna operación y el usuario sabe su configuración. Las máquinas de estado de protocolo no aceptan actividades.

Diagramas de Casos de Uso.

Los diagramas de casos de uso se utilizan para capturar los requerimientos y la funcionalidad de un sistema. Básicamente están compuestos por piezas de funcionalidad (casos de usos), personas o cosas que invocan tal funcionalidad (actores) y posiblemente los elementos responsables de implementar los casos de uso (sujetos). Este diagrama es muy útil para definir los

límites del sistema o subsistemas, así como las relaciones existentes con su entorno, permitiendo resaltar las funciones que realizará y cada una de los actores, humanos o físicos, que interactuarán de manera directa con él.

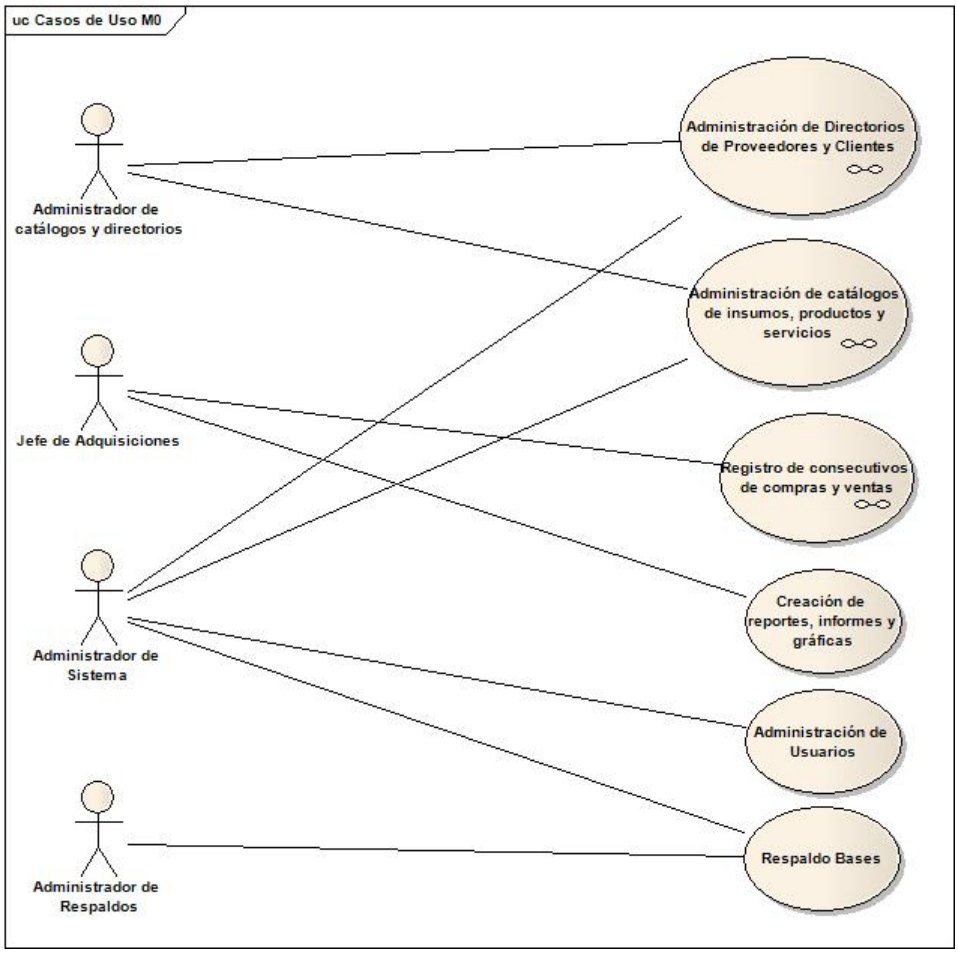


Figura 19: Ejemplo de diagrama de caso de uso

Una de las características importantes del lenguaje UML es su extensibilidad y básicamente puede ser de 2 maneras: agregando *restricciones*, *estereotipos*, *valores etiquetados* y *notas propias* en los mismos modelos de manera informal, o mediante la creación de un llamado *profile*. Un profile consiste en una colección de estereotipos y restricciones colocados en elementos UML genéricos, de tal forma que lo convierten en elementos UML de un dominio específico. Los estereotipos proporcionan una forma de definir nuevos elementos extendiendo y refinando la semántica de elementos ya existentes. Los valores etiquetados proporcionan una manera de definir nuevas propiedades de elementos ya existentes. Las restricciones proporcionan una manera de imponer reglas en los elementos y sus propiedades.

Todos los diagramas UML descritos se utilizan en el modelado de sistemas. Para su implementación, se hace uso de un lenguaje de programación que se elije basado en los requerimientos del proyecto. Existe una gran variedad de lenguajes de programación, sin embargo, en la sección siguiente solo se describen los lenguajes de programación más utilizados en la actualidad, así como sus características básicas.

1.4 Lenguajes actuales para el desarrollo de software.

En la actualidad existe una gran cantidad de lenguajes de programación, cada uno desarrollado con características muy específicas. Generalmente cuando nace un nuevo lenguaje, está orientado a un campo de aplicación específico, aunque existen lenguajes de propósito general. A continuación se describen solo aquellos que son los más utilizados para el desarrollo de software comercial, así como algunas de sus características principales.

1.4.1 C/C++

Ambos lenguajes de programación son distintos, sin embargo comparten muchas características debido a que el lenguaje C++ originalmente comenzó como una extensión a C también conocida como "*C with classes*".

C es un lenguaje de programación estructurado de propósito general, desarrollado por Dennis Ritchie en los Laboratorios Bell y orientado principalmente a la plataforma UNIX. A pesar de que estaba orientado al desarrollo de software de sistemas como es el caso de sistemas operativos, tiene un amplio uso en el desarrollo de software de aplicación, sobre todo en los sistemas de arquitectura cliente/servidor como es el caso de los servicios de internet. El lenguaje C tiene grandes capacidades de acceso a recursos de sistema a bajo nivel, especialmente para la administración de la memoria mediante el uso de punteros, así como la capacidad de incorporar instrucciones en lenguaje máquina o ensamblador.

Como se comentó anteriormente, C++ nace como una extensión al lenguaje C desarrollado por Bjarne Stroustrup a finales de los años 70 en los Laboratorios Bell. Es un lenguaje de programación orientado a objetos de propósito general. Se considera como un lenguaje de nivel medio ya que incorpora características de un lenguaje de bajo nivel y uno de alto. Además de

poseer las mismas características mencionadas para el lenguaje C, incorpora otras específicas para el desarrollo orientado a objetos como es el concepto de clases, funciones virtuales, sobrecarga de operadores, herencia múltiple, templates y manejo de excepciones. Al igual que C, C++ es muy utilizado para el desarrollo de software de sistemas, aunque debido a que C++ está orientado a objetos, permite el desarrollo de aplicaciones más complejas, por lo que es ampliamente utilizado para el desarrollo de software de aplicación multiplataforma a nivel de usuario final o aplicaciones de propósito específico.

Las características de control de sistemas a bajo nivel proporcionadas por ambos lenguajes de programación han permitido el nacimiento de técnicas de ataque a sistemas muy utilizadas como es el famoso 'buffer overflow'. Actualmente se han creado propuestas de código seguro muy útiles para esta plataforma de desarrollo, sin embargo, se sabe que la seguridad no es una de las principales características de estos lenguajes.

1.4.2 C#

C# es un lenguaje de programación orientado a objetos creado y estandarizado por la empresa Microsoft como parte de la plataforma de desarrollo .NET. Su sintaxis se deriva de los lenguajes de programación C y C++ incorporando características de otros lenguajes como Java y Delphi.

La plataforma .NET es muy similar a la plataforma Java con la ventaja que permite hacer uso de diversos lenguajes de programación de alto nivel. La herramienta de desarrollo compila el código fuente de cualquiera de los lenguajes soportados por .NET en un código intermedio conocido como MSIL (Microsoft Intermediate Language). Para ejecutar este código, se requiere de un compilador adicional llamado JIT (Just-In-Time), que es el encargado de generar código de máquina real. Es importante mencionar que en la plataforma .NET aún es posible generar código máquina directo, es decir, programas que no utilizan códigos intermedios.

En cuestión de seguridad, esta plataforma incluye algunas características similares a la máquina virtual Java como son la administración de memoria, seguridad de tipos de datos, inutilización de punteros, uso de dominios de aplicación que permite aislar dos aplicaciones que se ejecutan en el mismo framework, verificación de código intermedio, entre otras características adicionales. Una de las diferencias importantes entre la plataforma .NET y Java es que el primero no utiliza un interprete para ejecutar el código intermedio si no que utiliza un compilador adicional, que en teoría agiliza su ejecución.

Actualmente existe una implementación OpenSource del framework .NET conocida como MONO. Gracias al desarrollo de este proyecto, es posible crear aplicaciones .NET que se ejecuten en plataformas Unix, Solaris, Linux, Windows y Mac.

1.4.3 PHP

PHP en un lenguaje de programación interpretado, diseñado originalmente para la creación de sitios web dinámicos. Este lenguaje ha tenido grandes mejoras a lo largo de los últimos años como es el hecho de soportar programación orientada a objetos y el soporte de librerías para el desarrollo de aplicaciones de escritorio. Actualmente tiene un amplio uso en las aplicaciones web aunque no incluye características importantes de seguridad comparado con lenguajes como C# y Java. La seguridad proporcionada en las aplicaciones desarrolladas en PHP se basan en las buenas prácticas al momento de su desarrollo.

1.4.4 Java.

Java es un lenguaje de programación de alto nivel orientado a objetos desarrollado por la compañía SUN Microsystems a principios de los años 90´s. Mucha de su sintaxis la toma de los lenguajes C y C++, aunque utiliza un modelo de objetos más simple comparado con C++. Una característica importante es que Java no hace uso de los punteros, muy comunes en los lenguajes C y C++.

Este lenguaje de programación se diseñó con la idea de crear una plataforma que permitiera el desarrollo de aplicaciones para ambientes distribuidos y heterogéneos, a través de la creación de una máquina virtual multiplataforma que sirviera como base para la ejecución de aplicaciones Java. Originalmente, su uso se orientó a la plataforma web, incorporándose en los navegadores de internet. Esto traía como ventaja principal, la posibilidad de convertir Internet en un medio interactivo. Debido a que Java ha sido uno de los lenguajes de programación más recientes y desde sus inicios se encaminó a ambientes de red, la seguridad se convirtió en una de las características de más importancia. Dentro de algunas de sus características de seguridad son la seguridad del tipo de datos, controladores de acceso, mecanismo seguro de cargado de clases, verificación de código, así como una gran variedad de proveedores de librerías de servicios criptográficos.

En el capítulo siguiente se aborda el proceso de desarrollo propuesto por Jan Jürgens basado en el modelado de sistemas utilizando una extensión de UML diseñado para aquellos sistemas de seguridad crítica. Posteriormente, se analizarán más a fondo las características del lenguaje Java orientadas a la seguridad de sistemas.

Capítulo 2: El Lenguaje de Modelado UMLSEC.

Resumen: Existen dos métodos de desarrollo seguro de software que son básicamente los siguientes: 1) mediante pruebas de penetración posteriores al desarrollo de cierto sistema ó 2) mediante la implementación de un modelo matemático que se asume seguro. Existe una brecha importante entre ambos métodos ya que mientras uno resulta costoso e inadecuado, el otro requiere de alta especialización en materias complicadas y en constante desarrollo. En un intento por encontrar el punto medio y conociendo que gran parte de los problemas de seguridad de sistemas se deben a una mala planeación e implementación, Jan Jürjens ha desarrollado una extensión al lenguaje UML conocido como UMLSec. En este capítulo se presenta una descripción de este profile, sus estereotipos, valores etiquetados y restricciones. Por último, a manera de ejemplo, se presenta una herramienta CASE para el manejo de esta extensión desarrollada por el mismo Jan Jürgens.

2.1 El profile UMLSec.

UMLSec es una extensión formal del lenguaje de modelado UML que permite expresar características de seguridad dentro de las especificaciones y modelado de sistemas. Utiliza el mecanismo de extensión estándar de UML conocida como **'profile'**. Un profile UML es una especificación que se encarga de definir el subconjunto de un meta-modelo[7] UML y se define mediante el uso de estereotipos (*stereotypes*), valores etiquetados (*tagged values*) y restricciones (*constraints*).

Los estereotipos se utilizan junto con los valores etiquetados para expresar los requerimientos de seguridad y para hacer asunciones del ambiente del sistema. Por su parte, las restricciones UMLSec proporcionan los criterios que determinan si los requerimientos de seguridad expresados se cumplen o no.

UMLSec se diseñó con la idea de cumplir con los requerimientos necesarios que cualquier herramienta de modelado debe tener para el desarrollo de sistemas con seguridad crítica como son los siguientes (Jürgens, 2005):

[7] Un metamodelo UML es el modelo de un modelo, es decir, se refiere a un modelo que define y explica las relaciones existente entre los componentes de un modelo UML.

- **Requerimientos de seguridad:** UMLSec tiene la capacidad de expresar requerimientos de seguridad básicos en forma precisa como son confidencialidad, integridad, autenticación del origen de datos, entre otros.
- **Escenarios de amenaza:** Permite considerar diferentes situaciones que elevan las distintas posibilidades de ataques.
- **Conceptos de seguridad:** Permite emplear importantes conceptos de seguridad como son *tamper-resistant hardware*[8].
- **Mecanismos de seguridad:** Permite incorporar mecanismos de seguridad como son los protocolos de seguridad y controles de acceso.
- **Seguridad física:** Considera el nivel de seguridad proporcionado por la capa física.
- **Administración de la seguridad:** Considera cuestiones acerca de la administración de la seguridad.

Las características más importantes del profile son:

- Permite aplicarse a todos los diagramas UML.
- Define los estereotipos, restricciones y valores etiquetados necesarios para establecer características de seguridad en los modelos de sistemas. La Tabla 1 muestra un resumen de los elementos del profile.
- No requiere de requisitos previos.

[8] Dispositivos utilizados para proteger el acceso a llaves criptográficas. Generalmente se tratan de dispositivos externos a una computadora permitiendo de esta manera ofrecer una limitante física y lógica para el acceso a las llaves.
http://www.rsa.com/rsalabs/node.asp?id=2357

Tabla 1: Estereotipos de UMLSec.

Estereotipos	*Clase Base*	*Etiquetas*	*Restricciones*	*Descripción*
fair exchange	*subsystem*	*start, stop, adversary*	*Después de un 'start' eventuallmente llegar a un 'stop'.*	*Garantiza un intercambio justo*
probable	*subsystem*	*action, cert, adversary*	*Servicio de no-repudio en acciones determinadas*	*Requisito para el servicio de seguridad de no-repudio.*
rbac	*subsystem*	*protected, role, right.*	*Solo es posible realizar actividades permitidas.*	*Forza un control de acceso basado en roles.*
internet	*link*			*Conexión de Internet.*
encrypted	*link*			*Conexión cifrada.*
LAN	*link, node*			*Conexión LAN.*
wire	*link*			*Cable.*
smart card	*node*			*Nodo de 'smart card'.*
POS device	*node*			*Dispositivo de punto de venta.*
issuer node	*node*			*Nodo emisor.*
secrecy	*dependency*			*Asume confidencialidad.*
integrity	*dependency*			*Asume integridad.*
high	*dependency*			*Alta sensibilidad*
critical	*Object, subsystem*	*Secrecy, integrity, authenticity, high, fresh.*		*Objeto crítico.*
secure links	*subsystem*	*adversary*		*Forza la existencia de enlaces de comunicación seguros.*
secure dependency	*subsystem*			
data security	*subsystem*	*Adversary, integ., auth*	*Proporciona confidencialidad, integridad, autenticidad, freshness.*	*Requirimientos de seguridad de datos básicos.*
no down-flow	*subsystem*		*Evita down-flow*	*Condición de flujo de información.*
no up-flow	*subsystem*		*Evita up-flow*	*Condición de flujo de información.*
guarded access	*subsystem*			*Control de acceso utilizando objetos.*
guarded	*object*	*guard*		*Objeto resguardado.*

Tabla 2: Etiquetas de UMLSec.

Etiqueta	*Estereotipo*	*Tipo*	*Multip.*	*Descripción*
start	*fair exchange*	*Estado*	*	*Estado start.*
stop	*fair exchange*	*Estado*	*	*Estado stop.*
adversary	*fair exchange*	*Modelo de adversario*	*1*	*Tipo de adversario.*
action	*provable*	*Estado*	*	*Acción probable.*
cert	*provable*	*Expresión*	*	*Certificado*
adversary	*provable*	*Modelo de adversario*	*	*Tipo de adversario.*
protected	*rbac*	*Estado*	*	*Recursos protegidos.*
role	*rbac*	*(actor, rol)*	*	*Asigna un rol a un actor.*
right	*rbac*	*(rol, derecho)*	*	*Asigna un derecho a un rol.*
secrecy	*critical*	*dato*	*	*Confidencialidad de datos.*
integrity	*critical*	*(variable, expresión)*	*	*Integridad de datos.*
autenticity	*critical*	*(dato, origen)*	*	*Autenticidad de datos.*
high	*critical*	*mensaje*	*	*Mensaje de alto nivel.*
fresh	*critical*	*dato*	*	*Dato fresco.*
adversary	*secure links*	*Modelo de adversario*	*1*	*Tipo de adversario.*
adversary	*data security*	*Modelo de adversario*	*1*	*Tipo de adversario.*
integrity	*data security*	*(variable, expresión)*	*	*Integridad de datos.*
authenticity	*data security*	*(dato, origen)*	*	*Autenticidad de datos.*
guard	*guarded*	*Nombre del objeto*	*1*	*Objeto guardia.*

A continuación se describen cada uno de los estereotipos, valores etiquetados y restricciones definidos en el profile.

fair exchange: Este estereotipo se utiliza en los diagramas de casos de uso y diagramas de actividad.

En un diagrama de casos de uso, este estereotipo indica la necesidad de garantizar un intercambio justo entre diferentes partes de un sistema. Su uso en este diagrama es meramente informal puesto que solo indica que el subsistema que lo detalla deberá contener el mismo estereotipo para un correcto refinamiento.

En los diagramas de actividad este estereotipo tiene asociadas las etiquetas *{start}*, *{stop}* y *{adversary}*. Las etiquetas *{start}* y *{stop}* toman como parámetros *(good,state)*, donde *good* corresponde a un bien que se va a vender y *state* es el nombre de un estado. Si solo se va a vender un bien es posible evitar el parámetro *good*. La etiqueta *{adversary}* especifica un tipo o modelo de adversario del cual debemos mantener nuestro requisito de seguridad. La restricción asociada a este estereotipo indica que para cualquier bien que se va a vender, es necesario que una vez que alcance el estado definido en la etiqueta *{start}*, eventualmente se alcance un estado definido en la etiqueta *{stop}*, aún en presencia de un *{adversary}*.

provable: Este estereotipo tiene las etiquetas *{action}*, *{cert}* y *{adversary}*. La etiqueta *{cert}* define una expresión que se utilizará como prueba de que una acción en el estado dado en la etiqueta *{action}* se realizó. La etiqueta *{adversary}* especifica un tipo de adversario del cual se debe mantener este requisito de seguridad. Se asume que el certificado *{cert}* es único para cada instancia del subsistema. Este estereotipo ayuda al servicio de seguridad conocido como "no-repudio".

rbac (role-based access control): Este estereotipo se aplica a los subsistemas con diagramas de actividad y obliga al uso de un control de acceso basado en roles. Tiene asociadas las etiquetas *{protected}*, *{role}* y *{right}*. La etiqueta *{protected}* toma como valores los estados del diagrama de actividad cuyo acceso se debe controlar. La etiqueta *{role}* tiene como valores pares *(actor, role)* donde *actor* representa un actor en el diagrama de actividad y *role* especifica la función asociada dentro del diagrama de actividad. La etiqueta *{right}* tiene como valores una lista de pares *(role, right)* donde *role* continúa representando la función del actor y *right* representa el derecho de acceder un recurso protegido. La restricción asociada requiere que todos los actores del diagrama de actividad solo puedan realizar las actividades a las que tienen acceso. Para este estereotipo se recomienda utilizar los diagramas de actividad utilizando "particiones de actividad" de tal manera que se identifique claramente quien o que realiza una acción determinada.

internet, encrypted, LAN, wire, smart card, POS device, issuer node: Estos estereotipos se aplican sobre los enlaces y nodos en los diagramas de implementación, catalogando los diferentes tipos de enlaces de comunicación existentes en un subsistema. Se requiere que cada link o nodo tenga al menos uno de estos estereotipos. Por definición, cada estereotipo tiene asociado un número determinado de amenazas que dependen del tipo de atacante considerado. Básicamente, estas amenazas se resumen como amenazas de eliminación, lectura, inserción y acceso *(delete, read, insert, access)*. De esta forma podemos decir que el conjunto de amenazas dependerá del tipo de link/nodo a considerar, resumiéndose de la siguiente manera:

- Para un estereotipo de clase **"nodo"**, las amenazas son: *{access}*
- Para un estereotipo de clase **"enlace"**, las amenazas son *{delete, read, insert}*

Como ejemplo, en la Tabla 3 se muestran las posibles amenazas que tendría cada estereotipo si consideramos a un atacante externo con habilidades de cómputo promedio.

Tabla 3: Descripción de amenazas para un atacante externo con capacidades promedio.

Esterotipo	**Amenazas**	**Descripción**
Internet	{delete, read, insert}	Posibilidad de inserción, lectura y eliminación de mensajes
encrypted	{delete}	Posibilidad de eliminación de mensajes. Ej. Posibilidad de eliminar mensajes en una VPN provocando algún ataque de negación de servicios, sin embargo un atacante promedio no puede ser capaz de leer los mensajes o insertar mensajes utilizando la llave correcta.
LAN	Ø	Se considera que un atacante promedio no es capaz de acceder a los datos de la red local.
wire	Ø	Imposibilidad de acceder a los cables que interconectan dispositivos de seguridad crítica.
Smart card	Ø	Imposibilidad de acceder dispositivos smart card
POS device	Ø	Imposibilidad de acceder dispositivos de punto de venta.
Issuer node	Ø	Imposibilidad de acceder aun nodo emisor.

critical: Este estereotipo se aplica a objetos o instancias de subsistemas con datos que se consideran críticos para el sistema. Las etiquetas de este estereotipo son *{secrecy}*, *{integrity}*, *{authenticity}*, *{fresh}* y *{high}*. Cada una de estas etiquetas representa un requisito específico de seguridad.

Los valores de la etiqueta *{secrecy}* son los nombres de las expresiones, atributos o variables de argumentos de los mensajes del objeto actual, que deben tener servicio de confidencialidad. También es posible definir el nombre de la operación que requiere argumentos y valores de retorno confidenciales.

La etiqueta *{integrity}* toma como valores pares *(v,E)* donde *v* es una variable del objeto cuya integridad debe protegerse y *E* es el conjunto de expresiones válidas que pueden asignarse a *v*.

Los valores de la etiqueta *{authenticity}* son pares *(a,o)* de atributos del objeto o subsistema **<<*critical*>>** donde '*a'* se refiere a los datos que deben tener requisito de autenticidad y *o* se refiere al origen del dato. Este requisito también se le conoce como autenticidad de origen de datos.

La etiqueta *{fresh}* tiene como valores, datos atómicos generados recientemente.

Todos estas restricciones se refuerzan mediante la propia restricción del estereotipo **<<*data security*>>** definido posteriormente.

La etiqueta *{high}* toma como valores los nombres de los mensajes que deben protegerse. Se refuerzan con los estereotipos **<<*no down-flow*>>** y **<<*no up-flow*>>**.

secrecy, integrity, high: Estos estereotipos se utilizan en conjunto con el estereotipo **<<secure links>>** en los diagramas de estructura estática o en los diagramas de componentes. Etiquetan las dependencias que proporcionan los requerimientos de seguridad respectivos sobre los datos que se envían, ya sea como argumentos o como valores de retorno de operaciones o señales.

Los estereotipos **<<*secure links*>>**, **<<*secure dependencies*>>** y **<<*data security*>>**, así como **<<*secrecy*>>**, **<<*integrity*>>**, **<<*high*>>** y **<<*critical*>>**, definidos previamente, describen diferentes condiciones para garantizar la comunicación segura de datos. El estereotipo **<<*secure links*>>** se encarga de garantizar que los requerimientos de seguridad marcados en las dependencias de comunicación de los distintos componentes, estén soportados por la realidad física relativa al modelo de adversario bajo consideración. El estereotipo **<<*secure dependencies*>>** se encarga de la consistencia de los requerimientos de seguridad entre las diferentes partes del diagrama de estructura estática. Por último, el estereotipo **<<*data security*>>** se encarga de garantizar la seguridad a nivel de comportamiento.

secure links: Este estereotipo se utiliza a nivel de subsistema y ayuda a garantizar que los requerimientos de seguridad se cumplan a nivel de capa física, tomando en cuenta un adversario *"A"* que se especifica en la etiqueta asociada *{adversary}*. En otras palabras, este estereotipo se encarga de verificar que los requerimientos/estereotipos de *{secrecy}*, *{integrity}* y *{high}* definidos sobre los links de comunicación del subsistema se cumplan, tomando en cuenta el tipo de enlace, las posibles amenazas existentes en el enlace y el tipo de adversario considerado en el modelo. En resumen podemos decir que el estereotipo verifica lo siguiente:

- Si una dependencia tiene un estereotipo **<<*high*>>**, las posibles amenazas del atacante, definido en la etiqueta *{adversary}* sobre el link del subsistema, deben ser nulas.
- Si una dependencia tiene un estereotipo **<<*secrecy*>>**, las posibles amenazas del atacante, definido en la etiqueta *{adversary}* sobre el link del subsistema, no deben incluir *read*.

- Si una dependencia tiene un estereotipo **<<*integrity*>>**, las posibles amenazas del atacante, definido en la etiqueta *{adversary}* sobre el link del subsistema, no deben incluir *insert*.

secure dependency: Este estereotipo se aplica sobre subsistemas que contengan diagramas de estructura estática. Garantiza el cumplimiento de los requisitos de seguridad sobre los datos que se transmiten entre dependencias de objetos y subsistemas. Estos requisitos se definen mediante las etiquetas *{secrecy}*, *{integrity}* y *{high}* del estereotipo **<<*critical*>>**. Básicamente las condiciones a cumplir por este estereotipo son las siguientes:

- Para cada mensaje que aparece en la etiqueta *{secrecy}*, *{integrity}* o *{high}* del estereotipo **<<*critical*>>** de un objeto o subsistema *C* deberá incluirse en la etiqueta respectiva del objeto o subsistema dependiente *D*.
- Si un mensaje aparece en la etiqueta *{secrecy}*, *{integrity}* o *{high}* del estereotipo **<<*critical*>>** de un objeto o subsistema *C*, la dependencia deberá de tener un estereotipo **<<*secrecy*>>**, **<<*integrity*>>** o **<<*high*>>** según corresponda.

data security: Se aplica a los subsistemas y garantiza que el comportamiento de cualquier subsistema respete los requerimientos de seguridad dados por el estereotipo **<<*critical*>>** y las etiquetas asociadas contenidas en el subsistema. Este estereotipo considera el escenario de amenazas derivado del diagrama de implementación, tomando en cuenta el tipo de adversario especificado en la etiqueta *{adversary}* asociado al estereotipo.

Este estereotipo se cumple gracias a las siguientes condiciones:

- Secrecy: El subsistema conserva la confidencialidad de los datos designados por la etiqueta *{secrecy}* en contra de adversarios del tipo definido en la etiqueta *{adversary}*.
- Integrity: Si tenemos una etiqueta *{integrity}* de un estereotipo **<<*critical* >>** con un valor *(v,E)*, el subsistema protege la integridad de la variable *v* en contra de un adversario *{adversary}* respetando al conjunto de expresiones admisibles *E*. Si se omite el componente *E*, la integridad de *v* se protege de acuerdo al conjunto de expresiones que se pueden construir de aquellas que aparecen en la especificación del subsistema *S*. Esto quiere decir que el adversario no debe ser capaz de hacer que la variable *v* tome un valor previamente conocido solo por él.
- Authenticity: Para cada valor *(a,o)* de una etiqueta *{authenticity}*, el subsistema proporciona la autenticidad del atributo *a* de acuerdo a su origen *o*, en contra de cualquier adversario *{adversary}*.

- Freshness: Dentro del subsistema *S* con estereotipo **<<*data security*>>**, cualquier dato (incluyendo las llaves) con etiqueta *{fresh}* en la instancia del subsistema u objeto *D* con estereotipo **<<*critical*>>** en *S* deberá ser *fresh*???? en *D*.

no down-flow, no up-flow: Este estereotipo se aplica a subsistemas y su función es la de evitar fuga de información importante o corrupción de datos sensibles. Garantiza el flujo de información segura haciendo uso de la etiqueta *{high}* asociado al estereotipo **<<*critical*>>**.

guarded access: Este estereotipo se aplica a los subsistemas. Especifica que cada objeto del subsistema que tiene un estereotipo **<<*guarded*>>** solo se puede acceder a través de objetos especificados por la etiqueta *{guard}* del estereotipo **<<*guarded*>>**.

guarded: Este estereotipo se aplica a objetos. Tiene una etiqueta asociada *{guard}* el cual define el nombre del objeto protector correspondiente.

Para el uso de este profile, Jan Jürjens propone el uso de una herramienta que permite el análisis de los modelos basados en UMLSec. En la sección siguiente se explica su funcionamiento, así como ventajas y desventajas de su uso.

2.2 Herramientas de soporte para UMLSec.

Como en el caso del desarrollo de sistemas estándar, el desarrollo de sistemas seguro utilizando UMLSec propone el uso de herramientas CASE que permitan verificar la correcta utilización de cada uno de los estereotipos, valores etiquetados y restricciones antes mencionadas. Actualmente estas herramientas no han sido muy desarrolladas, ya que el mismo profile no ha sido muy extendido, sin embargo, se explica a continuación un poco de la misma,

La creación de una herramienta para el procesamiento de modelos UML se debe en gran medida al desarrollo del lenguaje XML como formato de almacenamiento de datos universal. En el año 2000, OMG emitió la primera especificación del lenguaje de intercambio de metadatos XML ó XMI, el cual se convirtió en el estándar para el intercambio de modelos UML entre las distintas herramientas de modelado. El lenguaje XMI, al igual que UML, siguen el estándar MOF (Meta-Object Facility). Este estándar se encarga de definir un lenguaje abstracto y una plataforma para la especificación, construcción y gestión de lenguajes de modelado también conocidos como meta-modelos. MOF utiliza distintas capas de información mostradas en la Figura 20.

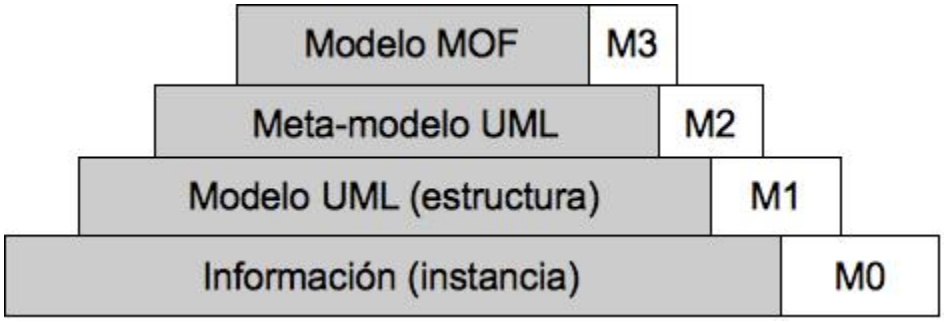


Figura 1: Framework MOF - meta-niveles.

De acuerdo con la figura anterior, el nivel M0 es el encargado de las instancias de datos por ejemplo *"Paul McCarney"* o *"Ciudad de México"*. El nivel M1 describe los modelos de datos que para el caso del desarrollo de software corresponde al modelo UML de cierta aplicación. Un ejemplo puede ser el concepto *Persona* o *Ubicación* representado con UML. El siguiente nivel de abstracción M2 se refiere al lenguaje de modelado como tal, y por último, el nivel M3 que se encarga de establecer la plataforma (*framework*) para el desarrollo de lenguajes de modelado con distintos campos de aplicación.

Por otro lado, el estándar MOF se relaciona ampliamente con dos estándares adicionales que son XMI (XML Metadata Interchange) y JMI (Java Metadata Interchange). XMI es un mapeo de MOF a XML y es muy utilizado como formato para el intercambio de datos de cualquier lenguaje descrito por MOF. Por su parte, JMI es un estándar que define el mapeo de MOF a Java.

Para la herramienta de soporte UMLSec se utiliza la librería MDR (Meta.data repository) el cual forma parte del proyecto Netbeans y es utilizado por la herramienta de desarrollo Poseidon 1.6 Community Edition.

La librería MDR implementa un repositorio MOF con soporte para los estándares XMI y JMI, de esta manera, el repositorio MOF tiene la arquitectura siguiente:

1. Se utiliza la descripción XMI del lenguaje de modelado UML para personalizar el repositorio MDR para que trabaje con el tipo de modelo en particular. En el caso de la herramienta para UMLSec, se utiliza la descripción XMI del lenguaje UML 1.5 publicado por la OMG.
2. Se crea el almacén personalizado para el tipo de modelo dado.
3. Basándose en la especificación XMI del lenguaje de modelado, la librería MDR crea la implementación JMI para acceder al modelo.

4. Se carga el modelo UML en el repositorio y es posible acceder al mismo mediante las interfaces JMI desde una aplicación Java. El modelo puede leerse, modificarse y guardarse como un archivo XMI.

La Figura 21 muestra la arquitectura de la aplicación desarrollada para el profile UMLSec utilizando la librería MDR.

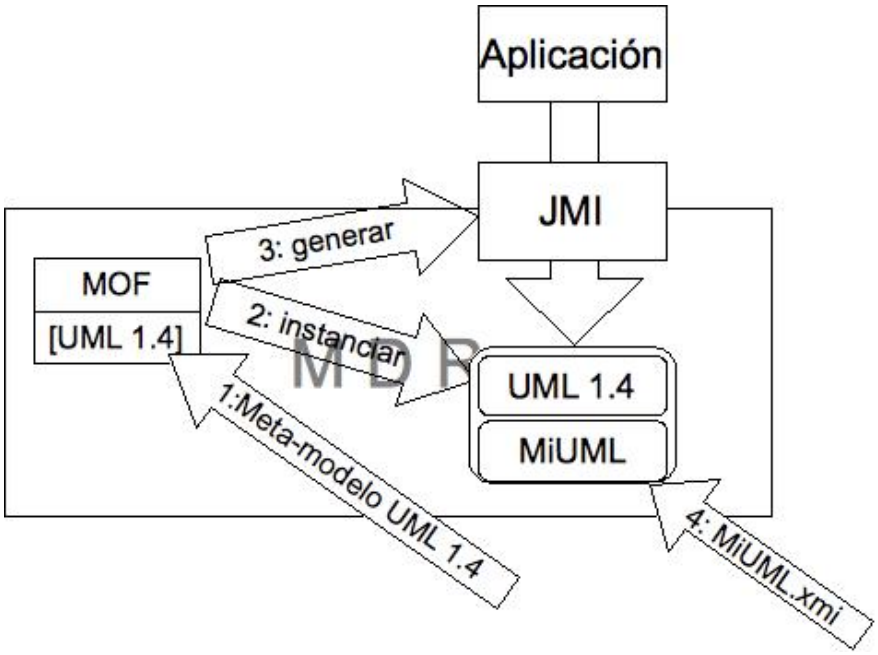


Figura 2: Uso de la biblioteca MDR.

Existen distintos niveles de funcionalidad en la verificación de los modelos UMLSec con esta herramienta CASE, y depende de los estereotipos a verificar. Los niveles son:

- **Características estáticas:** Capacidad para verificar las propiedades de seguridad incluidas como estereotipos en los diagramas de despliegue y de estructura, como es el caso de los estereotipos <<*secure links*>> y <<*secure dependency*>>.

- **Características dinámicas simples:** Se analiza el comportamiento del modelo, descrito por los diagramas de secuencia y de estados, para verificar el cumplimiento de los requerimientos de seguridad básicos definidos a nivel de comportamiento, como es el caso del estereotipo <<*fair exchange*>>.

- **Características dinámicas complejas:** El modelo UMLSec que describe el comportamiento dinámico se transforma en un lenguaje de entrada a una herramienta de análisis. De esta manera, es posible verificar el modelo de acuerdo a propiedades dinámicas complicadas e imperceptibles, como es el caso del estereotipo <<*data security*>>.

- **Acoplamiento hacía aplicaciones externas:** Es deseable contar con la posibilidad de conectar el framework de la herramienta UMLSec con aplicaciones externas que permitan proporcionar datos para que sean analizados junto con los modelos UML.

La Figura 3 muestra el conjunto de herramientas necesarias para el desarrollo de sistemas con UMLSec. El procedimiento debería ser como sigue: 1) El desarrollador crea un modelo en una herramienta de modelado estándar y lo almacena en un formato de archivo UML1.5/XMI1.2. 2) El archivo se importa por la herramienta de UMLSec en un repositorio MDR interno. 3) La herramienta accede el modelo a través de las interfaces JMI generadas por la librería MDR. 4) El verificador estático (static checker) parsea el modelo, verifica sus características estáticas y entrega sus resultados al analizador de errores (error analyzer). 5) El verificador dinámico (dynamic checker) traduce los fragmentos relevantes del modelo UML en un lenguaje de entrada al verificador de modelo (model-checker). 6) El verificador de modelo se genera como un proceso externo a la suite. 7) Los resultados de este proceso se regresan al analizador de errores. Este analizador utiliza la información recibida del verificador estático y el verificador dinámico para producir un reporte de texto útil para el desarrollador describiendo los problemas encontrados, así como un modelo UML modificado donde los errores encontrados se visualizan y, en la medida de lo posible, se corrigen.

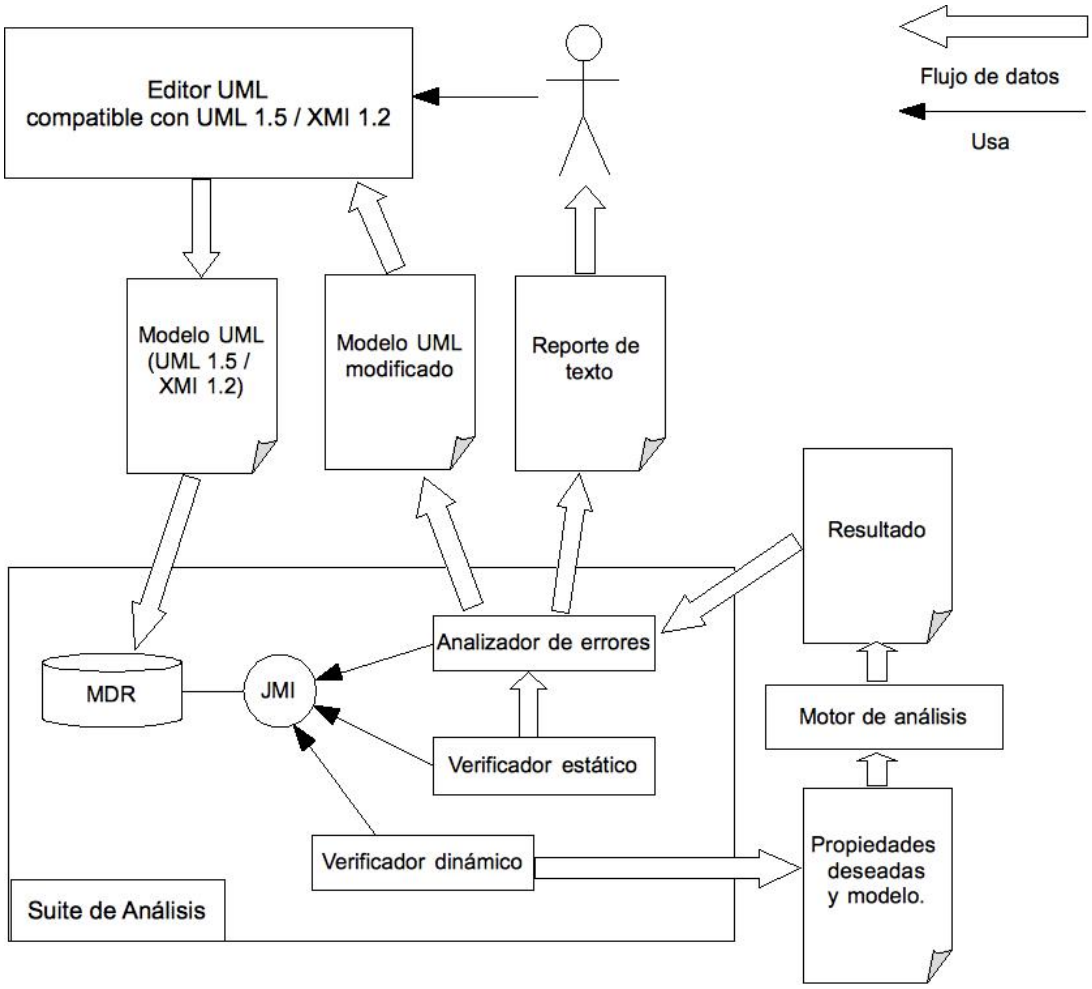


Figura 3: Suite de herramientas UML

Como se puede apreciar en la figura anterior el conjunto de herramientas propuesto por Jan Jürjens permite el análisis completo de las propiedades de seguridad de un sistema, desde las características estáticas hasta las características dinámicas simples y complejas, Realizando un análisis del profile y la herramienta de análisis hemos encontrado lo siguiente.

Ventajas.

1. La formalización para incorporar características de seguridad en el análisis y diseño de sistemas resulta en una buena propuesta para mejorar la seguridad en el desarrollo de software.
2. El uso del lenguaje de modelado UML y la creación de una extensión formal especializada en seguridad, permite una mejor y más rápida adopción del tema en el desarrollo de sistemas en la actualidad.

3. El uso de un lenguaje estándar para el intercambio de datos (XMI), así como el análisis del modelo a distintos niveles, permitiría elevar considerablemente la seguridad de los sistemas de software.

Desventajas.

1. Actualmente sólo trabaja con una herramienta de diseño que es ArgoUML. La herramienta Poseidon 1.6, con la cual se basó originalmente la herramienta, ya no está disponible. A pesar de que en teoría ArgoUML puede funcionar para generar los modelos que alimentan la suite, no es compatible al 100%.
2. En el diseño y análisis de sistemas comerciales se utilizan herramientas distintas a ArgoUML, y éstas no incorporan el profile de seguridad.
3. Esta herramienta trabaja con la versión 1.5 del lenguaje de modelado, sin embargo, la versión UML actual es 2.1. Por esto, es necesario generar una versión de la herramienta conocida como “viki” que trabaje sobre una versión más actual. Es bien sabido que las tecnologías de la información están en mejora continua, sin embargo sería deseable que la herramienta soportara al menos la versión 2.0 UML, más utilizada en la actualidad.
4. La herramienta de análisis sólo trabaja con un modelo o diagrama a la vez. Generalmente, el diseño de un sistema se basa en diversos modelos, como pueden ser el modelo de dominio, modelo conceptual, modelo de interfaz de usuario, modelos estáticos y modelos dinámicos.
5. Por otro lado, la misma herramienta en sí es un sistema de software, por lo que debe desarrollarse siguiendo las técnicas de programación apropiadas y siguiendo estándares de desarrollo adecuados.

En la Figura 4 se incluye una impresión de pantalla de la interfaz gráfica de la herramienta “viki”.

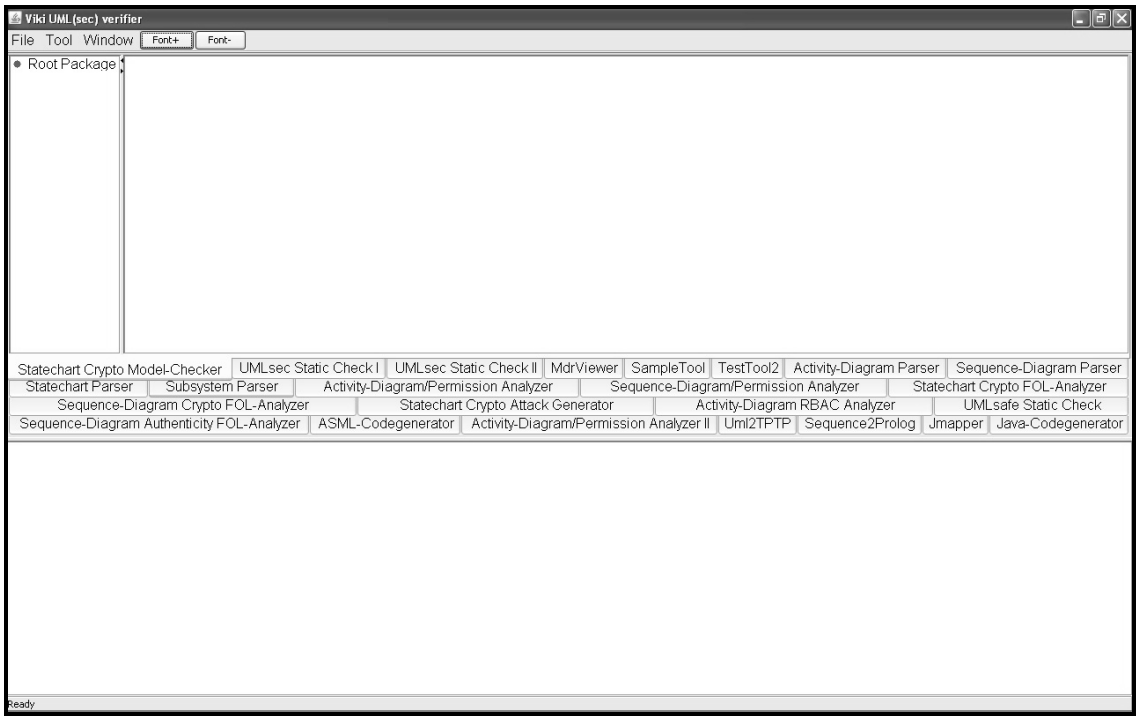


Figura 4: Suite de análisis UMLSec (viki).

Dado que gran parte de los problemas de seguridad en el desarrollo de sistemas se originan en las etapas de análisis y diseño, utilizar el profile UMLSec ayuda a reflejar requerimientos y restricciones de seguridad en cada uno de nuestros modelos, sin embargo es necesario adoptar un procedimiento de desarrollo que verifique los requerimientos de seguridad a lo largo de todo el ciclo de vida del software, así como la adopción de estándares aceptados internacionalmente para el desarrollo de sistemas seguros. En el capítulo siguiente, se abordan estos temas que son de gran importancia para todo desarrollo seguro de sistemas.

.

.

Capítulo 3: Procedimientos y Estándares para un Desarrollo Seguro.

Resumen: En el desarrollo de software existe un problema conocido como problema de refinamiento. A lo largo del ciclo de desarrollo se utilizan un conjunto de modelos que van de lo general a lo particular. En cada etapa se utiliza un modelo que resulta más concreto que el de la etapa anterior, sin embargo, que sea más concreto, no significa necesariamente que conserve sus características funcionales y de seguridad. Con el objeto de incorporar los requerimientos de seguridad directamente en el proceso de desarrollo de sistemas y eliminar los problemas de refinamiento, se describen las actividades de seguridad recomendadas que se deben incorporar en un proceso de desarrollo genérico, a su vez, se describen el "proceso de desarrollo seguro" o "Secure UP", descrito en (Steel, Nagappan, & Lai, 2006) que consiste en un proceso basado en el proceso de desarrollo unificado.

3.1 Problema de refinamiento de sistemas.

El proceso unificado es uno de los métodos mejor aceptados para el desarrollo de software y una de sus principales características es el de ser iterativo e incremental. Esto significa que sigue un método de arriba hacia abajo (top-down) o de lo general a lo específico, es decir, comienza especificando características básicas y generales, que se detallan en cada etapa posterior. Una de sus principales ventajas es que permite la detección de errores en etapas tempranas del desarrollo, así como la incorporación de nuevas funciones en etapas posteriores. Cada cambio a la especificación inicial del sistema durante el proceso de desarrollo se soporta mediante los llamados refinamientos. Un refinamiento relaciona dos descripciones del mismo modelo a dos niveles distintos de detalle, uno más concreto y otro más abstracto. En cada una de las etapas del proceso unificado de desarrollo de software se realiza un refinamiento de un modelo más abstracto. Es muy importante aclarar que la misma etapa de implementación es el refinamiento de una etapa más abstracta del sistema.

En teoría, se considera que el modelo más concreto representa el mismo modelo, pero a mayor detalle, sin embargo nada garantiza que así sea. El resultado podría ser un modelo totalmente distinto, o un modelo que ha perdido propiedades de diseño importantes. Para las propiedades de seguridad, es esencial que estas propiedades se mantengan a lo largo del desarrollo de sistemas y de aquí surge la necesidad de incorporar procedimientos para mantener las propiedades de seguridad de los sistemas a lo largo del ciclo de vida de los sistemas.

Debido a la importancia que existe en tener unos requerimientos de seguridad bien definidos y mantenerlos a lo largo del ciclo de vida del software, se han desarrollado mejores prácticas, modelos de madurez y estándares, que aplicados a los distintos procesos de desarrollo existentes, ayudan a conservar, verificar y monitorear los diferentes requerimientos de seguridad de un sistema. En las secciones siguientes se abordan las mejores prácticas recomendadas por el equipo CERT[9] para el desarrollo de sistemas, algunos ejemplos de procesos de desarrollo seguro, así como el modelo SSE-CCM que es un modelo de madurez orientado a la seguridad de sistemas.

3.2 Principios de Seguridad Recomendados.

Antes de entrar en una explicación detallada de las prácticas de seguridad recomendadas, es importante mencionar 8 principios que ayudan en el desarrollo seguro de sistemas. Estos principios fueron propuestos por Jerome H. Saltzer y son los siguientes:

1. **Economía del mecanismo:** Mantener el diseño tan simple y pequeño como sea posible. Un sistema complejo incrementa la probabilidad de cometer errores en la implementación, configuración y uso.
2. **Default seguros (Fail-safe defaults):** Basar las decisiones de acceso en permisos en vez de exclusiones. Por default, el acceso a un recurso se niega y el esquema de protección debe identificar las condiciones bajo las cuales el acceso se permite.
3. **Mediación completa:** Cada acceso a cada objeto debe verificarse antes de autorizarse. Requiere que el origen de cada solicitud a un objeto dado se identifique y autorice antes de acceder a un recurso dado.
4. **Diseño abierto:** El diseño no debe mantenerse en secreto. Un diseño seguro no debe basarse o depender de la ignorancia de atacantes potenciales o de mantener el código en secreto.
5. **Separación de privilegios:** Cada vez que sea factible, es recomendable utilizar un mecanismo de protección que requiera de dos llaves para desbloquear ya que es más robusto y flexible que aquellas que permiten el acceso mediante una sola llave. Elimina un único punto de falla, solicitando más de una condición para otorgar los permisos.
6. **Privilegios mínimos:** Cada programa y cada usuario del sistema debe operar utilizando los privilegios mínimos necesarios para realizar su trabajo y cualquier permiso elevado debe mantenerse por el menor tiempo posible. Este principio se puede implementar utilizando las técnicas siguientes:

[9] Computer Emergency Response Team

   a. Otorgar a cada sistema, subsistema y componente los privilegios mínimos necesarios con los que pueda operar.
   b. Adquirir y descartar privilegios de tal forma que en cualquier punto, el sistema solo tenga los privilegios necesarios para realizar su tarea.
   c. Descartar el privilegio de "cambiar privilegios" si no se requieren de cambios posteriores.
   d. Diseñar programas que utilicen los privilegios lo más pronto posible, antes de interactuar con algún adversario potencial y después descartarlos por el resto del programa.
7. **Mecanismo menos común:** Minimiza la cantidad de mecanismos comunes a más de un usuario.
8. **Aceptabilidad psicológica:** Actualmente conocido con el término de 'usabilidad', se refiere a la necesidad de diseñar sistemas fáciles de usar, de tal forma que los usuarios rutinariamente y de forma automática apliquen los mecanismos de protección de manera correcta. De acuerdo a (Seacord, 2006), una gran cantidad de vulnerabilidades encontradas en las bases de datos de CERT se atribuyen a problemas de usabilidad. De acuerdo a la referencia, el segundo tipo de vulnerabilidad más común, después del conocido "buffer overflow", es "configuración por default insegura después de la instalación". Algunos otros hechos relacionados con la usabilidad catalogados en la base de datos son:
   a. Los programas son difíciles de configurar de forma segura o fácil de desconfigurar.
   b. El procedimiento de instalación crea vulnerabilidades en otros programas.
   c. Problemas de configuración.
   d. Mensajes de confirmación y error confusos.

Gran parte de las prácticas de seguridad recomendadas a continuación se basan en los principios mencionados. Estas prácticas se dividen en actividades aplicables en distintas áreas de una organización y se clasifican de la siguiente manera:

1. Actividades de Ingeniería de Seguridad: Son aquellas actividades necesarias para implementar una solución segura y están ampliamente relacionadas con el proceso de desarrollo de sistemas, incluyen la captura de requerimientos, análisis, diseño, implementación, revisiones y pruebas de seguridad a los sistemas.

2. Actividades de Garantía de Seguridad: Estas actividades incluyen procesos de verificación, validaciones, revisiones de expertos y revisiones de artefactos creados.
3. Actividades de Administración de Proyecto y Seguridad Organizacional: En estas actividades se incluyen la creación de políticas y reglas organizacionales, actividades de administración de proyectos como planeación, asignación de recursos, así como actividades para la planeación, gestión y monitoreo de sistemas.
4. Actividades de Administración e Identificación de Riesgo de Seguridad: Es una de las actividades más importantes para la implementación de seguridad en una organización y se refiere al proceso de identificación de riesgos. Estas actividades son el origen de toda la seguridad de una organización.

En la actualidad, existen distintos procesos de desarrollo de software siendo de los más conocidos el proceso unificado o RUP, el proceso en cascada y la programación extrema. Independientemente del nombre del proceso utilizado, cada uno incluye fases bien definidas de planeación o requerimientos, análisis, diseño, codificación o implementación y pruebas. A lo largo de un ciclo de desarrollo de software genérico existen distintas estrategias de mitigación que incluyen actividades que se aplican en las distintas fases del desarrollo antes mencionadas. Estas actividades no asumen ningún proceso de desarrollo, sin embargo, deberían realizarse en algún punto del ciclo de vida del sistema. Las actividades recomendadas son las siguientes:

Ingeniería de Requerimientos de Seguridad.

El rol tradicional de la ingeniería de requerimientos es determinar las necesidades funcionales de un sistema. Es bien conocido escribir especificaciones de requerimientos funcionales, sin embargo, no es fácil escribir y expresar restricciones y/o requerimientos de seguridad. Cuando estos requerimientos no se definen de manera correcta, el sistema resultante no puede evaluarse de manera adecuada. En este caso, el Instituto de Ingeniería de Software (SEI por sus siglas en inglés) de la Universidad Carnegie-Mellon ha desarrollado un proceso de ingeniería de requerimientos conocido como SQUARE[10] diseñado para obtener, categorizar, priorizar y analizar requerimientos de seguridad. La metodología SQUARE está formada por 9 pasos, de los cuales los pasos del 1 al 4 son prerrequisitos obligatorios. Los pasos definidos por esta metodología son los siguientes:

[10] SQUARE: Security Quality Requirements Engineering.

1. **Acuerdo de definiciones:** Es un prerrequisito para la ingeniería de requerimientos de seguridad. Se recomienda utilizar fuentes de definiciones confiables como pueden ser *Internet Security Glossary (rfc 2828)*[11] *y The Guide to the Software Engieering Body of Knowledge*[12].
2. **Identificación de objetivos de seguridad:** Deben identificarse y priorizar objetivos de seguridad para la organización así como para el sistema de información a desarrollarse. Es importante involucrar a todos los 'stateholders' y una vez identificados se requiere priorizarlos.
3. **Desarrollo de artefactos:** Es importante desarrollar la documentación acerca del concepto de operaciones, metas del proyecto, uso normal y escenarios de amenaza, casos de mal-uso y demás documentos que soporten la definición de los requerimientos para evitar confusiones o falta de comunicación.
4. **Realizar evaluaciones de riesgos:** Para este paso, se requiere de un experto en métodos de evaluación de riesgos, el apoyo de todos los 'stakeholders' y de los ingenieros en requerimientos de seguridad. Para esto, se hará uso de los artefactos creados en el punto anterior. Este análisis de riesgos ayudará a identificar aquellas exposiciones de seguridad de alta prioridad.
5. **Seleccionar una técnica de adquisición:** Es importante seleccionar una técnica de adquisición de requerimientos adecuada para los 'stakeholders' de la organización, especialmente cuando hay distintos tipos de stakeholders.
6. **Adquisición de requerimientos de seguridad:** Se aplica la técnica de adquisición de requerimientos. La mayoría de las técnicas proporcionan una guía detallada de captura de requerimientos.
7. **Categorización de requerimientos:** Permite distinguir entre los requerimientos esenciales, los requerimientos deseados y las restricciones de la arquitectura.
8. **Priorización de requerimientos:** Se basa en la categorización de requerimientos y puede requerir de un análisis costo/beneficio. Existe una cantidad importante de métodos de priorización útiles como pueden ser BST o árbol de búsqueda binaria, técnica de asignación numérica, juego de planeación, método de los 100 puntos, teoría W también conocida como ganar-ganar, triage de requerimientos, método wiegers o el método AHP.
9. **Inspección de requerimientos:** Ayudará a definir un conjunto inicial de requerimientos de seguridad priorizados así como a identificar las áreas que requieran de un análisis posterior más detallado.

[11] http://www.faqs.org/rfcs/rfc2828.html
[12] SWEBOK (http://www.swebok.org/)

Uno de los puntos más importantes en la etapa de captura de requerimientos es no confundir requerimientos de seguridad con mecanismos de seguridad. Por ejemplo, especificar la necesidad del uso de un canal cifrado, no es un requerimiento, si no un mecanismo de seguridad.

Modelo de Amenazas.

Para crear un software seguro se requiere anticipar las amenazas a las cuales el sistema va ser objeto. Los modelos de amenazas contienen la definición de la arquitectura de la aplicación y una lista de amenazas para el escenario de la aplicación. Incluye la identificación de activos, identificación y categorización de amenazas para cada activo o componente, clasificación de amenazas basados en una clasificación de riesgos, y un desarrollo de estrategias de mitigación de amenazas.

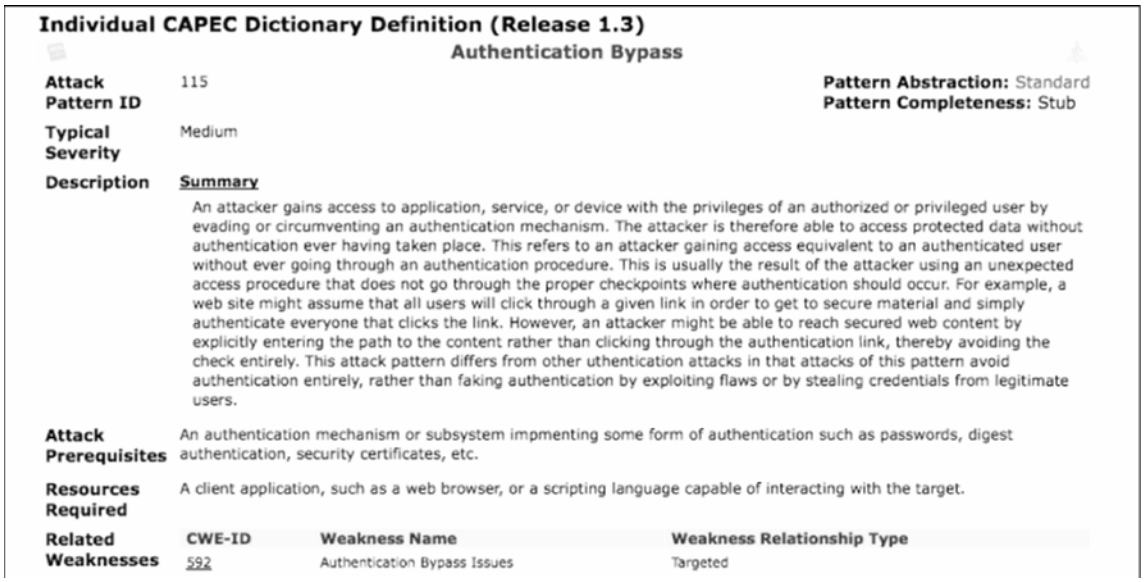
Casos de Uso y Mal Uso.

Así como se desarrollan casos de uso en la etapa de requerimientos, es importante realizar casos de mal uso. Un caso de mal uso de seguridad es una variación del caso de uso y se utiliza para describir un escenario desde el punto de vista del atacante. Estos mal usos se utilizan como una herramienta para la comunicación de posibles amenazas para un cliente o usuario final de un sistema. Para realizar estos diagramas una fuente importante son los patrones de ataque. Se define patrón de ataque como la representación genérica de un ataque deliberado y malintencionado que ocurre en contextos específicos. En este sentido cada patrón de ataque contiene lo siguiente:

- Resumen del objetivo del ataque especificado por el patrón.
- Una lista de precondiciones de su uso.
- Los pasos para realizar el ataque.
- Una lista de postcondiciones que serán verdaderas si el ataque es exitoso.

Actualmente existen referencias importantes de patrones de ataque, siendo la más completa la llamada CAPEC (Common Attack Pattern Enumeration and Classification), la cual es una lista de patrones de ataque comunes. Un ejemplo de patrón de ataque consultada en el sitio web de CAPEC[13] se muestra a continuación:

[13] http://capec.mitre.org/



Individual CAPEC Dictionary Definition (Release 1.3)

Authentication Bypass

Attack Pattern ID	115	**Pattern Abstraction:** Standard **Pattern Completeness:** Stub
Typical Severity	Medium	
Description	**Summary** An attacker gains access to application, service, or device with the privileges of an authorized or privileged user by evading or circumventing an authentication mechanism. The attacker is therefore able to access protected data without authentication ever having taken place. This refers to an attacker gaining access equivalent to an authenticated user without ever going through an authentication procedure. This is usually the result of the attacker using an unexpected access procedure that does not go through the proper checkpoints where authentication should occur. For example, a web site might assume that all users will click through a given link in order to get to secure material and simply authenticate everyone that clicks the link. However, an attacker might be able to reach secured web content by explicitly entering the path to the content rather than clicking through the authentication link, thereby avoiding the check entirely. This attack pattern differs from other uthentication attacks in that attacks of this pattern avoid authentication entirely, rather than faking authentication by exploiting flaws or by stealing credentials from legitimate users.	
Attack Prerequisites	An authentication mechanism or subsystem impmenting some form of authentication such as passwords, digest authentication, security certificates, etc.	
Resources Required	A client application, such as a web browser, or a scripting language capable of interacting with the target.	

Related Weaknesses	**CWE-ID**	**Weakness Name**	**Weakness Relationship Type**
	592	Authentication Bypass Issues	Targeted

Figura 1: Patrón de Ataque "Authentication Bypass"

En este sentido, los patrones de ataque son un recurso invaluable en cada una de las etapas del ciclo de desarrollo de software independientemente del proceso de desarrollo elegido.

1. En la etapa de captura de requerimientos ayudarán a identificar requerimientos de seguridad positivos y negativos;
2. Muchos ataques explotan errores en la arquitectura y diseño de software por lo que considerar los patrones de ataque aplicados en esos casos ayudará a evitar cometer ciertos errores o aplicar características de seguridad importantes en los sistemas.
3. Para las etapas de codificación e implementación, los patrones de ataque ayudarán al equipo de desarrollo a evitar vulnerabilidades comunes al momento de la implementación y a pesar de que los patrones de ataque no son garantía de una codificación segura, ayudarán a orientar el tipo de herramientas a utilizar en el análisis de código fuente.
4. Para las etapas de pruebas, los patrones de ataque son utilizados por personas individuales que actúan como “hackers” y tratan de vulnerar el sistema.

En este sentido, es recomendable realizar un análisis detallado de los posibles patrones de ataque aplicados en el sistema a desarrollar y de esta manera identificar los atacantes del sistema y la forma en que posiblemente realizarían el ataque.

Arquitectura y Diseño

La arquitectura y diseño forman parte esencial en la seguridad final de un sistema. Si la arquitectura y diseño tienen defectos, no hay nada que se pueda hacer para garantizar los requerimientos de seguridad establecidos. En este caso, es importante invertir el tiempo necesario para el análisis y diseño de un sistema, utilizar en la medida de lo posible los principios de seguridad mencionados con anterioridad y basarse ampliamente en los requerimientos de seguridad capturados al principio del desarrollo. Por otro lado, existen patrones de desarrollo seguro que ayudan a definir una arquitectura de seguridad apropiada a las necesidades de cierta organización. Uno de los retos más importantes de esta etapa no es solo la creación de un diseño y arquitectura segura si no que debe considerarse una arquitectura flexible que permita cambios posteriores en los requerimientos de seguridad.

Verificaciones de compilador.

En la medida de lo posible, es recomendable utilizar herramientas automáticas que ayuden a verificar la seguridad a nivel de implementación. En el caso de los lenguajes compilados, es importante la utilización de banderas del compilador que realicen verificaciones de tipos de datos y habilitar el despliegue de 'warnings'. El objetivo será realizar las modificaciones necesarias de código y diseño, para evitar malas prácticas de programación detectables en tiempo de compilación.

Validación de entradas.

Una de las causas más comunes de las vulnerabilidades de sistemas es la falta de validación de entrada de datos. Esta validación requiere de los siguientes pasos:

1. **Identificar los orígenes de entrada de datos:** Es importante detectar todas las entradas al sistema, como pueden ser, entrada por línea de comandos, interfaces de red, variables ambiente y archivo controlados por el usuario.
2. **Especificar y validar datos:** Todos los datos de entrada deben ser bien identificados y definir su origen, limites, valores y longitudes mínimos y máximos, contenido válido, requerimientos de inicialización y re-inicialización, así como requerimientos de cifrado para su almacenamiento y transmisión.
3. **Garantizar que todas las entradas cumplen con la especificación.** Usar encapsulación de datos para definir y encapsular las entradas. La validación de las entradas debe realizarse lo más pronto posible dentro del sistema.

Para esto, generalmente se utiliza un diccionario de datos en donde, además de indicar las restricciones anteriores, se define el tipo de dato a utilizar para almacenar la variable. Dentro de la misma validación de entradas, es importante considerar la validación de entradas que provengan de un almacén persistente, como puede ser datos provenientes de alguna base de datos, directorio o archivo.

Limpieza de Datos

Existen básicamente dos métodos para la limpieza de datos que son el listado negro y el listado blanco, Dentro de un listado negro se intenta excluir entradas que son inválidas mientras que en un listado blanco, sólo se aceptan entradas válidas. Generalmente se recomienda un listado blanco, ya que es más fácil detectar las entradas válidas que las no-válidas.

Para el caso de un listado negro, uno de los métodos utilizados es la sustitución de caracteres peligrosos por otros que no lo sean. El problema con este método es que se requiere que el programador identifique todos los caracteres peligrosos y la combinación de los mismos dependiendo de su contexto.

Por su parte, en un listado blanco, se define una lista de caracteres aceptables y se eliminan aquellos que no se encuentren en la lista. Esta lista de caracteres generalmente es predecible, lo que facilita su implementación.

Análisis estático.

Una de las prácticas más útiles para el desarrollo seguro de sistemas es la auditoría de código fuente. Ya que realizarlo de forma manual es complicado y está sujeto a errores, generalmente se utilizan herramientas automáticas que ayudan a detectar errores de implementación o malas prácticas de programación, sin embargo, no pueden detectar todos los problemas de seguridad de un sistema, por lo que no se recomienda que se utilice como una medida irrevocable de seguridad. Uno de los principales problemas con estos programas es que arrojan demasiados falsos-positivos.

Garantía de la Calidad.

Algunas técnicas para garantizar la calidad de un desarrollo de software que ayudan a la detección de defectos y vulnerabilidades son: pruebas de penetración, pruebas de fuzzing, auditorías de código, así como guías de desarrollo y mejores prácticas. Una prueba de penetración implica realizar diversas pruebas sobre una aplicación, sistema o red, desde el punto de vista del atacante. Es importante tomar en cuenta el análisis de riesgos realizado en las etapas del desarrollo anteriores, ya que sin estas, no es posible realizar una prueba de penetración exitosa ya que no se tiene un conocimiento adecuado sobre los activos, los riesgos y las vulnerabilidades de un sistema. Por su parte, una prueba de "fuzzing" es un método para encontrar huecos de seguridad proporcionando un conjunto de valores de entrada aleatorios sobre un sistema. Ayuda a garantizar una implementación correcta de la validación y limpieza de datos.

Actualmente existen diversos procesos bien definidos que intentan incorporan estas actividades dentro del ciclo de vida de desarrollo. Uno de los más conocidos es el desarrollado por Microsoft conocido como Security Development Lifecycle o SDL. Otro de los procesos conocidos, basado en el Proceso Unificado, es el llamado "Secure UP". En la siguiente sección se describe las etapas del proceso "Secure UP", donde se incluyen roles y actividades bien definidas para un desarrollo seguro de software.

3.3 Proceso Unificado Seguro (Secure UP).

El proceso de desarrollo de software tradicional y el mejor aceptado es el llamado proceso unificado, también conocido como RUP. Este proceso se conforma por un conjunto de actividades necesarias para transformar los requerimientos de usuario en un sistema de software.

Las características principales del proceso unificado son:

1. Basado en casos de uso.
2. Centrado en la arquitectura.
3. Iterativo e incremental.

Con esta metodología, el ciclo de vida de software se divide básicamente en 4 grandes fases llamadas concepción, elaboración, construcción y transición, a su vez, cada fase se divide en pequeñas iteraciones donde se aplica el procedimiento de desarrollo clásico también conocido como desarrollo en cascada. El desarrollo en cascada se divide en las etapas de captura de requerimientos, análisis, diseño, implementación, pruebas y despliegue. La Figura 25 muestra una visión general del ciclo de vida de desarrollo de software utilizando el proceso unificado.

En la etapa de captura de requerimientos se analizan tanto los requerimientos funcionales como los no-funcionales[14]. Algunos ejemplos de requerimientos no-funcionales tienen que ver con la precisión, las propiedades de la interfaz de usuario, desempeño, tiempo de respuesta, seguridad del sistema, entre otros.

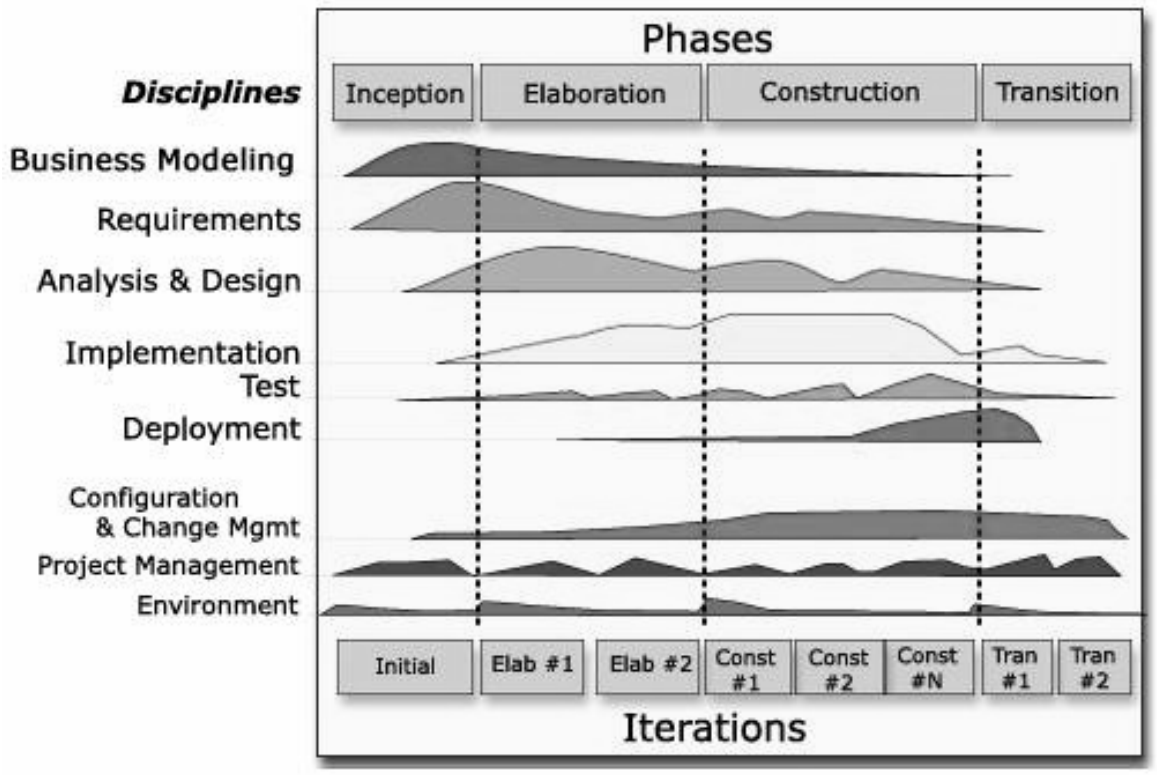


Figura 2: Fases del proceso unificado de sistemas.

Después de realizar la captura de requerimientos, el proceso unificado considera los requerimientos funcionales para el desarrollo del modelo de casos de uso, mientras que los requerimientos no-funcionales se anexan a una lista de requerimientos suplementarios que deben ser monitoreados de manera continua. Todo el proceso unificado está basado en los casos de uso derivados de los requerimientos funcionales, con lo cual se garantiza su cumplimiento, mientras que para los requerimientos no-funcionales no existe una integración directa en el proceso que pueda garantizar su cumplimiento.

[14] Requerimientos que tienen que ver con propiedades generales del sistema

Para incorporar los requerimientos de seguridad al proceso de desarrollo, existe un proceso de desarrollo basado en el proceso unificado que incorpora actividades orientadas al seguimiento de los requerimientos de seguridad a lo largo de todo el desarrollo. A continuación se describen las actividades definidas por este proceso.

1. **Requerimientos de seguridad:** En esta actividad, uno o varios analistas definen los requerimientos de seguridad imperativos al negocio, que pueden determinarse de acuerdo a ciertas regulaciones de la industria, políticas corporativas y necesidades específicas del negocio.

2. **Arquitectura de seguridad:** Esta actividad se enfoca en la creación de una arquitectura de seguridad global. Los arquitectos de software toman los requerimientos de seguridad determinados por los analistas y crean una propuesta de la arquitectura de seguridad. Esta actividad clasifica las decisiones arquitectónicas del sistema a través de procesos de análisis de riesgos y de compensación bien definidos, con el objetivo de identificarlos y mitigarlos. La arquitectura propuesta también identificará los patrones de seguridad que permitirá cumplir con todos los requerimientos de seguridad y los detallará, indicando los riesgos y vulnerabilidades conocidas. En esta actividad, también se creará una arquitectura prototipo y se perfeccionará antes de que se comience con la siguiente actividad. Es importante que en esta actividad se tomen en cuenta los requerimientos no-funcionales restantes con el objetivo de asegurarse que la arquitectura propuesta no los comprometerá.

3. **Diseño de seguridad:** Esta actividad toma la arquitectura propuesta en la etapa anterior y lo refina utilizando métodos como análisis de factor, análisis de capa, diseño de la política de seguridad, esbozo de las amenazas, clasificación de la información y etiquetado. Un desarrollador de seguridad 'senior' creará y documentará el diseño basado en la arquitectura propuesta en la etapa anterior, resultados de los análisis, tomando en cuenta las mejores prácticas y los riesgos de cada uno de los patrones.

4. **Implementación de seguridad:** En esta etapa, los desarrolladores expertos en seguridad, implementarán el diseño de seguridad. Un buen diseño de seguridad deberá separar los componentes de seguridad de los componentes de negocio, de tal manera que los desarrolladores de seguridad no tengan una fuerte interacción e integración con los desarrolladores de los componentes del negocio. Es importante que cada uno de los desarrolladores realicen la implementación del diseño siguiendo las mejores prácticas de código seguro.

5. **Pruebas de caja blanca:** En esta actividad se revisa el código y se buscan huecos de seguridad que pueden ser explotados. Se probarán diferentes ataques de seguridad orientados a comprometer el sistema, tratando de demostrar la manera en que se podría corromper los requerimientos de seguridad.

6. **Pruebas de caja negra:** En esta actividad, se realizan pruebas de seguridad intentando romper el sistema. En este caso, las pruebas se realizan por personal que no tiene conocimiento alguno acerca de su implementación. Se utilizan distintas herramientas y métodos para 'hackear' la aplicación.

7. **Monitoreo:** Esta es una actividad constante, se realiza en todo momento mientras la aplicación está en producción y se debe realizar en todos los niveles, no solo a nivel de aplicación, sino también a nivel perimetral.

8. **Auditoría de seguridad:** Esta actividad la realizarán auditores de sistemas. Estos buscarán que los sistemas cumplan con las regulaciones del negocio y de la industria.

Estas actividades se fusionan para formar la base de una infraestructura de seguridad robusta y cada una de ellas pertenece a diferentes fases del proceso unificado. El siguiente diagrama muestra la forma en que se conjugan con las actividades del proceso unificado.

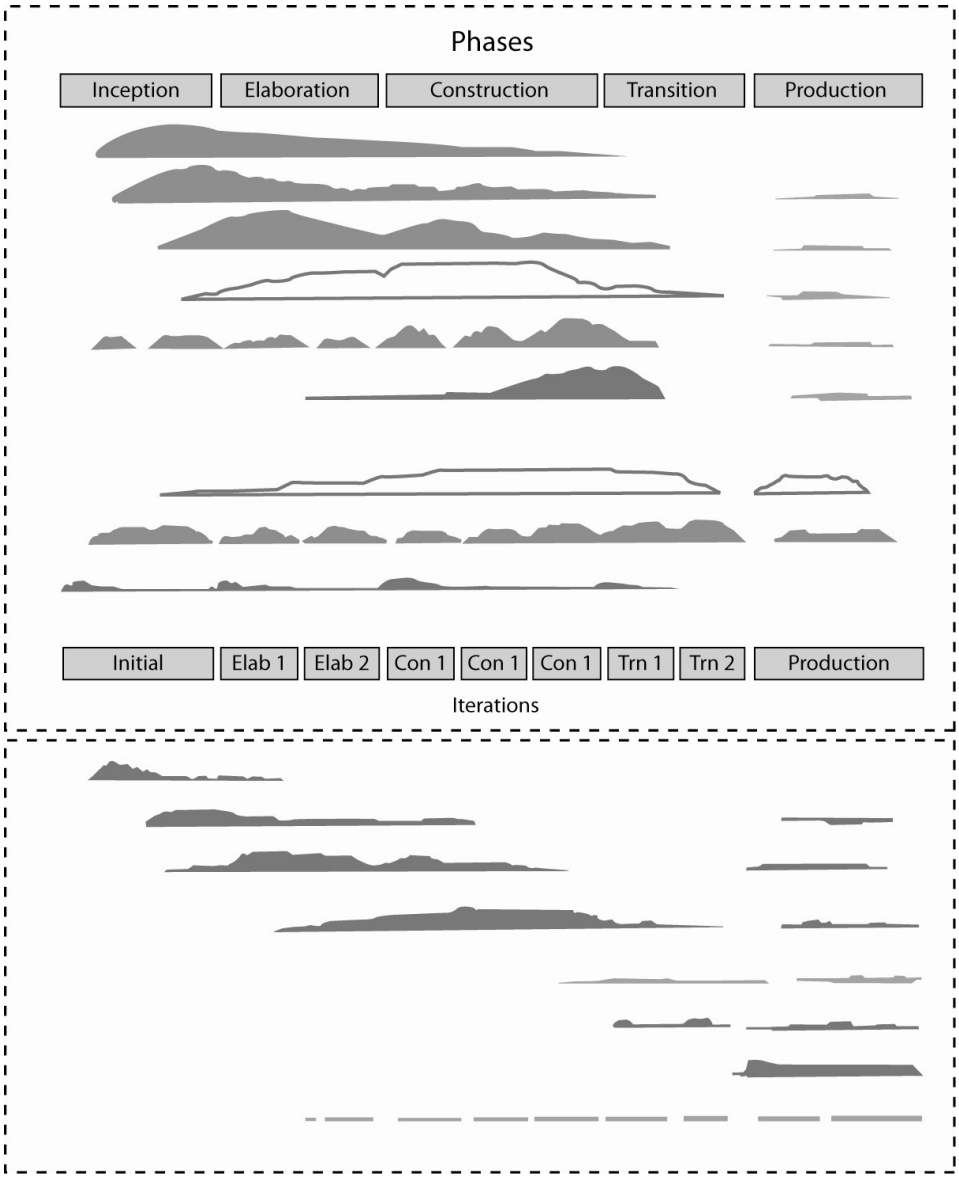


Figura 3: Actividades del Proceso Unificado Seguro

El diagrama siguiente muestra los roles y actividades en el proceso de desarrollo descrito anteriormente

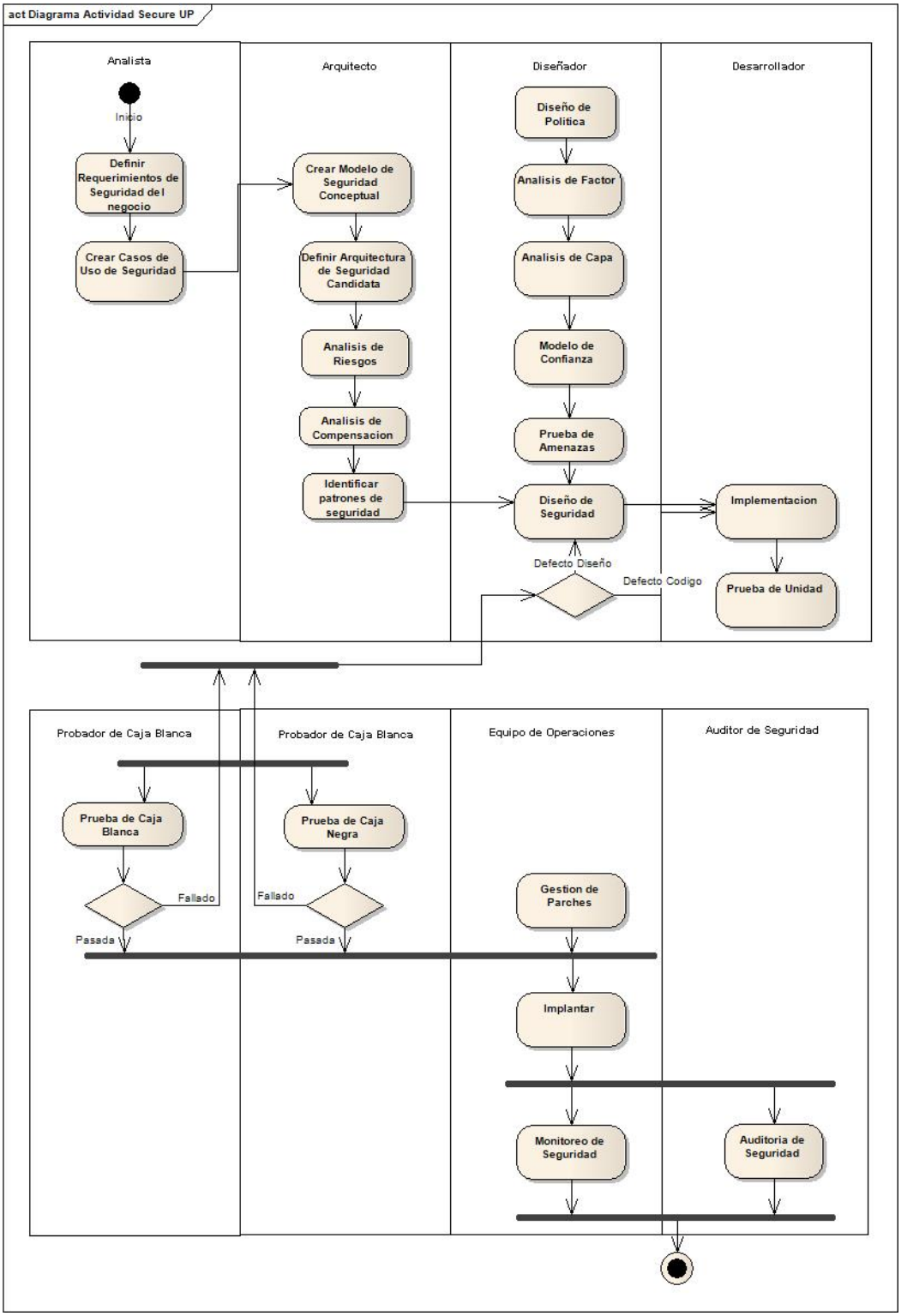


Figura 4: Diagrama de actividad del Proceso Unificado Seguro.

Como se muestra en el diagrama anterior, al inicio de ciclo de vida de una aplicación, el analista recoge los requerimientos de seguridad obligatorios. Una vez capturada la lista de requerimientos de seguridad, un arquitecto de seguridad crea un modelo conceptual y define una arquitectura de seguridad tentativa sin especificar detalles de la misma. En esta etapa se realiza un análisis de riesgos así como un análisis de compensación (TOA[15]). Un diseñador de seguridad toma este modelo y lo descompone para crear un diseño de seguridad que cumpla con todos los requerimientos de seguridad establecidos. Este diseño de seguridad deberá cumplir con otros requerimientos no-funcionales adicionales a los requerimientos de seguridad, análisis de factor, diseño de política de seguridad, análisis de capas, modelo de confianza y modelo de amenaza. Una vez diseñada la arquitectura de seguridad, los desarrolladores implementan el sistema teniendo siempre en la mente la seguridad del mismo. El analista toma el código desarrollado y realiza diversas pruebas de seguridad sobre el sistema para verificar que los requerimientos funcionales y no funcionales se cumplan. Un equipo especial realiza las pruebas de seguridad al sistema como son pruebas de penetración, escaneo de vulnerabilidades, crackeo de contraseñas, pruebas de código malicioso, pruebas de integridad de archivos, entre otras cosas. Dos equipos de pruebas realizan estas actividades en paralelo, un equipo con pleno conocimiento del sistema y con acceso al código del mismo, mientras que el otro equipo las realiza sin conocer el sistema. Si se encuentra algún hueco de seguridad en el sistema, se regresa el sistema al diseñador para que verifique si existe algún error en el diseño o en la implementación y se corrija. Cuando no se encuentren errores, el sistema se pondrá en producción y será permanentemente monitoreado y auditado, verificando el sistema en su totalidad.

En años recientes, las prácticas mencionadas anteriormente se han formalizado mediante estándares y mejores prácticas. Esta formalización es importante ya que las prácticas se basan en el análisis de las vulnerabilidades encontradas a través de la historia y el origen de las mismas. En la sección siguiente se describen los estándares actuales utilizados en el desarrollo seguro de sistemas.

[15] Trade-off Analysis (TOA): Análisis de compensación o balance.

3.4 Estándares actuales para el Desarrollo Seguro de Software

3.4.1 Modelo de Capacidad y Madurez SSE-CMM

Los modelos de capacidad y madurez proporcionan un modelo de referencia de prácticas maduras para una disciplina de ingeniería específica. Una organización utiliza estos modelos para comparar sus propias prácticas e identificar áreas potenciales de mejora. En este caso, los CMM's proporcionan definiciones de alto nivel y atributos claves para procesos específicos pero no proporcionan guías operacionales para realizar el trabajo específico, es decir, no definen un proceso pero si sus características. Las evaluaciones basadas en estos modelos están orientadas en la identificación de áreas de proceso que tengan ciertas debilidades o deficiencias y que se puedan mejorar.

El modelo SSE-CMM es un modelo de referencia de procesos que define los requerimientos para la implementación de seguridad dentro de uno o más sistemas relacionados. Es una compilación de las mejores prácticas conocidas para la ingeniería de seguridad.

Este modelo divide la ingeniería de seguridad en 3 áreas principales que son: riesgo, ingeniería y garantía. A continuación se presenta una breve explicación de cada una de estas áreas.

Riesgo.

El objetivo principal de la ingeniería de seguridad es reducir riesgos. Un riesgo se define como la probabilidad de que una acción o evento afecte la habilidad de una organización para lograr sus objetivos. En el área de seguridad de la información se expresa como la probabilidad de que una amenaza en particular explote cierta vulnerabilidad y provoque un resultado dañino.

La evaluación del riesgo es el proceso de identificar los problemas que no han ocurrido. Los riesgos se evalúan examinando la probabilidad de la amenaza y la vulnerabilidad[16], considerando el impacto potencial de que esto suceda. Asociado a la probabilidad está el factor de incertidumbre, que varía de acuerdo a la situación. Esto significa que la probabilidad solo se puede predecir dentro de ciertos límites, además, el impacto evaluado para un riesgo en particular también tiene un grado de incertidumbre asociado. Debido a este grado de incertidumbre, la planeación y justificación de la seguridad es una de las tareas más complicadas.

[16] De acuerdo al modelo de referencia SSE-CMM, las vulnerabilidades son propiedades del activo que pueden ser explotadas o aprovechadas por una amenaza y que se convierten en una debilidad.

La administración del riesgo es el proceso de evaluar y cuantificar el riesgo y de esta manera establecer un nivel de riesgo aceptable para una organización. Los riesgos se mitigan implementando barreras que pueden lidiar con la amenaza, la vulnerabilidad, el impacto o el riesgo en sí mismo, sin embargo, es importante mencionar que no es viable mitigar todos los riesgos de un sistema. Esto se debe en gran medida al costo relativo a la mitigación de riesgos y a la incertidumbre asociada. En este caso, el modelo SSE-CMM incluye actividades que garantizan que cierta organización analiza amenazas, vulnerabilidades, impactos y riesgos asociados.

Ingeniería.

La ingeniería de seguridad es un proceso que pasa por etapas distintas como son concepción, diseño, implementación, pruebas, implantación, operación, mantenimiento y retiro. Este proceso requiere estar dentro de un contexto más amplio y no solo considerar las características de seguridad por sí mismas, sino que debe considerar análisis de costos, desempeño, riesgos técnicos, así como facilidad de uso.

Garantía.

La garantía se define como el grado de confianza de que las necesidades de seguridad se han satisfecho. En este caso, el modelo SSE-CMM basa esta garantía en la repetición de los resultados del proceso de ingeniería de seguridad, asumiendo que una organización madura es más probable que repita sus resultados que una organización inmadura.

El modelo SSE-CMM se forma por 2 dimensiones conocidas como dominio y capacidad. El "dominio" está formado por el conjunto de prácticas que definen la ingeniería de seguridad. Estas prácticas se conocen como "prácticas base". Por otra parte, la dimensión de "capacidad" representa las prácticas que indican la gestión del proceso y la capacidad de institucionalización. Estas prácticas se conocen como "prácticas genéricas" ya que se aplican sobre distintos dominios de una organización.

El modelo SSE-CMM contiene 129 prácticas base, organizadas en 22 áreas de proceso. De todas estas, existen 61 prácticas base, dividas en 11 áreas de proceso, encargadas del área de ingeniería de seguridad, mientras que las 68 prácticas restantes, organizadas en 11 áreas de proceso, se encargan del dominio de la organización y el proyecto.

Las once áreas de proceso orientadas a la ingeniería de seguridad son las siguientes:

- PA01 Administración de los controles de seguridad.
- PA02 Evaluación del impacto.

- PA03 Evaluación del riesgo de la seguridad.
- PA04 Evaluación de las amenazas.
- PA05 Evaluación de las vulnerabilidades.
- PA06 Construcción de un argumento de garantía.
- PA07 Coordinación de la seguridad.
- PA08 Monitoreo de la postura de seguridad.
- PA09 Proporcionar entrada de seguridad.
- PA10 Especificar necesidades de seguridad.
- PA11 Verificar y validar la seguridad

Por su parte, las once áreas del modelo relacionadas con las prácticas organizacionales y de proyecto son las siguientes:

- PA12 Garantizar la calidad.
- PA13 Administración de la configuración.
- PA14 Administración del riesgo del proyecto.
- PA15 Monitoreo y control del esfuerzo técnico.
- PA16 Planeación del esfuerzo técnico.
- PA17 Definir el proceso de ingeniería de sistemas de la organización.
- PA18 Mejora del proceso de ingeniería de sistemas de la organización.
- PA19 Administración de la evolución de la línea del producto.
- PA20 Administración del ambiente de soporte de la ingeniería de sistemas.
- PA21 Proveer conocimiento y habilidades continuas.
- PA22 Coordinación con proveedores.

Las prácticas genéricas son actividades que se aplican en todos los procesos. Se encargan de los aspectos de gestión, medición e institucionalización de un proceso. Estas actividades se utilizan durante la evaluación para determinar la capacidad de una organización para realizar un proceso. Las prácticas genéricas se agrupan en áreas lógicas llamadas “características comunes” que se organizan en 5 niveles de capacidad. Estas prácticas se ordenan de acuerdo al nivel de madurez.

De esta forma, las características comunes y sus respectivos niveles son las siguientes:

- Nivel 1: Este nivel se enfoca en la realización de las prácticas base de manera informal.
    - 1.1 Se cumple con la realización de las prácticas base
- Nivel 2: Se enfoca en los asuntos de definición, planeación y ejecución de un proyecto.
    - 2.1 Planeación de la ejecución.
    - 2.2 Ejecución disciplinada.
    - 2.3 Verificación de la ejecución.
    - 2.4 Monitoreo de la ejecución.
- Nivel 3: Se enfoca en una adaptación disciplinada de los procesos definidos a nivel de organización.
    - 3.1 Definir un proceso estándar.
    - 3.2 Realizar un proceso definido.
    - 3.3 Coordinación del proceso
- Nivel 4: Su propósito en empatar las medidas del proceso con los objetivos propios del negocio.
    - 4.1 Establecer objetivos de calidad medibles.
    - 4.2 Gestión de la ejecución objetiva
- Nivel 5: Mejora constante de las prácticas realizadas.
    - 5.1 Mejoramiento de la capacidad organizacional.

De esta manera podemos decir que el modelo SSE-CMM es un conjunto de prácticas básicas que realizadas a distintos niveles ayudan a garantizar un nivel de seguridad aceptable para cierto proyecto u organización. La relación de las 2 dimensiones del modelo (dominio y capacidad) se pueden abstraer mediante la siguiente matriz.

	Áreas de proceso																								
5.2 Mejoramiento de la efectividad de los procesos																									
5.1 Mejoramiento de la capacidad de la organización																									
4.2 Gestión de la ejecución objetiva.																									
4.1 Establecer objetivos de calidad medibles.																									
3.3 Coordinación del proceso																									
3.2 Realizar un proceso definido																									
3.1 Definir un proceso estándar.																									
2.4 Monitoreo de la ejecución																									
2.3 Verificación de la ejecución.																									
2.2 Ejecución disciplinada.																									
2.1 Planeación de la ejecución.																									
1.1 Realización de las prácticas base																									
Características Comunes	PA01 - Administrar controles de seguridad.	PA02 - Evaluación del impacto	PA03 Evaluación del riesgo de la seguridad	PA04 Evaluación de las amenazas.	PA05 Evaluación de las vulnerabilidades.	PA06 Construcción de un argumento de garantía.	PA07 Coordinación de la seguridad	PA08 Monitoreo de la postura de seguridad.	PA09 Proporcionar entrada de seguridad.	PA10 Especificar necesidades de seguridad.	PA11 Verificar y validar la seguridad	PA12 Garantizar la calidad.	PA13 Administración de la configuración.	PA14 Administración del riesgo del proyecto.	PA15 Monitoreo y control del esfuerzo técnico.	PA16 Planeación del esfuerzo técnico.	PA17 Definir el proceso de ing. de sistemas de la organización.	PA18 Mejora del proceso de ing. de sistemas de la organización.	PA19 Administración de la evolución de la línea del producto.	PA20 Administración del ambiente de soporte de la ing. de sistemas.	PA21 Proveer conocimiento y habilidades continuas.	PA22 Coordinación con proveedores.		

Figura 5: Matriz de dominio y capacidad del modelo SSE-CMM

Como se puede reflejar en este modelo de referencia, la seguridad de los sistemas debe formar parte del proceso completo del equipo de ingeniería y de una organización en su totalidad. Para este trabajo se describirán en mayor detalle las actividades relacionadas con las áreas de proceso de la ingeniería de seguridad, mientras que las actividades del área de proceso organizacional y proyecto solo se mencionan de forma general.

Como se mencionó anteriormente, existen 11 áreas de proceso orientadas a la ingeniería de seguridad, donde se definen 61 actividades. Las áreas de proceso son las siguientes.

PA01 Administrar controles de seguridad: Su propósito es garantizar que los controles de seguridad se configuraron y se utilizan de manera correcta.	
BP.01.01 Establecer responsabilidades de seguridad.	Establecer y comunicar responsabilidades de seguridad en toda la organización, de tal forma que los responsables de la misma tengan el poder para realizar las acciones de manera clara y responsable.
BP.01.02 Gestión de la configuración de la seguridad.	Administrar la configuración de los controles de seguridad del sistema como pueden ser actualizaciones o cambios realizados a dispositivos de hardware, software, así como los distintos procedimientos.
BP.01.03 Realización de programas de educación, entrenamiento y conciencia.	Realizar programas de educación, entrenamiento y conciencia para todos los usuarios y administradores, así como almacenar registros de los mismos y sus resultados.
BP.01.04 Administración de servicios de seguridad y mecanismos de control.	Realizar mantenimientos periódicos y administrar servicios de seguridad y mecanismos de control. Llevar el registro y control de cada uno de los mantenimientos realizados, excepciones de seguridad realizadas, motivos, duración.
PA02 Evaluación del Impacto: Su objetivo es identificar los impactos posibles al sistema en cuestión, así como evaluar la probabilidad de que ocurran. Pueden ser impactos tangibles o no.	
BP.02.01 Priorizar capacidades.	Identificar, analizar y priorizar las capacidades operacionales, de negocio y misión del sistema.
BP.02.02 Identificar activos del sistema.	Identificar los recursos del sistema y los datos necesarios para soportar los objetivos de seguridad o las capacidades claves del sistema.
BP.02.03 Seleccionar métricas del impacto.	Seleccionar las métricas de impacto a utilizarse por la evaluación.
BP.02.04 Identificar la relación de métricas.	Evaluar cada impacto relacionando distintas métricas para obtener un resultado consolidado.
BP.02.05 Identificar y caracterizar los impactos.	Una vez identificados los activos y las capacidades, identificar las consecuencias que podrían causar daño, así como las métricas a utilizarse para cada impacto.
BP.02.06 Monitoreo del impacto.	Monitoreo continuo de los impactos existentes y verificación continua de nuevos impactos potenciales.

PA03 Evaluación del riesgo de seguridad: Identificar los riesgos de seguridad asumidos al depender de un sistema en un ambiente definido. Se identifica y se evalúa la probabilidad de ocurrencia de una exposición[17].	
BP.03.01 Selección del método de análisis de riesgos.	Seleccionar los criterios, técnicas y métodos a utilizarse para analizar, evaluar y comparar los riesgos de seguridad.
BP.03.02 Identificación de exposiciones.	Identificar las amenazas y vulnerabilidades que preocupan al sistema e identificar el impacto de la ocurrencia de alguna de ellas. Estos ayudarán a seleccionar las barreras que permitirán proteger el sistema.
BP.03.03 Evaluación de la exposición del riesgo.	Identificar la probabilidad de la ocurrencia de una exposición.
BP.03.04 Evaluación de la incertidumbre total.	Evaluar la incertidumbre total asociada con los riesgos y amenazas identificadas.
BP.03.05 Priorizar riesgos.	Ordenar los riesgos identificados de acuerdo a la prioridad de la organización, probabilidad de la ocurrencia, la incertidumbre asociada y los fondos disponibles. Un riesgo se puede mitigar, evitar, transferir o aceptar o una combinación de todos lo anteriores. Esto se debe seleccionar de acuerdo a las necesidades de los *'stakeholders',* prioridades del negocio y la arquitectura completa del sistema.
BP.03.06 Monitorear los riesgos y sus características.	Monitorear los cambios constantes en el conjunto de riesgos, así como cambios en sus características.
PA04 Evaluación de Amenazas: Su objetivo es la de identificar las amenazas, así como sus propiedades y características.	
BP.04.01 Identificación de las amenazas naturales.	Identificar las amenazas a la organización que tengan un origen natural, como puede ser terremotos, tsunamis, tornados, según sea el caso
BP.04.02 Identificación de amenazas hechas por el hombre.	Identificar aquellas amenazas originadas por el hombre, ya sea de manera accidental o intencional, que apliquen para el sistema en cuestión.
BP.04.03 Identificar las unidades de medida de las amenazas.	Identificación de las unidades de medida apropiadas, así como los rangos aplicables para cada una de las amenazas identificadas dentro del ambiente especificado.
BP.04.04 Identificación de capacidad del agente amenaza.	Se enfoca en identificar las habilidades y capacidades de un adversario (ser humano) potencial para realizar un ataque en contra del sistema en cuestión.
BP.04.05 Evaluación de la probabilidad de la amenaza.	Evaluar la probabilidad de que un evento de amenaza ocurre, tanto para las amenazas naturales como las amenazas hechas por el hombre.

[17] Una exposición se refiere a la combinación de amenaza, vulnerabilidad e impacto que puedan causar un daño significativo.

BP.04.06 Monitoreo de las amenazas y sus características.	Monitorear los cambios constantes en el conjunto de amenazas y cambios en sus características.
PA05 Evaluación de vulnerabilidades: Su propósito es identificar y caracterizar las vulnerabilidades[18] de seguridad del sistema. Incluye un análisis de los activos del sistema.	
BP.05.01 Selección del método de análisis de vulnerabilidades.	Seleccionar los métodos, técnicas y criterios que se utilizaran para identificar y caracterizar las vulnerabilidades de un sistema.
BP.05.02 Identificación de vulnerabilidades.	Identificar las vulnerabilidades del sistema ya sea en partes con funciones de seguridad o en aquellas que no las tengan. Es importante llevar un registro de las vulnerabilidades descubiertas.
BP.05.03 Reunir datos de las vulnerabilidades.	Reunir información acerca de las vulnerabilidades encontradas. Esta información puede ser unidades de medida, facilidad de explotar la vulnerabilidad y su probabilidad.
BP.05.04 Sintetizar las vulnerabilidades del sistema.	Evaluación de las vulnerabilidades del sistema encontradas y las relaciones entre ellas, probabilidades de ocurrencia, así como recomendaciones sobre las mismas.
BP.05.05 Monitoreo de las vulnerabilidades y sus características.	Monitoreo constante de los cambios en las vulnerabilidades y sus características. Incluyen reportes de monitoreo de vulnerabilidades y reportes de cambio de vulnerabilidades.
PA06 Construcción de un argumento de aseguramiento: Su propósito es expresar el cumplimiento de las necesidades de seguridad del cliente.	
BP.06.01 Identificar objetivos de aseguramiento.	El cliente determina los objetivos de aseguramiento e identifica el nivel de confianza necesaria por el sistema.
BP.06.02 Definir estrategia de aseguramiento.	Definición de un plan o estrategia de protección de los objetivos identificados.
BP.06.03 Control de la evidencia de aseguramiento.	Identificación y control de las evidencias de garantías de seguridad. Puede ser dentro de distintos repositorios de información como bases de datos, cuadernos de trabajo, bitácoras del sistema, etc.
BP.06.04 Analizar evidencia.	Realización de análisis de las evidencias tomadas que permitan determinar que los controles de seguridad y mecanismo implementados cumplen con las necesidades del cliente.
BP.06.05 Proporcionar argumento de garantía de seguridad.	Proporcionar un argumento de garantía de seguridad, basado en las evidencias, que demuestre el cumplimiento de los objetivos de necesidad establecidos por el cliente.

[18] Una vulnerabilidad es un aspecto de un sistema que puede ser explotada para propósitos distintos a los que fue creado originalmente.

PA07 Coordinación de la Seguridad: Su objetivo es asegurarse que todas las partes involucradas en la seguridad del sistema están consientes e involucrados con las actividades de seguridad.	
BP.07.01 Definición de los objetivos de la coordinación	Definir las relaciones y los objetivos en la coordinación de la ingeniería en seguridad.
BP.07.02 Identificación de los mecanismos de coordinación.	Identifica los mecanismos que ayuden a comunicar y coordinar las decisiones y recomendaciones de seguridad entre los diferentes grupos de trabajo.
BP.07.03 Facilidad de la coordinación.	Facilita la coordinación entre los distintos miembros de la ingeniería de seguridad.
BP.07.04 Coordinación de las recomendaciones y decisiones de seguridad.	Comunicar las recomendaciones y decisiones de seguridad entre los diferentes ingenieros de seguridad, grupos de ingeniería, entidades externas y otras partes apropiadas.
PA08 Monitoreo de la postura de seguridad: Su propósito es asegurarse que todas las posibles brechas de seguridad se identifican, se reportan y se responden de manera adecuada.	
BP.08.01 Análisis de registro de eventos.	Examinar registros de eventos históricos que revelen información de seguridad importante. Adicionalmente a los eventos, es importante identificar los factores que los relacionan.
BP.08.02 Monitoreo de cambios.	Buscar cambios en las amenazas, vulnerabilidades e impactos que puedan alterar la efectividad de la postura de seguridad, ya sea de manera positiva o negativa.
BP.08.03 Identificar incidentes de seguridad.	Determinar si han ocurrido incidentes de seguridad, sus detalles y realizar reportes de los mismos.
BP.08.04 Monitoreo de las barreras de seguridad.	Monitorear del desempeño y la efectividad de las barreras de seguridad, así como identificar los cambios en el mismo desempeño.
BP.08.05 Revisión de la postura de seguridad.	Debido a que la postura de seguridad está sujeta a cambio constante basado en el ambiente de amenazas, requerimientos de seguridad y configuración del sistema, es importante re-examinar las razones de seguridad y los requerimientos.
BP.08.06 Gestión de la respuesta a incidentes de seguridad.	En muchos casos la disponibilidad del sistema es muy importante. Muchos eventos no pueden prevenirse por lo que es importante tener un plan de contingencia en caso de interrupción inesperada del servicio. En estos casos, es importante identificar el periodo máximo de no funcionamiento, así como los elementos esenciales del sistema. Deberá incluir planes de mantenimiento, planes de prueba y una estrategia de recuperación sel mismo.
BP.08.07 Protección de los artefactos de monitoreo de seguridad.	Es importante garantizar la protección de todos los artefactos relacionados con el monitoreo de la seguridad, de otra forma, no tendrían valor.

PA09 Proporcionar entrada de seguridad: Su propósito es la proporcionar de información de seguridad necesaria a todos los diseñadores, desarrolladores y usuarios. Incluye arquitectura de seguridad, diseño, alternativas de implementación y guías de seguridad.	
BP.09.01 Comprender las necesidades de entrada de la seguridad.	La ingeniería de seguridad se coordina con otras disciplinas para determinar los tipos de entradas de seguridad. Estas pueden ser guías, diseños, documentos o ideas relacionados con la seguridad.
BP.09.02 Determinar consideraciones y restricciones de seguridad.	Identificar todas las consideraciones y restricciones de seguridad necesarias para realizar elecciones de ingeniería bien informadas. El grupo de ingeniería de seguridad realizará análisis para determinar cualquier consideración o restricción de seguridad en los requerimientos, diseño, implementación, configuración y documentación. Es importante considerar que las restricciones pueden ser positivas (hacerlo siempre) o negativas (nunca hacerlo).
BP.09.03 Identificar alternativas de seguridad.	Identifica soluciones alternativas para los problemas de ingeniería es cuestión de seguridad. Es un proceso iterativo y transforma los requerimientos de seguridad en implementaciones. Pueden proporcionar alguna arquitectura, modelos o prototipos que proporcionen una solución alternativa a los requerimientos de seguridad.
BP.09.04 Analizar las alternativas de ingeniería de seguridad.	Su objetivo es analizar y priorizar alternativas de seguridad. Utilizando las restricciones y consideraciones de seguridad identificadas, los ingenieros de seguridad evalúan cada alternativa y realizan una recomendación al grupo de ingeniería.
BP.09.05 Proporcionar una guía de ingeniería de seguridad.	Desarrollo de guías de seguridad y proporcionarlas a los diferentes grupos de seguridad. Ayudarán en caso de toma de decisiones sobre la arquitectura, diseño e implementación.
BP.09.06 Proporcionar una guía operacional de seguridad.	Desarrollo de una guía de seguridad orientada a los usuarios y administradores del sistema. Contiene información para la instalación, configuración, operación y retiro del sistema, de manera segura. Es recomendable que esta comience en las etapas iniciales del desarrollo del sistema.
PA10 Especificar necesidades de seguridad: Identificar las necesidades de seguridad del sistema. Define la base de seguridad que cumpla con los requerimientos organizacionales, políticas y legales en cuestiones de seguridad.	
BP.10.01 Ganar comprensión de las necesidades de seguridad del cliente.	Colectar toda la información necesaria para la comprensión de las necesidades de seguridad del cliente.
BP.10.02 Identificar restricciones, políticas y leyes aplicables.	Reunir información acerca de todas las entidades externas que influyen en la seguridad de un sistema. Identificación de leyes, regulaciones, políticas y

	estándares comerciales que gobiernan el ambiente destino del sistema. Es importante indicar el orden de precedencia entre leyes internas y externas según sea el caso.
BP.10.03 Identificar el contexto de seguridad del sistema.	Identificar el contexto del sistema y de que manera impacta la seguridad del mismo. Requiere la evaluación de los requerimientos funcionales y desempeño y posibles amenazas e impactos de seguridad. Al mismo tiempo, requiere de la revisión de las restricciones operacionales para conocer sus implicaciones en la seguridad del sistema.
BP.10.04 Capturar vista de seguridad de la operación del sistema.	Realizar una vista de alto nivel de la seguridad de una empresa que incluya roles, responsabilidades, flujos de información, activos, recursos, protección del personal y protección física
BP.10.05 Capturar objetivos de seguridad de alto nivel.	Identificar los objetivos de seguridad que deben cumplirse para proporcionar una seguridad apropiada para el sistema en su ambiente operacional.
BP.10.06 Definir requerimientos relacionados a la seguridad.	Definir requerimientos de seguridad del sistema. Debe garantizar que estos requerimientos son consistentes con las políticas aplicables, leyes, estándares y limitantes del sistema. Se requiere considerar que la seguridad impacta lo menos posible la funcionalidad y desempeño del sistema. Debe considerar requerimientos de seguridad técnicos y no-técnicos.
BP.10.07 Obtención de acuerdo de seguridad.	
PA11 Verificar y validar la seguridad: Su objetivo es garantizar que las soluciones de seguridad cumplen con los requerimientos de seguridad, arquitectura y diseño utilizando la observación, demostración análisis y pruebas. Las soluciones se validan de acuerdo a las necesidades de seguridad operacional del cliente.	
BP.11.01 Identificar los objetos de verificación y validación.	Identificar los objetos de las actividades de verificación y validación. La verificación demuestra que la solución se implemento correctamente mientras que la validación demuestra lo efectiva de la solución.
BP.11.02 Definir métodos de verificación y validación.	Define los métodos de validación y verificación del sistema. Se define la manera en que cada requerimiento de seguridad se verificará y validará.
BP.11.03 Realizar verificación.	Verificar que la solución implementa los requisitos asociados utilizando los métodos definidos por la práctica base anterior.
BP.11.04 Realizar validación de seguridad.	Validar que la solución satisface las necesidades asociadas al nivel de abstracción anterior.
BP.11.05 Proporciona resultados de verificación y validación.	Captura y proporciona los resultados de la verificación y validación a los distintos grupos de ingeniería.

3.4.2 Common Criteria.

Su nombre completo es “Common Criteria for Information Technology Security Evaluation” aunque es mejor conocido con su nombre corto “Common Criteria”. Es un estándar para la evaluación de seguridad desarrollado en 1996. Se documenta en tres secciones: la introducción describe su historia, propósito y los conceptos y principios generales de la evaluación de la seguridad y describir el modelo de evaluación; la segunda sección describe un conjunto de requerimientos funcionales de seguridad que los usuarios de productos quieren especificar y que funcionan como plantillas estándares para los requerimientos funcionales de seguridad. Los requerimientos funcionales se catalogan y clasifican proporcionando un menú de requerimientos funcionales de seguridad que los usuarios pueden seleccionar. La tercera sección del documento incluye requerimientos para garantizar la seguridad que incluye distintos métodos para garantizar que un producto es seguro. En esta sección también se incluyen siete conjuntos predefinidos de requerimientos de garantía conocidos como Evaluation Assurance Levels (EAL).

Existen dos artefactos que se crean durante la evaluación CC: un perfil de protección (PP) y un objetivo de seguridad (ST). Ambos documentos se crean basados en plantillas específicas proporcionadas por el estándar. Un perfil de protección (PP) identifica las propiedades de seguridad deseadas (requerimientos de seguridad de usuario) para un tipo de producto específico. Los perfiles de protección se crean seleccionando componentes apropiados de la sección dos del Common Criteria, ya que hay muchas probabilidades de que los requerimientos de usuario ya existen dentro del catálogo de requerimientos. Los perfiles de protección representan las necesidades de seguridad para un tipo de producto independientemente de la implementación y pueden incluir tanto los requerimientos funcionales como sus garantías. Por su parte, un objetivo de seguridad (ST) es una declaración de las necesidades de seguridad de un producto en específico dependiendo de la implementación.

De esta manera, la evaluación de un producto siguiendo el Common Criteria se realiza de la siguiente manera:

1. Una organización que desea adquirir o desarrollar un producto de seguridad específico, define sus necesidades de seguridad utilizando un perfil de protección (PP). Este artefacto es evaluado y publicado.
2. El desarrollador del producto toma el perfil de protección, escribe un Security Target que cumpla con los requerimientos del Protection Profile y se evalúa.
3. El desarrollador del producto construye un TOE (Target Of Evaluation) y se evalúa comparándolo con el Security Target.

Los 7 niveles de evaluación son los siguientes:

1. Evaluation Assurance Level 1 (EAL1) Probado funcionalmente.
2. Evaluation Assurance Level 2 (EAL2) Probado estructuralmente.
3. Evaluation Assurance Level 3 (EAL3) Probado y verificado metódicamente.
4. Evaluation Assurance Level 4 (EAL4) Diseñado, probado y revisado metódicamente.
5. Evaluation Assurance Level 5 (EAL5) Diseñado y probado semi-formalmente.
6. Evaluation Assurance Level 6 (EAL6) Diseñado verificado semi-formalmente y probado.
7. Evaluation Assurance Level 7 (EAL7) Diseño verificado formalmente y probado.

Una vez conocidas las prácticas de seguridad recomendadas para el desarrollo seguro de sistemas y conociendo los estándares principales disponibles en la actualidad, es necesario abordar uno de los elementos principales en cualquier desarrollo de sistemas. La etapa de implementación de un desarrollo de software es una de las etapas más importantes de todo el proceso, y es en donde se aplican todas las actividades relacionadas con la ingeniería de seguridad. En la actualidad existen cientos de lenguajes de programación, cada uno con características particulares y orientadas para un tipo de aplicación en particular, sin embargo uno de los lenguajes mejor adoptados es el lenguaje Java. Este lenguaje es muy conocido por que se dice que es un lenguaje “seguro”, sin embargo, como cualquier lenguaje de programación, la seguridad no se logra por el lenguaje de programación en sí mismo, si no depende de un análisis en la materia a distintos niveles de la organización, sin embargo, a nivel de diseño de lenguaje, Java incorpora características importantes para un desarrollo seguro. En el siguiente capítulo, se describen las características de seguridad más importantes del lenguaje que, con una correcta utilización, permiten disminuir vulnerabilidades importantes en los sistemas.

Capítulo 4: Java como lenguaje de programación de software seguro.

Resumen: Una de las etapas más importantes en el desarrollo de software es la implementación. Al momento que se llega a esta etapa, se ha elegido un lenguaje de programación y la arquitectura del sistema está prácticamente definida en su totalidad. Es de suponerse que para el caso de los sistemas con requerimientos de seguridad importantes, se ha definido un diseño seguro, por lo que la implementación, que corresponde a un refinamiento del diseño, deberá realizarse siguiendo las mejores prácticas de programación, tomando en cuentas las características del lenguaje elegido. Como es sabido, la gran mayoría de los problemas de seguridad se derivan de malas prácticas de programación y en muchas ocasiones del desconocimiento de las ventajas y desventajas del lenguaje a utilizar. En este capítulo, se toma como ejemplo el lenguaje de programación Java, considerado uno de los lenguajes más seguros en la actualidad; se describen sus características básicas, su arquitectura de seguridad, API's criptográficas y de seguridad útiles, así como las mejores recomendaciones al momento de programar aplicaciones con este lenguaje.

4.4 Introducción.

El lenguaje de programación JAVA se desarrolló con la idea de crear una plataforma que permitiera ejecutar aplicaciones de manera heterogénea, es decir, que una vez realizado el software, éste pueda ejecutarse de manera transparente en cualquier plataforma, independiente de su arquitectura y/o sistema operativo. Para cumplir con este objetivo, la plataforma JAVA está compuesta por el lenguaje de programación JAVA, la máquina virtual o JVM y el conjunto de API's de programación. De esta manera, todo programa desarrollado bajo el lenguaje de programación JAVA debe pasar por dos etapas conocidas como "etapa de compilación" y "etapa de interpretación". En la primera etapa, el compilador de JAVA se encarga de transformar el código fuente, desarrollado en un lenguaje de alto nivel, a un código de instrucciones propio de la especificación JAVA conocido como código intermedio, código binario o 'bytecode'[19]. Por su parte, la etapa de interpretación hace de la máquina virtual JAVA (intérprete) que se encarga de traducir cada una de las instrucciones de código intermedio a un conjunto de instrucciones propio de la máquina donde se ejecuta.

[19] El formato de los archivos creados en esta etapa es conocido como formato 'class'

El lenguaje de programación Java se desarrolló en una época en la que el uso de computadoras en red comenzaba a extenderse, por lo que su arquitectura se debió adaptar a estas necesidades. La posibilidad de distribuir código intermedio a través de la red (código móvil) originó la necesidad de incorporar características de seguridad importantes en las etapas iniciales del lenguaje, por lo que ahora se considera que el lenguaje Java ha sido diseñado para disminuir considerablemente gran parte de los problemas de seguridad, comparado con otros lenguajes de programación ampliamente utilizados como C o C++. Algunas de las características importantes propias de la plataforma JAVA son las siguientes:

- Manejo seguro de tipos de datos: A diferencia de otros lenguajes de programación donde no se hace una verificación de la longitud de los tipos de datos en tiempo real, la infraestructura Java incluye un proceso de verificación en tiempo real.
- Evita el uso de punteros: Los punteros son variables que se encargan de almacenar direcciones de memoria, lo que permite acceder con facilidad a áreas de memoria no deseables y a partir de ahí poder modificar el comportamiento normal de un sistema. Gran parte de los *exploits* de software comienzan con una alteración en el manejo de punteros. El hecho de que la plataforma Java no permita el uso de punteros, minimiza gran parte de los problemas de seguridad en los sistemas, ya que imposibilita el acceso a áreas de memoria de uso reservado.
- Existencia de un sistema '*Garbage Collector*' para la administración de memoria automática en tiempo real.
- Uso de un objeto '*ClassLoader*' que se encarga, entre otras cosas, de separar las clases mediante los llamados *namespaces,* lo que evita que dos aplicaciones que se ejecutan en la misma máquina virtual Java, compartan áreas de memoria.
- Uso de un sandbox por default.. Cuando no se define un gestor de seguridad para cierta aplicación, la máquina virtual Java define una política de seguridad por default conocida como sandbox, donde código JAVA remoto se ejecuta con altas restricciones de seguridad.
- Incluye mecanismos de verificación que aseguran la ejecución de código java legítimo.
- Incluye un conjunto de API´s que proporcionan implementaciones de algoritmos de seguridad, mecanismos y protocolos de uso común.

La anatomía general de una aplicación JAVA se muestra en el diagrama siguiente:

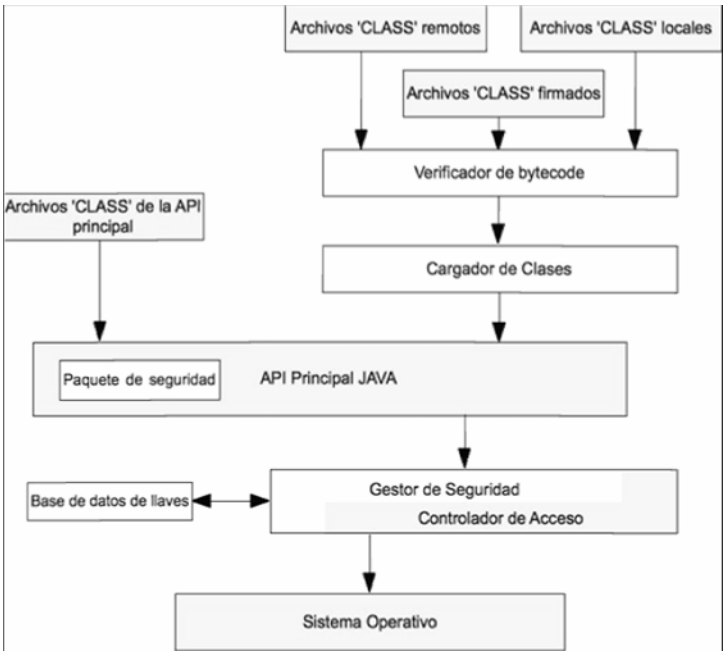


Figura 1: Anatomía de una aplicación JAVA.

En la tabla siguiente se describe cada uno de estos elementos.

Tabla 1: Descripción de los elementos de la plataforma JAVA.

Componente	Descripción
Verificador de código	*Es el encargado de asegurarse que todos los archivos 'class' cumplen con las reglas del lenguaje. No todas las clases están sujetas al verificador de código.*
Cargador de clases	*Es el responsable de cargar las clases en memoria. Pueden establecer los permisos para cada clase al momento de cargarse.*
Controlador de Acceso	*Otorga o niega los accesos al sistema operativo basados en un conjunto de políticas.*
Gestor de Seguridad	*Misma función que el controlador de acceso, aunque permanece en la plataforma por cuestiones históricas.*
Paquete de Seguridad	*Clases que pertenecen al paquete 'java.security' así como en cada una de sus extensiones como son JCE, JSSE, JAAS que proporcionan funciones criptográficas al lenguaje.*
Base de datos de llaves	*Conjunto de llaves utilizadas por la infraestructura de seguridad para crear o verificar las firmas digitales.*

En las secciones siguientes, se describen los componentes de la plataforma mencionados de manera más detallada.

4.5 Seguridad del lenguaje JAVA y proceso de verificación.

La integridad de los recursos de memoria es una de las características de seguridad más importantes de la arquitectura JAVA y para garantizarla, todo programa JAVA debe pasar por diferentes etapas de verificación donde se debe analizar lo siguiente:

- **Verificación del control de acceso:** Cada entidad (clase, variable, método) tiene un nivel de acceso asociado que puede ser *'private'*, *'protected'*, *'public'* o acceso por *'default'*. A diferencia de otros lenguajes de programación, la verificación de los niveles de acceso se realiza en tiempo de ejecución, no solo en tiempo de compilación, exceptuando para este caso los objetos serializados. Se dice que serializamos un objeto cuando almacenamos el estado actual de un objeto como una serie de bytes y lo utilizamos para crear el mismo objeto, con su mismo estado, en alguna otra aplicación a partir de este flujo de bytes. Esta funcionalidad es muy utilizada en las aplicaciones cliente-servidor cuando se desea intercambiar objetos. Almacenar el estado de un objeto significa inevitablemente almacenar gran parte de sus variables privadas, ya que sólo de esta manera se logra reconstruir nuevamente. El uso de esta funcionalidad puede anular completamente la protección dada por los niveles de acceso, ya que la información privada, por ejemplo, puede ser alterada en algún otro sitio fuera de la máquina virtual, sin embargo, el lenguaje de programación Java ofrece alternativas que permiten implementar técnicas más seguras de serialización.

- **Imposibilidad de acceder a direcciones de memoria arbitrarias:** Como se mencionó anteriormente, el lenguaje de programación Java no permite el uso de punteros para el direccionamiento de memoria arbitraria.

- **Entidades declaradas *'final'* no deben cambiar:** Esta es una característica muy importante, ya que gran parte de las clases del sistema son declaradas *'final'*. Si no se garantizara esto, sería posible alterar gran parte del funcionamiento de la máquina virtual.

- **Imposibilidad de utilizar variables sin antes inicializarse:** En Java, todas las variables deben de inicializarse antes de utilizarse ya sea con un valor específico o con un valor por default.

- **Verificación de los límites de los arreglos en cada acceso:** Esta característica también tiene que ver con los accesos a memoria arbitrarias, ya que si pudiéramos acceder a direcciones de memoria más allá de los límites del arreglo, podríamos cambiar el valor de algunas variables, por ejemplo el valor de una constante, ó acceder a información privada.

- **Estrictas reglas de conversión de objetos:** Sólo se puede realizar un '*cast*' entre objetos que forman parte del mismo árbol jerárquico. Si no fuera de esta manera, podríamos crear objetos falsos que engañen a la máquina virtual y acceder o realizar operaciones no válidas.

Para que estas características se cumplan, el lenguaje de programación Java ofrece mecanismos de verificación en 3 etapas distintas, realizadas a lo largo del desarrollo y ejecución de aplicaciones. Estas etapas son la etapa de compilación, etapa de enlace y etapa de ejecución. No es posible realizar todas las pruebas en cada una de las etapas, sin embargo, se garantiza su ejecución en al menos una de ellas.

Gran parte de las reglas del lenguaje se pueden verificar en etapa de compilación, sin embargo hay algunas en las que no se posee de suficiente información sino hasta el momento de su ejecución, como por ejemplo la verificación de los accesos a arreglos o el '*casting*' de objetos no del todo conocidos. Por otro lado, debido a que el lenguaje de programación Java utiliza un código intermedio para su ejecución, todas las reglas del lenguaje deben verificarse en alguna etapa distinta a la de compilación. Sin esta etapa, no habría forma de garantizar que el código a ejecutar es un código Java válido. Esta etapa es conocida como etapa de ligado o cargado de clases. Es muy similar a la etapa de ligado de otros lenguajes de programación, sin embargo en la plataforma Java se realizan funciones más específicas, particulares al lenguaje. En esta etapa, las clases utilizadas por el sistema se crean como objetos '*class*' y se cargan en la memoria del sistema. Antes de cargarse en memoria, es necesario probar que nuestra clase es válida, es decir, que no viola ninguna de las reglas marcadas en la especificación del lenguaje[20]. Esta validación es realizada por el *'verificador de bytecode'*, también conocido como *'probador de teorema'* o *'theorem prover'*. Comprender su objetivo es sencillo, no así su funcionamiento. Por cuestiones de eficiencia, algunas de las pruebas no se realizan al momento de cargarse, sin embargo, se garantiza que se realicen antes de la ejecución. Un ejemplo de esto es la validación de los modificadores de acceso. Otra de las funciones importantes de esta etapa es la definición de los llamados 'namespaces' lo que permite proteger el área de memoria para cada una de las aplicaciones que se ejecutan en la máquina virtual. De igual forma que el compilador, el verificador de bytecode no puede garantizar el cumplimiento de todas las reglas del lenguaje, por lo que la máquina virtual es responsable de realizarlas en tiempo de ejecución. Las protecciones de seguridad realizadas por la máquina virtual tienen que ver con aquellas cuyos parámetros de ejecución no pueden considerarse en tiempo de compilación o de cargado como es la verificación de los límites de los arreglos o el 'casting' de objetos.

[20] The Java Specification Language es la referencia técnica definitiva para el lenguaje de programación JAVA

Para controlar manualmente el nivel de verificación bytecode, las opciones en línea de comando, disponibles a partir de la versión 1.2 son:

- -Xverify:remote: Ejecuta el proceso de verificación en aquellas clases cargadas a través de la red.
- -Xverify:all: Verifica todas las clases cargadas.
- -Xverify:none: No se realiza ninguna verificación.

Es importante mencionar que la plataforma Java no tiene forma de garantizar que un código externo a la misma pueda monitorear el segmento de memoria de la máquina virtual y modificar su contenido, por lo que se recomienda implementar mecanismos adicionales propios a la máquina *host* donde se instala la máquina virtual. Lo único que garantiza la máquina virtual es la protección de la máquina del usuario contra código malicioso.

4.6 Arquitectura de Seguridad de JAVA 2.

En esta sección se describe la arquitectura de Seguridad de la plataforma de desarrollo Java, para lo cual se consideran las características de la versión 1.2 de Java, también conocida como Java 2, y/o posteriores. Sus principales características son:

1. Completa separación entre la política de seguridad y sus mecanismos de implementación, lo que permite configurar la política de manera independiente al ambiente de ejecución.
2. Mecanismo de cargado de clases seguro, junto con un mecanismo de delegación de cargadores de clases.

La arquitectura de seguridad introducida en la versión 2 utiliza una política de seguridad para decidir que permisos de acceso se otorgan a un código en ejecución. Estos permisos se basan en características del código tales como quien ejecuta el código, de donde viene, si está firmado digitalmente y si es así, quien lo firmó. Cada intento de acceder a un recurso protegido requieren de una verificación de seguridad que compara los permisos otorgados con aquellos necesarios para el acceso. Si no se define una política de seguridad, la política por default es la implementada en el '*sandbox*' de las versiones 1.0 y 1.1. Esta política se basaba en el origen del código y consistía en que todo el código cuyo origen era ajeno a la máquina local, se le asignaban permisos muy restrictivos, mientras que si el código tenía origen local, tenía todos los permisos sobre la máquina.

Por otro lado, para conservar la independencia de la plataforma, la arquitectura de seguridad de Java 2 no depende de las características de seguridad proporcionadas por un sistema operativo en particular, sin embargo, respeta los mecanismos de protección proporcionados por el mismo.

El proceso completo de cómo trabaja la arquitectura de seguridad de Java 2 se resume en los puntos siguientes:

1. Se obtiene un archivo *class* y es aceptado por la máquina virtual, si y solo si, completa satisfactoriamente todo el proceso de verificación de bytecode.
2. Se determina el origen del código del archivo class. Si el código es firmado digitalmente, se verifica la validez de la firma.
3. Una vez obtenido el origen del código y comprobado su validez, éste se ocupa para determinar el conjunto de permisos estáticos que se otorgarán a la clase.
4. Cuando se carga la clase, se crea y asocia un dominio de protección que marca el origen del código y mantiene su conjunto de permisos.
5. La clase puede instanciarse en objetos y sus respectivos métodos pueden ejecutarse. Se realizan todas las verificaciones de tipos de datos requeridas en tiempo de ejecución.
6. Cuando se invoca una verificación de seguridad y uno o más métodos de esta clase están en la cadena de la llamada, el controlador de acceso examina el dominio de protección. En este punto, se consulta la política de seguridad y se consulta el conjunto de permisos que se otorgan a la clase, basados en el origen del código de la clase. En este paso, si no se ha construido el objeto *Policy*, se construye, y este se encarga de mantener una representación de la política de seguridad en tiempo de ejecución.
7. El conjunto de permisos se evalúa para saber si se han otorgado los permisos suficientes para el acceso solicitado. Si se han otorgado, la ejecución continúa, si no, se lanza una excepción.
8. Cuando se lanza una excepción y no es cachada, la máquina virtual aborta.

Existen muchos elementos de la plataforma esenciales para el funcionamiento de la arquitectura de seguridad. Los elementos principales son: el cargador de clases, la política de seguridad y el gestor de seguridad. Cada uno de estos tiene un funcionamiento específico que se explicará a continuación.

4.7 Cargador de Clases.

La arquitectura y funcionamiento del cargador de clases (*ClassLoader*) juega un papel importante en la seguridad del lenguaje. El cargador de clases es responsable de buscar un archivo de clase o *"class file"*, generalmente en el sistema de archivos, y definirlo en la máquina virtual. El nombre completo de cada clase se define por la concatenación del *nombre del cargador de clase* + *nombre del paquete* + *nombre de la clase*, de esta manera si dos aplicaciones declaran un paquete con el mismo nombre y el mismo nombre de clase, existirá una división por la *instancia* del cargador de clase que definió a cada una. Esto se entiende ya que cada cargador de clases mantiene referencias internas a cada clase que define.

El proceso de búsqueda y cargado de clases está diseñado para evitar la usurpación de clases importantes del sistema, como puede ser la creación de una clase falsa *String* o cualquier tipo de datos básico del lenguaje. Para esto, el sistema crea un árbol de delegación en tiempo de ejecución, el cual específica la ruta a seguir en el proceso de búsqueda y cargado de clases. Básicamente existen 3 tipos de cargadores de clases: el cargador *bootstrap*[21], el cargador de extensión y el cargador de clases de aplicación. El cargador *bootstrap* se encarga de todas las clases del sistema; el cargador de extensión es el encargado de las clases pertenecientes a los paquetes instalados de forma opcional, y por último, el cargador de aplicación se encarga de todas las clases que se encuentran en la ruta definida por la variable CLASSPATH.

Una de las funciones de seguridad más importantes del cargador de clases es la de asociar a cada clase con un dominio de protección, así como con un conjunto de permisos estáticos asociados al origen del código de la clase. Los dominios de protección de cada una de las clases permiten definir los permisos de ejecución que cada una de ellas tendrá. Este procedimiento se realiza al momento de definir la clase.

Como lo muestra el diagrama de la Figura 30, el proceso de cargado de clases comienza por buscar si la clase en cuestión ya ha sido cargada en memoria, si es así, no hay necesidad de cargarla nuevamente. El proceso de búsqueda de clases se delega a un cargador de clases padre, y esto se realiza recursivamente hasta alcanzar el cargador 'bootstrap', garantizando así su prioridad en la búsqueda. Si éste cargador ('bootstrap') no es el responsable de cargar la clase en cuestión, la búsqueda comienza en sentido descendente del árbol de delegación hasta llegar al cargador que inició la búsqueda.

[21] El cargador bootstrap también es conocido como cargador de clases primordial o cargador de clases 'null'.

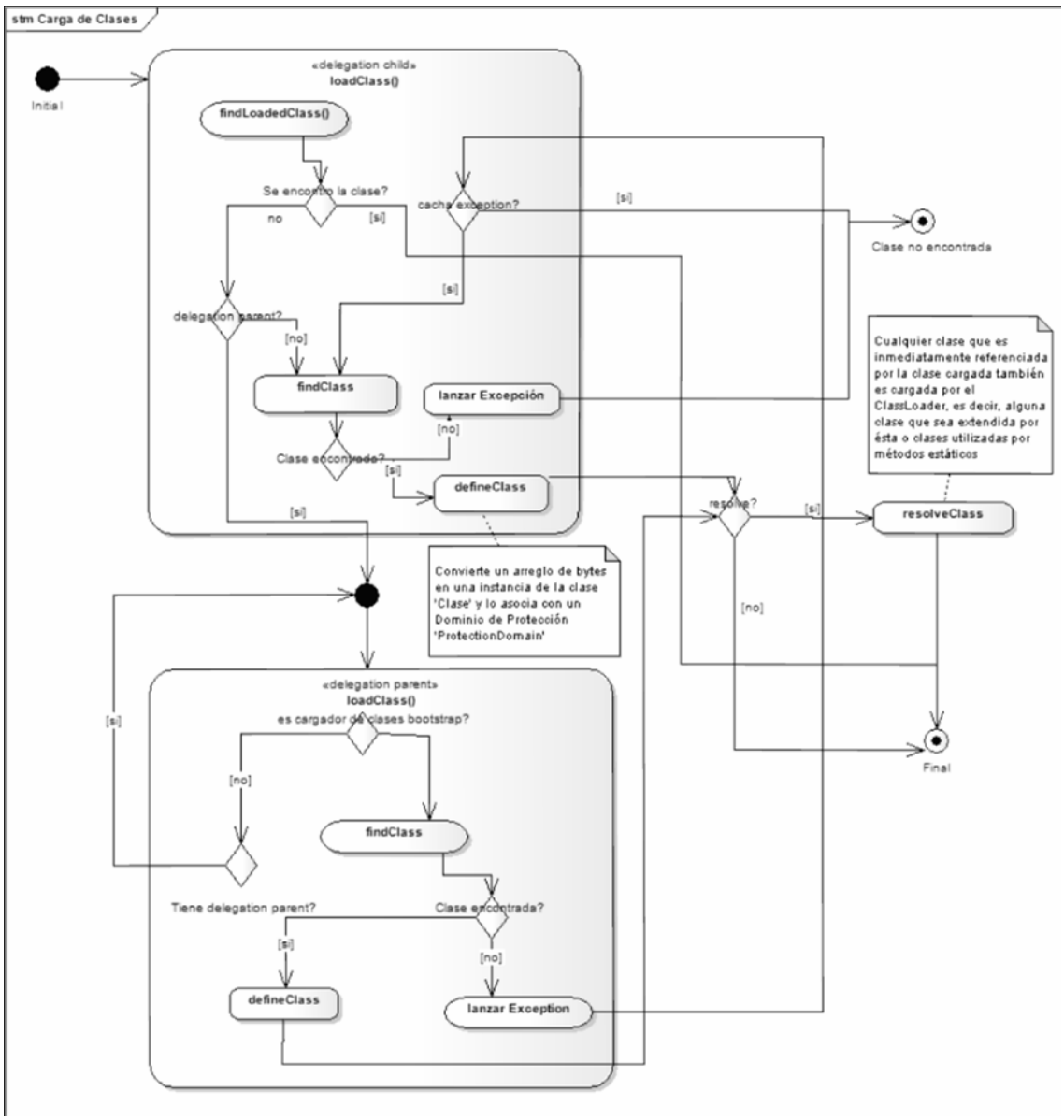


Figura 2: Algoritmo del *ClassLoader*.

El API de JAVA define la jerarquía de clases mostrada en la Figura 31.

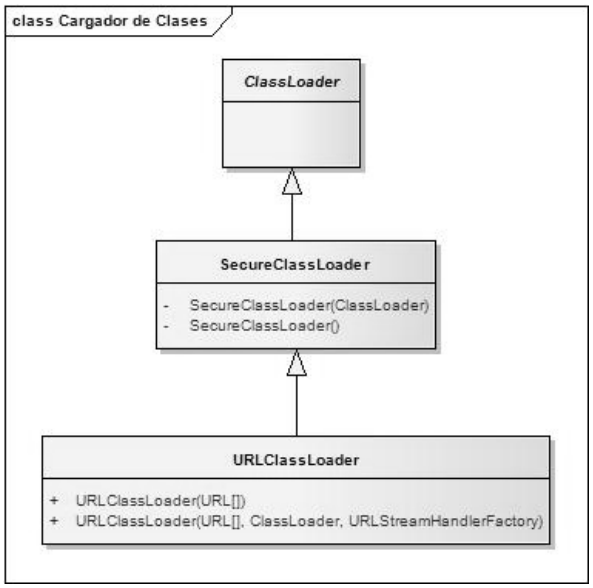


Figura 3: Diagrama jerárquico de los cargadores de clase principales.

La clase '*ClassLoader*' es una clase abstracta, por lo que no se puede instanciar pero está diseñada para proporcionar toda la funcionalidad necesaria para crear la definición de la clase. La clase '*SecureClassLoader*' a pesar de no ser una clase abstracta, no se puede instanciar debido a que todos sus constructores tienen un acceso del tipo '*protected*'. Esta clase proporciona cierta funcionalidad al cargador de clases basándose en el origen del código y ofreciendo los llamados dominios de protección. En ambas clases (*ClassLoader* y *SecureClassLoader*), no se define la manera en la que se obtienen los 'bytecodes', por lo que al momento de crear una subclase de cualquiera de las anteriores se debe pensar en la manera en que se obtendrán. La clase '*URLClassLoader*' es la única subclase que ofrece la funcionalidad de cargar las clases provenientes del sistema de archivos o desde un servidor HTTP.

Cualquier implementación de una subclase del cargador de clases se debe realizar con mucho cuidado debido a las características de seguridad que ofrece a la plataforma de seguridad del lenguaje. Debe ponerse especial atención en respetar el modelo de delegación, y generalmente se recomienda crear una subclase de '*SecureClassLoader*'.

Los motivos principales por lo que se requiere crear una nueva subclase *ClassLoader* es debido a que se requiera proporcionar un método alterno para la obtención de los archivos 'class' que no es un URL, o porque se necesite cambiar la política de seguridad.

El administrador de la máquina virtual debe otorgar los permisos necesarios para que un determinado código pueda crear un cargador de clases.

4.8 Elementos de la Política de Seguridad (Policy)

Todos los permisos de ejecución de la plataforma de seguridad de Java se basan en una política de seguridad. Esta política establece que recursos del sistema pueden accederse y por quién. Básicamente consiste en un mapeo entre características que describen un código en ejecución (origen, firma digital, usuario) y un conjunto de permisos de acceso a recursos otorgados al código.

Los elementos necesarios para comprender el funcionamiento de la política de seguridad en Java se describen por las clases *Permission*, *PermissionCollection*, *CodeSource*, *Principal*, *ProtectionDomain* y *Policy*.

4.8.1 Permisos (Permission)

Los permisos se representan mediante la clase abstracta *Permission* y se forman de la descripción de un recurso del sistema y las operaciones soportadas por él, por lo tanto, cada permiso se define por un nombre y una lista de acciones. Los permisos cumplen con dos funciones dentro de la máquina virtual:

- Se utilizan para describir los accesos permitidos por la política de seguridad instalada.
- Definen el tipo de permisos requeridos en tiempo de ejecución, antes de realizar una operación protegida.

La plataforma Java basa su política de seguridad en permisos positivos, es decir, cuando los declaramos dentro de la política de seguridad significa que estamos otorgando el permiso determinado al código que cumpla con las características especificadas. El diagrama siguiente muestra la estructura jerárquica de las clases *Permission* más representativas de la plataforma Java.

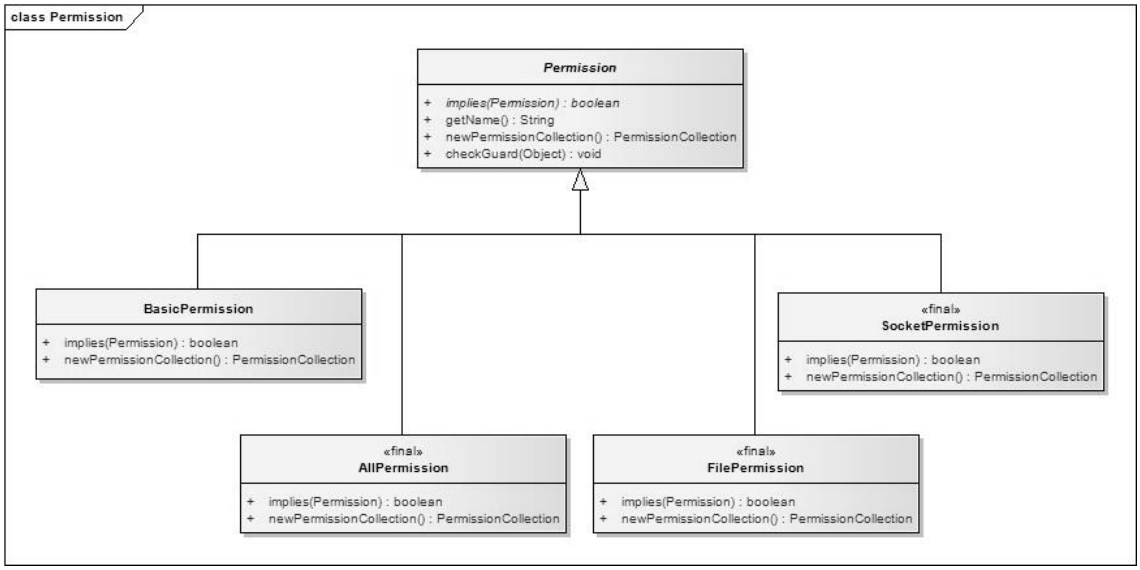


Figura 4 Jerarquía de clases *Permission*

4.8.2 Colección de Permisos (PermissionCollection)

La función más importante de los objetos *Permission* está determinada por el método *implies()*. Esta función recibe como parámetro un objeto *Permission*. En una llamada al método *implies()* de la forma *x.implies(y)*, el método se encarga de determinar si tener el permiso 'x' implica tener el permiso 'y'. Esta función es primordial para el correcto funcionamiento de la política de seguridad, sin embargo, por cuestiones de desempeño de la máquina virtual, no conviene realizar tales comparaciones uno a uno por cada permiso requerido. Como solución, la máquina virtual utiliza la clase *PermissionCollection* para almacenar un conjunto de permisos homogéneos, es decir, permisos del mismo tipo. Cada subclase de *Permission* debe tener una subclase del tipo *PermissionCollection*, utilizada para almacenar objetos de su tipo, de tal manera que una clase *FilePermission* debe utilizar una clase *FilePermissionCollection* para esta función. Cualquier subclase de *PermissionCollection* implementa un método *implies()* que cumple con la misma función del método de la clase *Permission*, sin embargo , no realiza la comparación permiso a permiso, si no que se basa en un conjunto de permisos del mismo tipo. Generalmente, la implementación del método *implies()* de la clase *PermissionCollection* no se basa en el método *implies()* de la clase *Permission*, si no que está implementada para optimizar su algoritmo.

Una de las subclases más importantes de *PermissionCollection* es la clase *Permissions*. Esta clase, a diferencia de *PermissionCollection*, almacena permisos heterogéneos. Está formada por un conjunto de clases *PermissionCollection*, una subclase de *PermissionCollection* por cada tipo.

4.8.3 Descripción de código.

Las características utilizadas para describir código son:

1. Su origen, especificado por un URL[22],
2. Sus certificados digitales, en dado caso que el código haya sido firmado digitalmente.
3. Quien ejecuta el código.

Estas características se representan por las clases *CodeSource* y *Principal*. La clase *CodeSource* es inmutable[23], lo que significa que no se puede modificar una vez creada. Esta característica es importante ya que solo así se protege la integridad de los objetos *CodeSource*. Su integridad es de vital importancia ya que las decisiones de control de acceso se basan en el origen del código ejecutado. Dos objetos *CodeSource* son iguales si sus URL son iguales y si los certificados de ambos objetos son idénticos.

La interfaz *Principal* se utiliza para representar a un usuario o servicio que se ha autenticado y está asociado al contexto de ejecución actual, de tal forma que cualquier ejecución subsecuente se realice en nombre del usuario o servicio definido.

4.8.4 Dominio de Protección (ProtectionDomain).

Al momento de que cada clase se carga en la máquina virtual, se le asigna una instancia de la clase *ProtectionDomain*, la cual encapsula las características de la clase que son el origen de la clase (*CodeSource*), los usuarios autenticados (*Principal*) y el conjunto de permisos asociados (*PermissionCollection*). Los permisos otorgados a cada *ProtectionDomain* se pueden asignar de manera estática (al momento de que se crea la instancia *ProtectionDomain*) o dinámica (al momento de que se realiza una verificación de control de acceso).

---

[22] URL (Uniform Resource Locator) es un URI (Uniform Resource Identifier ó identificador de recurso uniforme) que especifica el lugar y el protocolo donde el recurso se encuentra disponible.
El formato general de un URL es:
protocolo://maquina:puerto/directorio/archivo
También puede incluir datos como:
protocolo://usuario:contraseña@maquina:puerto/directorio/archivo

[23] Un objeto inmutable es aquel del que a pesar de obtener una referencia al mismo, sus contenidos no pueden modificarse. Las características de un objeto inmutables son: a) Todos sus campos son 'private', b) No incluye métodos 'set' para alterar los campos de la clase, c) Los métodos no pueden sobrescribirse, ya sea especificando la clase o los métodos como 'final' y d) Cada uno de los campos de la clase deben ser inmutables, si no son campos primitivos o inmutables por si mismos, realizar copias a fondo de los mismos, es decir, no utilizar el paso-por-valor para los objetos de tipo referencia.

La primera vez que se encuentra un *CodeSource* en la etapa de cargado de clases, se crea un dominio de protección, lo que significa que por cada instancia *ClassLoader*, todas las clases con el mismo *CodeSource* se mapean al mismo *ProtectionDomain*.

Como se mencionó anteriormente, la asignación de permisos a cada clase se puede realizar de forma estática o dinámica. El procedimiento de asignación estática de permisos se realiza al momento de que se carga la clase.

Una vez que la clase se encuentra en la máquina virtual, ésta tiene un dominio de protección asociado que incluye el conjunto de permisos otorgados por la política de seguridad instalada.

Por cada requisito de verificación de seguridad posterior, la máquina virtual no consultará la política de seguridad instalada sino que verificará el objeto *PermissionCollection* asociado al *ProtectionDomain*. Un diagrama de secuencia del procedimiento de asignación estática se muestra en la Figura 33.

Por su parte, la asignación dinámica de permisos a cada dominio de protección se realiza al momento que se realiza la primera verificación de seguridad, es decir, al momento que se llama al método *checkPermission()* ya sea de la clase *SecurityManager* o *AccessController*.

En este caso el cargador de clases no consultará el objeto *Policy* al momento de cargar la clase, si no que asignará un objeto *PermissionCollection* vacío. Para obtener los permisos, cuando se realiza la verificación de seguridad, la llamada al método *implies()* del *ProtectionDomain* se delegan al método *implies()* del objeto *Policy*. La Figura 6 muestra el diagrama de secuencia del proceso de asignación dinámica de permisos.

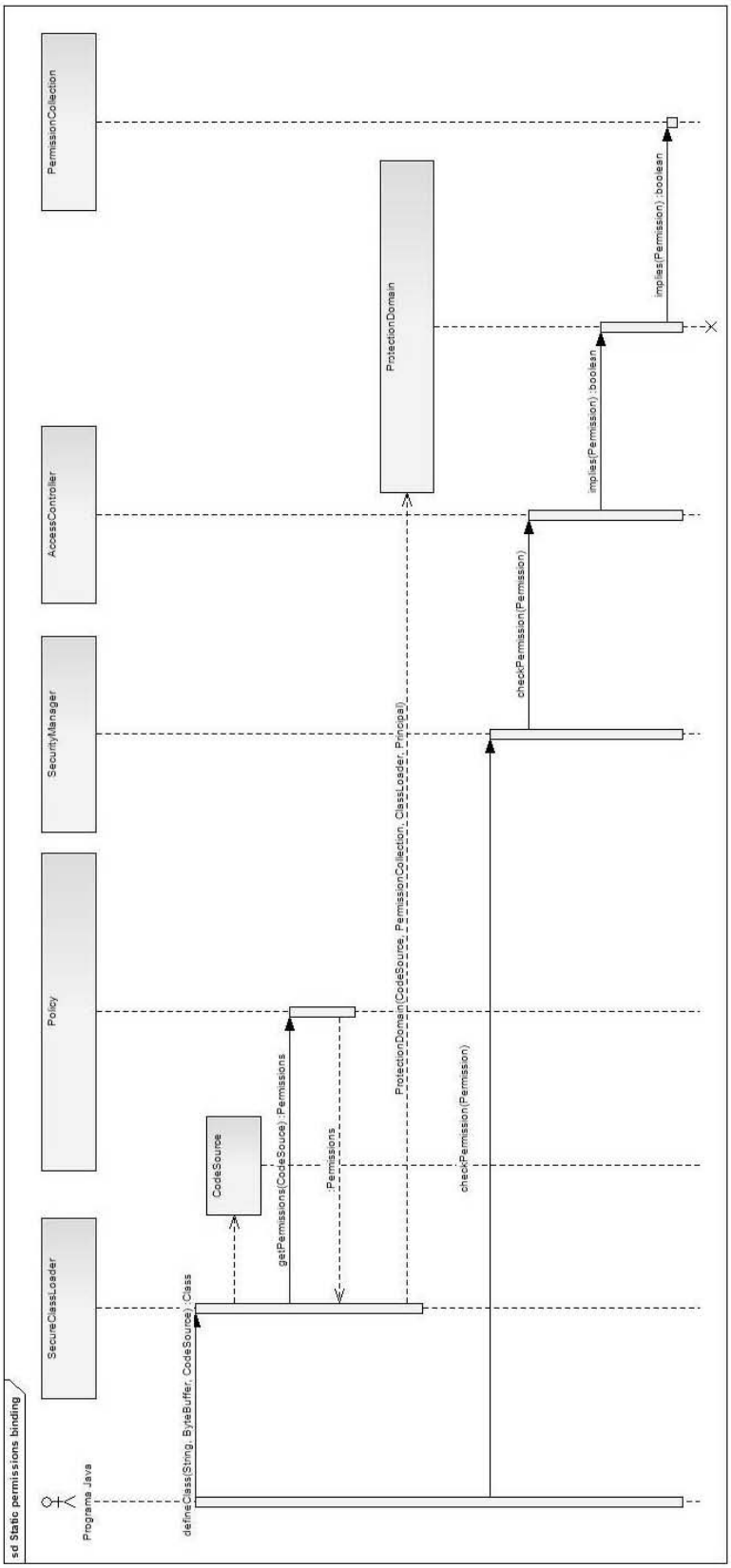


Figura 5: Asignación estática de permisos.

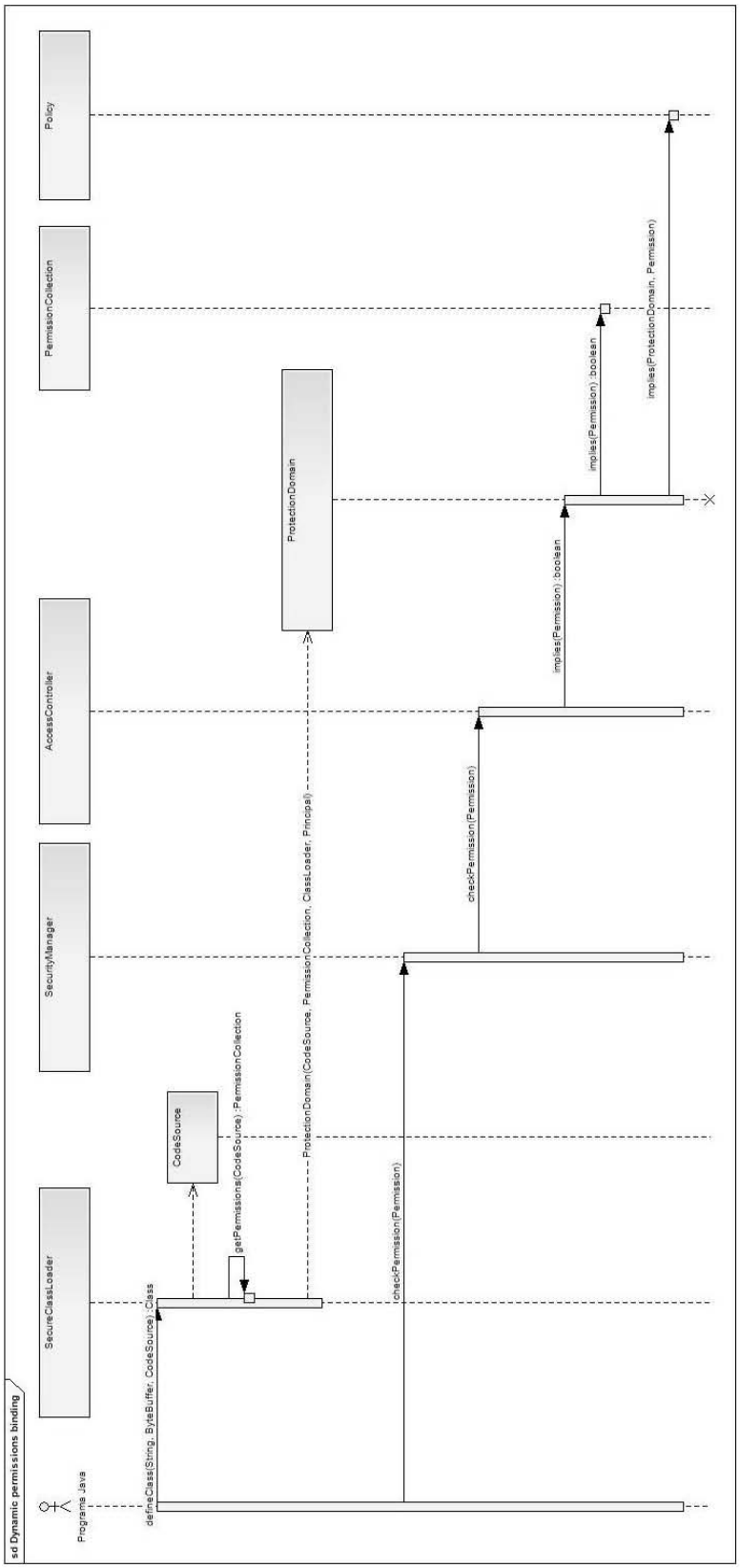


Figura 6: Asignación dinámica de permisos.

4.8.5 Policy.

En Java la expresión de la política de seguridad es declarativa por naturaleza, es decir, se expresa de forma 'no programada' y externa a la máquina virtual, esto proporciona flexibilidad al momento de programar, instalar y configurar las aplicaciones. Por default, la política de seguridad se expresa en un archivo con un formato particular.

Dentro de la máquina virtual, la política de seguridad se representa por un objeto del tipo *Policy*. A pesar de que pueden existir múltiples instancias del objeto *Policy*, solo una se consulta al momento de la toma de decisiones de accesos a recursos protegidos.

Como gran parte de las clases de la API de seguridad Java, la clase *Policy* está diseñada como una estructura proveedora, es decir, existe una implementación por default, sin embargo, es posible instalar una implementación alternativa.

Es importante mencionar que si se instala un administrador de seguridad (*SecurityManager*) pero no se define una política de seguridad, la máquina virtual creará una que funcionará como las versiones iniciales de Java (default sandbox).

En la implementación por default, la política de seguridad incluye dos archivos de políticas, uno definido de manera global para todas las instancias de la máquina virtual (*<java.home>/lib/security/java.policy*) y otro que se encuentra en el directorio '*home*' de cada usuario (*<user.home>/.java.policy*). Para agregar, eliminar o configurar la ubicación de los archivos que conforman la política de seguridad se requiere editar el archivo de configuración *java.security* en los parámetros *policy.url.n*, donde '***n***' es un número a partir del 1, que indica la prioridad en su lectura.

La implementación default de la clase *Policy* (*sun.security.provider.PolicyFile*) se conforma por cero o una entrada '*keystore'* y cero o más entradas *'grant'*. La entrada 'keystore' especifica el nombre y ubicación de un archivo que contiene la base de datos de las llaves privadas y sus certificados digitales asociados. Este archivo se utiliza para buscar las llaves públicas de las entidades firmantes especificadas en las entradas 'grant' del mismo archivo de política. Las entradas 'grant' indican los permisos otorgados a la aplicación, así como las circunstancias bajo las cuales se otorgan. El formato general de esta entrada es el siguiente:

```
grant signedBy "signer-names", codeBase "URL",
        principal principal-class-name "principal-name",
        principal principal-class-name "principal-name",
        ...{
        permission permission-class-name "target-name","action",
                signedBy "signer-names";
        permission permission-class-name "target-name","action",
                signedBy "signer-names";
};
```

Por último, es importante mencionar que estos archivos de configuración de política se pueden modificar manualmente con cualquier editor de texto, sin embargo, se recomienda utilizar la herramienta '*policytool*' para realizarlo.

4.8.6 Personalización de la Política de Seguridad.

Es muy probable que en las aplicaciones Java se requiera personalizar el funcionamiento de la política de seguridad. Uno de los elementos importantes a modificar es la creación de nuevos tipos de permisos propios de la aplicación en desarrollo.

Cuando se desea personalizar o crear clases del tipo *Permission* es importante considerar los siguientes puntos:

1. Declarar los nuevos tipos *Permission* con modificador *final*, de tal manera que no sea posible modificarlos o extenderlos.
2. Una implementación de un tipo *Permission* requiere la creación de una clase *PermissionCollection* para el tipo de datos correspondiente. Es importante poner especial atención en la implementación de la semántica de su método *implies()*.
3. Si se requiere crear una jerarquía especial de permisos basados en una clase base, que generalmente es *abstract*, y se tiene una implementación propia del método *implies()*, éste método debe considerar de manera adecuada el tipo exacto de la clase. El fragmento de código recomendado es:

```
public boolean implies(Permission permission){
        if(!(permission instanceof BasicPermission))
                return false;
        BasicPermission bp=(BasicPermission) permission;
        if(bp.getClass() != this.getClass())
                return false;
        …
}
```

Por su parte, la política de seguridad se procesa por el objeto *Policy* y se hace cumplir por las clases *SecurityManager* y/o *AccessController*, por lo tanto, la personalización de cualquiera de estas clases, cambiará el comportamiento de la política de seguridad. Es por esto que es muy importante poner especial atención en la posible instalación de clases de éste tipo y comprender su funcionamiento completo.

Generalmente una reimplementación de la clase *Policy* se debe a la necesidad de implementar un formato de almacenamiento especial o un lenguaje de expresión de la política distinto.

4.9 SecurityManager / AccessController

Actualmente y por cuestiones históricas se mantienen dos objetos encargados de la verificación de la política de seguridad en la plataforma Java. Estos objetos se representan por las clases *SecurityManager* y *AccessController*. El controlador de acceso (*AccessController*) existe a partir de la liberación de la versión Java 1.2, también conocida como Java 2, mientras que en versiones anteriores todos los cambios a la política de seguridad requerían cambios en el gestor de seguridad (*SecurityManager*). Actualmente, todas las peticiones realizadas al gestor de seguridad son delegadas al controlador de acceso, pero por razones de compatibilidad no se elimina la interfaz proporcionada por el gestor, por lo tanto, la estructura de seguridad de la plataforma contempla ambas interfaces de la siguiente manera:

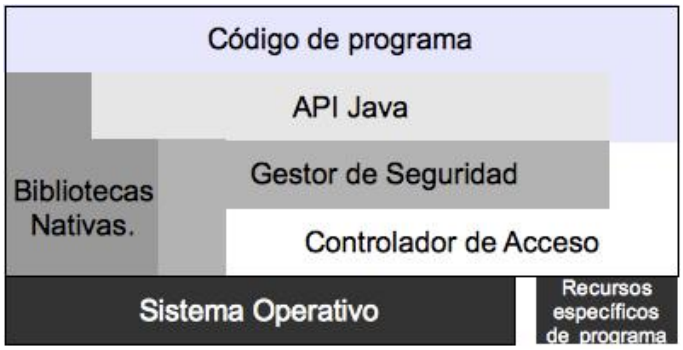


Figura 7: Coordinación entre el gestor de seguridad (*SecurityManager*) y el controlador de acceso (*AccessController*).

El gestor de seguridad no está presente de manera predeterminada en todas las aplicaciones desarrolladas en Java. Generalmente para las aplicaciones de escritorio se debe declarar explícitamente con el parámetro en línea de comando –Djava.security.manager, por lo tanto, es importante aclarar que todas las aplicaciones desarrolladas en Java no tienen un gestor de seguridad por default, mientras que los applets tienen un gestor de seguridad muy restrictivo que sigue el modelo de seguridad de las primeras versiones del lenguaje.

El algoritmo general del gestor es el siguiente:

1. El programador, mediante código, realiza una petición a la API de Java para que realice una operación.
2. El API de Java verifica a través del gestor de seguridad si esa operación es permitida.
3. Si la operación no está permitida, el gestor de seguridad envía una *Exception* que se propaga hasta el usuario.
4. Si la operación es válida, simplemente se realiza.

Es importante resaltar que por diseño, el API de JAVA no intenta conservar un estado después de verificar ciertos permisos, por lo que siempre consultará a la política de seguridad instalada antes de realizar cualquier operación de seguridad.

La mayoría de las implementaciones del gestor de seguridad se basan en el controlador de acceso. El controlador de acceso decide a quién se le otorgan ciertos permisos de ejecución basándose en información del código como es su origen, sus firmas digitales, así como el usuario/servicio que ejecuta el código. El controlador de acceso obtiene esta información de las clases *CodeSource*, *Principal*, *Permission*, *ProtectionDomain* y *Policy*.

En Java 2 cada clase pertenece a un dominio. La máquina virtual mantiene el mapeo de cada clase del código con su dominio de protección. Este mapeo se establece solo una vez, al momento de que la clase es definida por el cargador de clases, y no se puede cambiar a lo largo del tiempo de vida del objeto *Class*, por lo tanto, la clase '*ProtectionDomain*' encapsula los objetos *CodeSource*, *Principal*, *ClassLoader* y *PermissionCollection.*

Que el controlador de acceso otorgue o rechace cierto permiso depende del conjunto de dominios de protección que se encuentran en el stack de ejecución al momento de llamar al controlador de acceso. Cuando se llama al método *checkPermission()* , este se encarga de consultar los permisos asociados al dominio de protección de cada método que se encuentra en el stack. La clase *AccessController* engloba todo el algoritmo de seguridad de la plataforma Java, y es en ésta clase donde se basa el gestor de seguridad, por lo que es esencial comprender claramente su funcionamiento.

La clase *AccessController* es una clase del paquete 'java.security' declarada 'final', es decir no se puede reimplementar. Esta clase se encarga de todas las decisiones y operaciones relacionadas con el control de acceso. Sus principales funciones son las siguientes:

- Decidir si el acceso a un recurso protegido se otorga o niega basándose en la política de seguridad instalada.
- Marcar código como 'privilegiado', afectando las determinaciones de acceso siguientes.
- Obtener un 'snapshot' del contexto actual de tal forma que sea posible tomar decisiones del contexto guardado desde un contexto distinto.

La Figura 36 muestra los métodos definidos en esta clase.

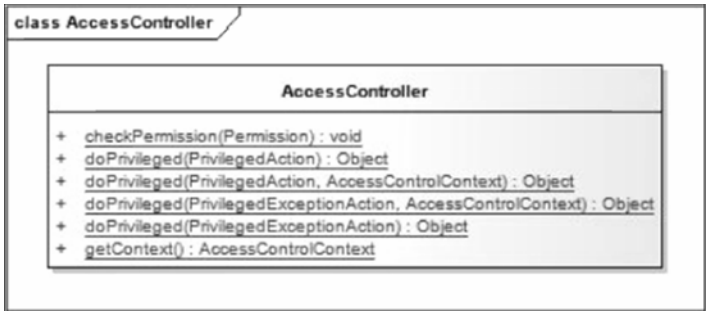


Figura 8: Métodos principales de la clase *AccessController*.

El algoritmo del funcionamiento del *AccessController* se puede describir mediante el siguiente pseudocódigo:

```
1: i=m;
2: while(i>0){
3:      if(el dominio del 'caller' i no tiene permiso)
4:              Throw AccessControlException;
5:      else is(el 'caller' es marcado como 'privileged'){
6:              if(se especificó un contexto  en la llamada a doPrivileged)
7:                      context.checkPermission(permission);
8:              return;
9:      }
10:     i=i-1;
11:}
12:
13:// Después, verifica el contexto heredado cuando el 'hilo de ejecución' se creó.
14:// Siempre que se crea un nuevo 'hilo de ejecución' se almacena el
15:// AccessControlContext y se asocia con el nuevo 'hilo de ejecución'
16:// como un contexto "inherited"
17:
18:inheritedContext.checkPermission(permission);
19:return;
```

Figura 9: Algoritmo general del controlador de acceso (*AccessController*).

A continuación damos una breve explicación del algoritmo. Comenzamos por definir el término 'caller', que se refiere a cualquier dominio de protección dentro de un contexto de ejecución, es decir, corresponde a cada uno de los dominios de protección de las clases de los métodos que se encuentran en el stack de ejecución. Por cada uno de los callers, se determina si la política de seguridad instalada permite la ejecución para tal dominio de protección. Si la política de seguridad determina que no es posible otorgar determinado permiso al dominio de protección se lanza una excepción del tipo *AccessControlException.* El primer dominio de protección a verificar es el correspondiente al método de la clase que se llamó recientemente, y así sucesivamente hasta alcanzar la llamada más antigua del stack, de esta manera, para que se ejecute una acción en el contexto de seguridad de Java, cada una de los dominios de protección asociados a los métodos de las clases que se encuentran en el stack, deben tener el permiso de ejecutar tal acción, de acuerdo a lo indicado en la política de seguridad instalada.

En muchas ocasiones, una aplicación requiere acceder momentáneamente a un recurso protegido. El ejemplo más común es cuando un usuario requiere hacer un cambio de su contraseña de inicio de sesión. En este caso, el usuario necesita disponer de un permiso momentáneo de acceso al archivo de contraseñas, que generalmente se encuentra protegido. El algoritmo del controlador de acceso incluye la posibilidad de ejecutar código de manera privilegiada, diseñado para proporcionar una solución para estos casos especiales.

La forma de implementarse es a través de un método estático de la clase *AccessController* llamado *doPrivileged*. Este método toma como parámetro un objeto del tipo *PrivilegedAction* ó *PrivilegedExceptionAction,* las cuales son interfaces del paquete 'java.security' que proporcionan un método por implementar conocido como *run()* con un tipo de retorno del tipo *Object*.

Todo el código que se encuentre dentro del método *run()* se ejecutará de manera privilegiada. Es muy importante aclarar que ésta forma de ejecutar código "privilegiado" no significa que el código escala privilegios que no tiene, sino que simplemente evita que el controlador de acceso siga su recorrido por todos los métodos del stack y verifique el permiso en cada dominio de protección restante. Esta funcionalidad queda claramente mostrada a través de la llamada 'return' en la línea 8 del algoritmo mostrado en la Figura 37.

Como se puede ver en el algoritmo (líneas 5 a 9), es posible someter a un contexto de seguridad adicional cada una de las llamadas a '*doPrivileged*'. Que un código sea privilegiado, no significa que deba tener todos los privilegios.

Esta funcionalidad se realiza a través de los métodos *'doPrivileged'* que toman como parámetro un objeto del tipo *AccessControlContext*. Por otro lado y tomando en cuenta el amplio uso de los hilos de ejecución en la plataforma Java, es importante comprender el funcionamiento de este algoritmo al momento de crear hilos de ejecución.

Cada vez que se crea un nuevo hilo de ejecución, éste hereda el contexto de ejecución de todos sus antecesores, es decir, al momento de crear un nuevo hilo de ejecución, el contexto actual se congela y se hereda por el hilo de ejecución, de tal manera que el hilo padre y el hilo hijo puedan continuar con su propia ejecución basados en un copia de contexto para ambos.

Por lo tanto, cada una de las verificaciones de seguridad realizadas por el controlador de acceso deben incluir una verificación del permiso específico para todos los dominios de protección que se encuentran en el contexto de seguridad heredado. Toda esta funcionalidad se muestra claramente en la línea 18 del algoritmo de la figura anterior.

Como se mencionó anteriormente, las clases *AccessController* y *SecurityManager* existen por cuestiones históricas y de compatibilidad, sin embargo, las recomendaciones generales para usar una u otra clase son las siguientes:

1. El gestor de seguridad no se encuentra de manera predeterminada en las aplicaciones Java, requiere de un parámetro en la llamada a la máquina virtual. Mientras que el controlador de acceso se encuentra de forma predeterminada, por lo tanto, si necesitamos asegurarnos que una verificación de seguridad se realice, necesitamos llamar a los métodos del controlador de acceso.

2. Una llamada al gestor de seguridad no garantiza el cumplimiento del algoritmo de control de acceso descrito anteriormente, ya que esta clase (*SecurityManager*) se puede extender e instalar de forma predeterminada y con una funcionalidad distinta, mientras que una llamada a los métodos del controlador de acceso garantiza el algoritmo descrito anteriormente.

La máquina virtual Java utiliza dos técnicas para ligar el conjunto de permisos a los dominios de protección que son mediante un '*binding'* estático ó dinámico.

Se dice que la política de seguridad se carga de manera estática, cuando los permisos que la describen se consultan al momento de cargar las clases, mientras que en el método dinámico, la lectura de la política de seguridad se realiza en el instante en que se desea realizar la primera consulta de acceso a un recurso protegido.

La política de seguridad está diseñada como una estructura proveedora, es decir, existe una implementación por default, pero es posible instalar una implementación alterna, de tal manera que el contenido de la misma puede obtenerse por diferentes medios, ya sea, mediante un archivo de texto, un servicio de directorio, etc. o en distintos formatos.

4.10 Criptografía (JCA/JCE[24]).

La arquitectura criptográfica de Java (JCA, por sus siglas en inglés) es un framework que permite el acceso y desarrollo de funciones criptográficas para la plataforma Java. Su arquitectura permite proporcionar independencia de algoritmos gracias a que define distintos tipos de motores (*engines*) criptográficos que definen los servicios proporcionados por el framework. Algunas de las clases que representan estos servicios son *MessageDigest*, *Signature*, *KeyFactory* y *Cipher*.

La independencia en la implementación se logra a través de su arquitectura basada en proveedores o CSP (*Cryptographic Service Provider*), es decir, puede hacer uso de un conjunto de clases, desarrollada por un tercero, que implementa uno o más servicios criptográficos a través de uno o más algoritmos. En la Figura 38 se pueden observar la arquitectura de JCA.

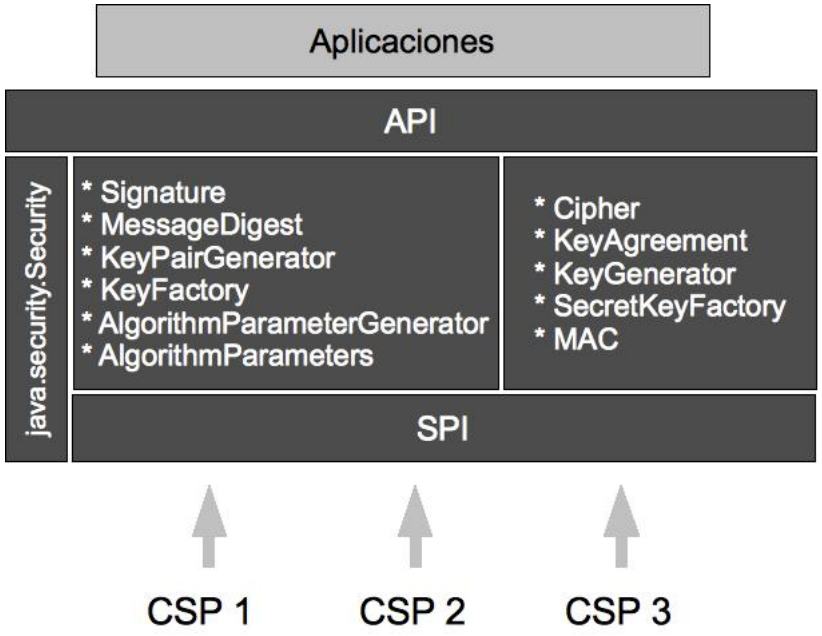


Figura 10: Arquitectura JCA

Desde sus inicios, el framework ha estado compuesto por dos paquetes conocidos como JCA y JCE, estando el primero totalmente integrado a la API principal de JAVA y el segundo proporcionado como un paquete opcional, regida principalmente bajo los controles de exportación

[24] JCA/JCE Java Cryptography Arquitecture/ Java Cryptography Extension

de criptografía de Estados Unidos[25]. En la actualidad ambos paquetes se proporcionan en la instalación estándar de la máquina virtual y se conocen como SUN y SUNJCE, aunque pueden existir proveedores adicionales instalados por default que dependen de la distribución de la máquina virtual.

Los servicios proporcionados por la arquitectura criptográfica de Java (JCA) son:

- Creación de llaves.
- Administración de llaves y creación de 'keystore'.
- Gestión de parámetros de algoritmos.
- Generación de parámetros de algoritmos.
- Creación de certificados.
- Generación de números aleatorios.

Por su parte, los servicios proporcionados por la extensión criptográfica de Java (JCE) son:

- Cifrado.
- Generación de llaves.
- Acuerdo de llaves.
- Código de autenticación de mensajes (MAC).
- Constructores de cadenas de certificación.
- Validadores de cadenas de certificación.

La arquitectura de JCA/JCE permite agregar diferentes proveedores de servicios criptográficos, es decir, agregar diferentes implementaciones de diversos motores criptográficos, ya sea de manera dinámica o estática. Cuando un programa solicita un algoritmo en particular, la máquina virtual se encarga de consultar a cada uno de los proveedores instalados en un orden específico, y en el momento que encuentra el servicio y algoritmo solicitado, éste se utiliza para proporcionar el servicio de seguridad correspondiente. Generalmente la especificación de los proveedores criptográficos se realiza en el archivo *java.security* que generalmente se encuentra en *$JREHOME/lib/security*. Las clases que están listadas de esta manera se deben instalar en el *classpath* del sistema. La infraestructura de seguridad utilizará únicamente el cargador de clases del sistema para localizar estas clases, por lo que generalmente se instalan en la ruta *$JREHOME/lib/ext*.

[25] http://www.bis.doc.gov/encryption/

Cada uno de los servicios criptográficos de JCA/JCE se define por una clase '*engine*' Una clase *'engine'* define un servicio criptográfico de manera abstracta, es decir, sin ninguna implementación concreta. Un servicio criptográfico siempre es asociado con un algoritmo en particular y puede realizar una de las siguientes funciones:

- Proporciona operaciones criptográficas
- Genera o proporciona material criptográfico, como puede ser llaves o parámetros, requeridos por las diferentes operaciones criptográficas.
- Genera objetos de datos, como son certificados o keystores, que encapsulan llaves criptográficas.

Las clases 'engine' definidas a partir de la versión 1.4 se definen en la siguiente tabla:

Tabla 2: Clases engine

CLASE 'ENGINE'	USO
MessageDigest	Calcula el hash del dato especificado.
Signature	Firma de datos y verificación de firmas digitales.
AlgorithmParameters	Administración de parámetros para un algoritmo en particular, incluyendo parámetros de codificación y decodificación.
AlgorithmParameterGenerator	Genera un conjunto de parámetros adecuado para un algoritmo específico.
KeyPairGenerator	Genera un par de llaves pública y privada apropiados para un algoritmo específico.
KeyFactory	Convierte claves opacas[26] del tipo Key en especificaciones de clave[27] y viceversa.
CertificateFactory	Crear certificados digitales de llave pública así como listas de revocación de certificados.
KeyStore	Crea y administra un 'keystore', el cual es una base de datos de llaves y certificados.
SecureRandom	Genera números aleatorios o pseudo-aleatorios.
Cipher	Cifra o descifra datos.
KeyGenerator	Genera llaves secretas para los algoritmos de cifrado simétrico.
SecretKeyFactory	Convierte llaves opacas del tipo *javax.crypto.SecretKey* en especificaciones de llaves, y viceversa.
KeyAgreement	Proporciona funciones para el protocolo de acuerdo de llaves.
Mac	Proporciona funciones de código de autenticación de mensajes (MAC).
CertPathValidator	Valida rutas de certificación.
CertPathBuilder	Construye una ruta o cadena de certificación.
CertStore	Recupera certificados y listas de revocación de certificados de un repositorio.

[26] Llave opaca: Representación de una llave en la cual no es posible acceder a todo el material que constituye una llave. Solo se puede acceder al algoritmo, formato y codificación.

[27] También se le conoce como llave transparente, ya que permite acceder a todo el material que forma a la llave a través de los métodos get respectivos.

A continuación se proporciona descripción importante de cada una de las clases mencionadas en la tabla anterior.

4.10.1 Principales clases criptográficas.

Una de las clases más importantes es la clase *java.security.Security*, la cual se encarga de administrar los proveedores de seguridad instalados así como todas las propiedades de seguridad.

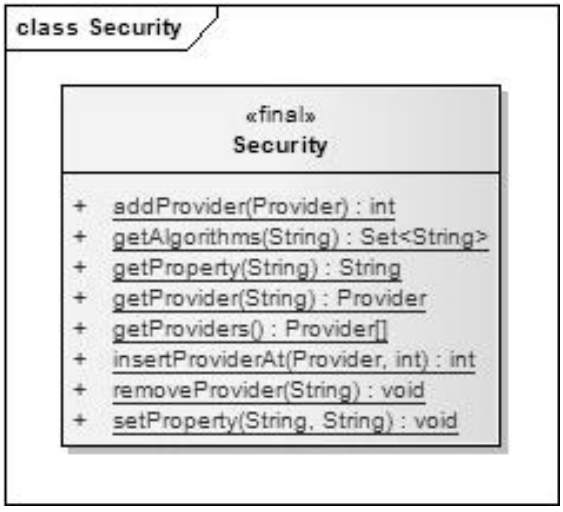


Figura 11: Clase Security

Un proveedor de servicio criptográfico se forma por un conjunto de paquetes que proporcionan una implementación concreta de un subconjunto de aspectos criptográficos de la API de seguridad de Java y se representa mediante la clase *java.security.Provider*.

Cada clase *Provider* tiene un nombre, número de versión y una cadena de descripción de proveedores y servicios. No se ocupa únicamente para registrar implementaciones de servicios criptográficos, sino que también se puede ocupar para registrar implementaciones de otros servicios de seguridad.

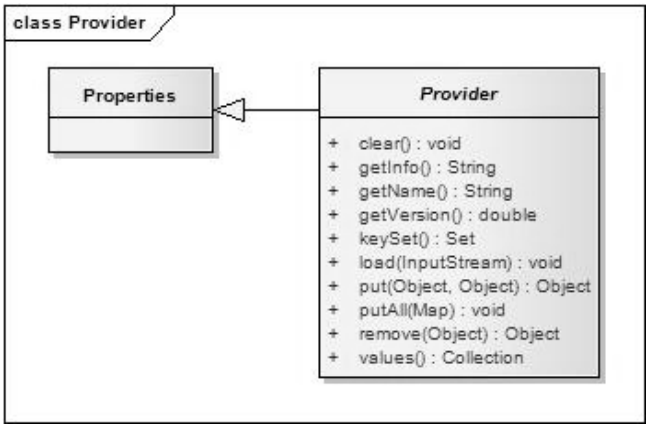


Figura 12: Clase Provider.

La clase *java.security.MessageDigest* es una clase '*engine*' diseñada para proporcionar servicios de 'hash' criptográficos, como son los algoritmos SHA-1 o MD5.

Para calcular un 'hash' utilizando la API de Java, primero creamos una instancia del objeto *MessageDigest* llamando su método estático *getInstance()*, pasando como parámetro el nombre del algoritmo a utilizar. La llamada a éste método regresará un objeto *MessageDigest* inicializado.

Una vez inicializado, antes de calcular el 'hash' de un dato, utilizamos algunos de los métodos *update()* para proporcionar los datos al objeto. El hash de los datos se calcula al realizar una llamada al método *digest*(). Algunos ejemplos de las implementaciones disponibles para algoritmos hash son MD2, MD5, SHA-1, SHA-256, SHA-384 y SHA-512.

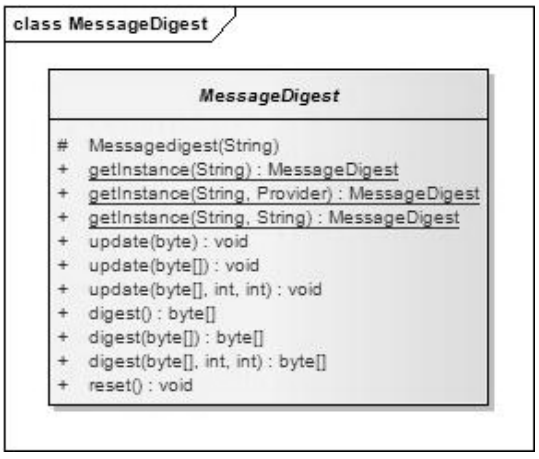


Figura 13: Clase MessageDigest.

La clase '*engine' java.security.Signature* está diseñada para proporcionar servicios de firma digital, como son *SHA1withDSA* o *MD5withRSA*. Un objeto *Signature* puede utilizarse para generar la firma de un dato o para verificarla. Esto se implementa mediante 3 estados representados como constantes dentro de la misma Clase, los cuales son:

1. UNITILIALIZED
2. SIGN
3. VERIFY

Antes de poder firmar o verificar una firma, se crea una instancia *Signature* mediante el método estático *getInstance()*, pasando como parámetro el nombre del algoritmo deseado. Cuando una instancia *Signature* es recién creado se encuentra en el estado UNITILIALIZED. La inicialización del mismo depende de si lo vamos a utilizar para firmar o verificar una firma. Si se va a ocupar para firmar datos, el objeto debe inicializarse con la llave privada de la entidad que generará la firma, utilizando el método *initSign()*. La llamada al método *initSign* pondrá al objeto en el estado SIGN. Por otro lado, si el objeto *Signature* se utilizará para la verificación de una firma, debe inicializarse con la llave pública de la entidad cuya firma se va a verificar. En este caso, se utilizará uno de los métodos *initVerify()*. La llamada a este método pone al objeto *Signature* en el estado VERIFY.

Una vez que el objeto *Signature* se encuentra en el estado SIGN, alimentamos el mismo con los datos que se van a firmar. Esto se realiza mediante una o más llamadas a los métodos UPDATE. Cuando ya hemos proporcionado todos los datos a firmar, la firma digital se generará mediante una llamada al método *sign*. Este método generará la firma y restablecerá el objeto a su estado SIGN con la misma llave privada previamente proporcionada.

El procedimiento de verificación es semejante al de firma. Comenzamos por alimentar el objeto con el conjunto de datos cuya firma se va a verificar mediante una o más llamadas *update*. El método para realizar la verificación es *verify*, y una vez completado, restablecerá el objeto *Signature* a su estado VERIFY con la misma llave pública.

Algunos nombre de algoritmos disponibles al momento de crear una instancia del objeto Signature son: *SHA1withDSA*, *MD2withRSA*, *MD5withRSA*, *SHA1withRSA* y en general siguen el estándar de nombres <digest>with<encription>.

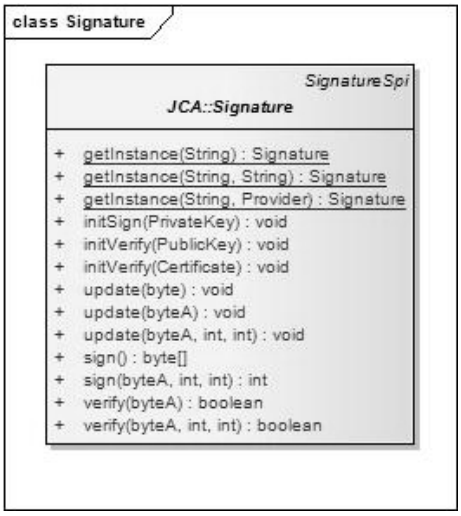


Figura 14: Clase Signature

Los algoritmos manejados por JCA son muy diferentes entre sí. Cada uno tiene requerimientos diferentes, parámetros distintos, tamaños distintos y constantes específicas. Para organizar los parámetros de cada algoritmo se crea lo que se denomina como "especificación de parámetros de algoritmo", representada por la interfaz *AlgorithmParameterSpec*.

Esta especificación es una representación "transparente" del conjunto de parámetros utilizada por un algoritmo determinado. Esto significa que es posible acceder a los valores de cada parámetro de forma individual a través de uno de los métodos *get* definidos en la propia especificación. Por ejemplo, la clase *DSAParameterSpec* define los métodos *getP*, *getQ* y *getG*, para acceder a los valores de las variables *p*, *q* y *g*, propias del algoritmo. Por otro lado, en una representación "opaca", como es el caso de la clase *AlgorithmParameters*, no tenemos acceso directo a los valores de los parámetros. Solo se incluyen métodos para obtener el nombre del algoritmo y la codificación de los parámetros. Para convertir una representación opaca en su correspondiente representación transparente, utilizamos el método *AlgorithmParameters.getParameterSpec*.

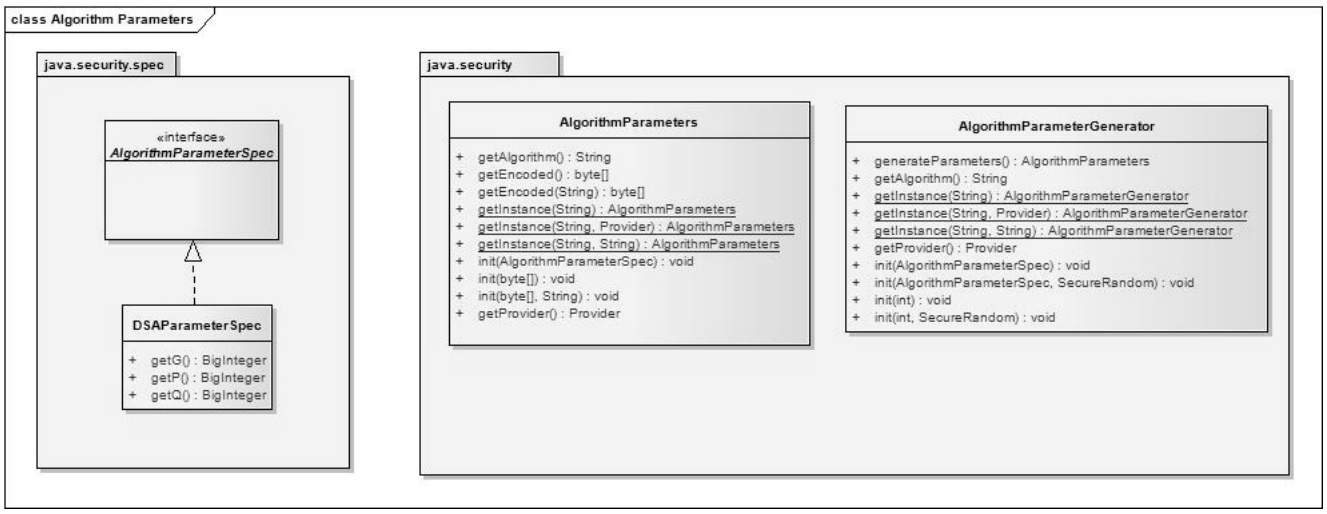


Figura 15: Clases e interfaces para la representación de parámetros de algoritmo

La interfaz de más alto nivel de todas las claves opacas es *java.security.Key.* Esta interfaz define la funcionabilidad compartida por todas las representaciones "opacas" de claves. Una representación opaca proporciona un acceso limitado a la clave, ya que solo define los métodos *getAlgorithm()*, *getFomat()*, *getEncoded()*. En una representación "*transparente*", es posible acceder a cada valor de manera individual a través de alguno de los métodos *get*. Para utilizarla representación transparente utilizamos la interfaz KeySpec.

Las interfaces java.security.PublicKey y java.security.PrivateKey extienden la interfaz Key y no incluyen métodos, por lo que solo se ocupan para identificar los tipos de datos.

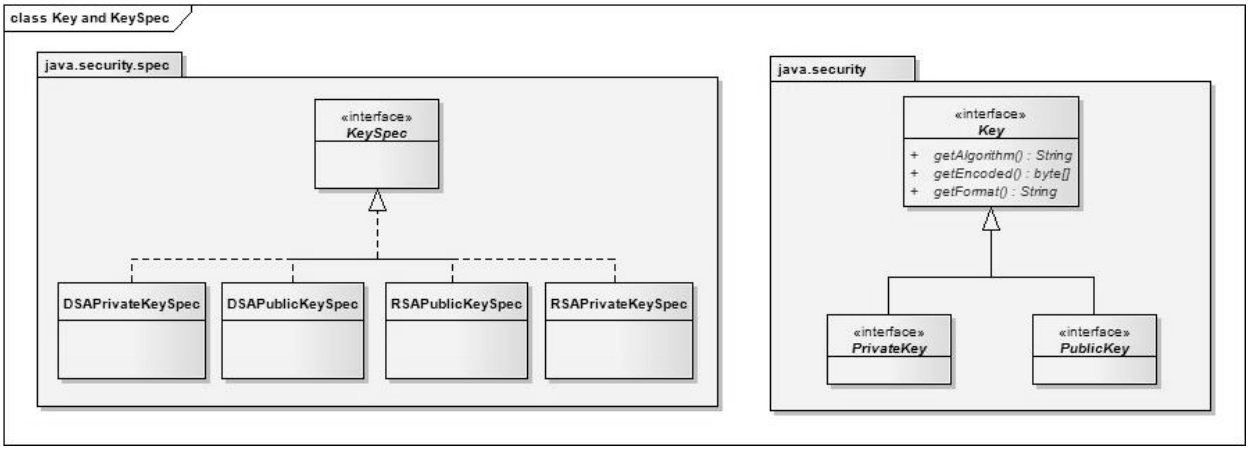


Figura 16: Representación gráfica de las relaciones entre las interfaces *Key* y *KeySpec*, con sus implementaciones más representativas.

Para poder convertir llaves “opacas” en “transparentes” y viceversa utilizamos la clase ‘*engine’ java.security.KeyFactory*, aunque también se puede ocupar para traducir entre especificaciones de llaves compatibles.

Por su parte, la clase *java.security.cert.CertificateFactory* es una clase ‘*engine’* que define la funcionabilidad de una fábrica de certificados, utilizada para generar certificados y listas de revocación de certificados a partir de sus codificaciones. En este caso, el tipo de certificado más utilizado es el conocido como X.509.

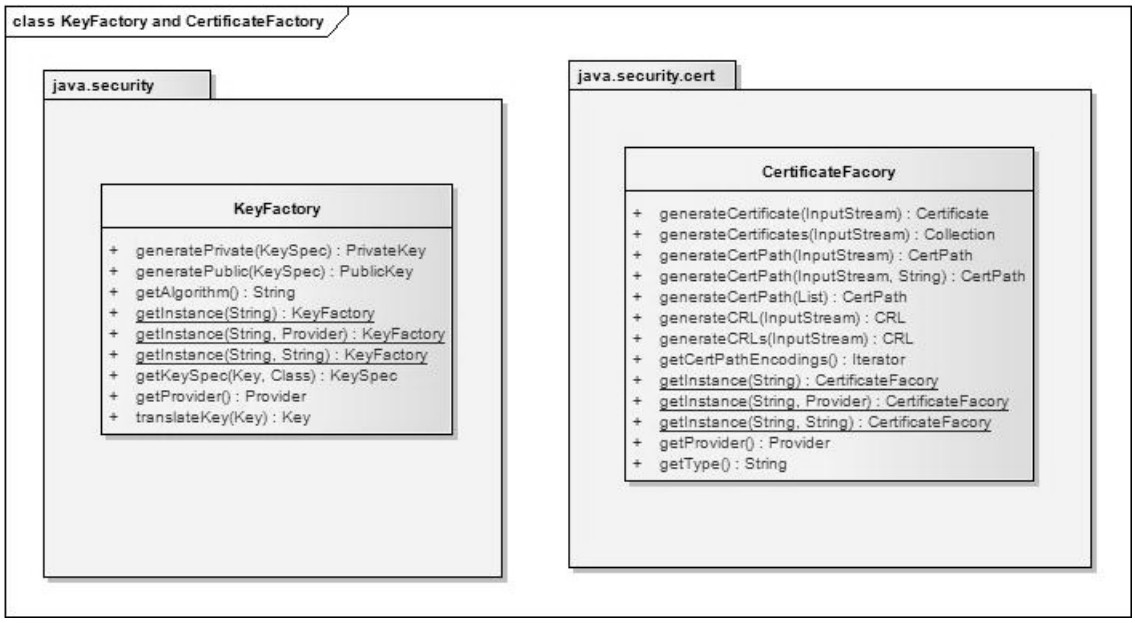


Figura 17: Clases *KeyFactory* y *CertificateFactory*.

La clase *java.security.KeyPair* es un contenedor de un par de llaves: una llave pública y una llave privada. Por su parte, la clase *java.security.KeyPairGenerator* es una clase ‘engine’ utilizada para generar pares de llaves públicas y privadas. La forma en como se generen este par de llaves dependerá de cómo se inicialice el objeto. En la mayoría de los casos, inicializar el objeto con independencia del algoritmo es suficiente.

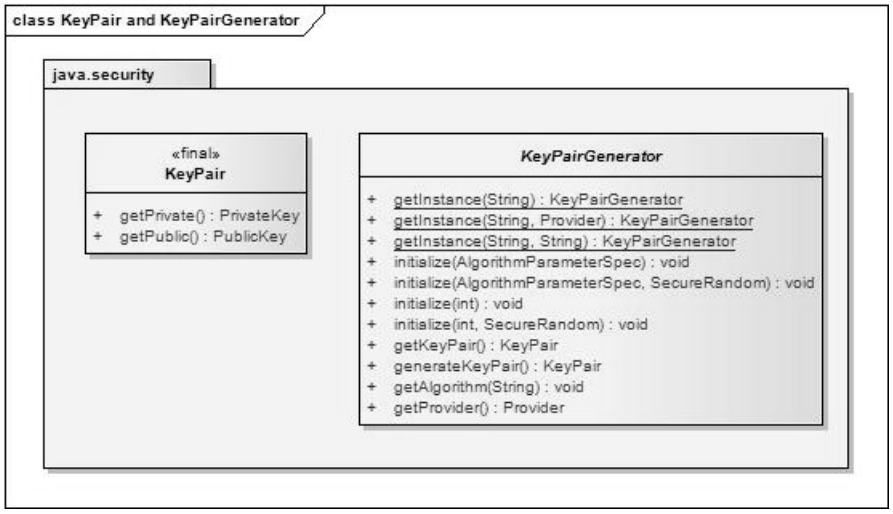


Figura 18: *KeyPair* y *KeyPairGenerator*

La clase java.security.KeyStore es una clase 'engine' que define las interfaces para accesar y modificar la información en un 'keystore'. Existen dos herramientas en línea de comandos que hacen uso de un KeyStore y son: keytool y jarsigner, así como una herramienta con interfaz gráfica conocida como Policy Tool.

La implementación de un keyStore está basada por un proveedor de seguridad. En este caso, existe una implementación por default proporcionada por el proveedor SUN que implementa el 'keystore' mediante un archivo con formato JKS.

Por su parte, el proveedor SunJCE también incluye una implementación de KeyStore en forma de archivo con un formato conocido como JCEKS. La clase KeyStore representa una colección de llaves y certificados, por lo que se encarga de gestionar dos tipos de entradas que son: entradas de llaves (públicas y privadas) y certificados digitales.

Figura 19: Clase KeyStore

Existen diferentes tipos de *keystore*, dentro de los cuales los más conocidos son JKS (proporcionado por el proveedor SUN), PKCS12 y JCEKS (proporcionado por el proveedor SunJCE).

Uno de los conceptos más importantes en el campo de la criptografía es la generación de números aleatorios. La clase base de un generador de números aleatorios es java.util.Random, introducida a partir de la versión 1.0 de Java, sin embargo, esta producía números pseudoaleatorios. De aquí surge la necesidad de una clase java.security.SecureRandom ya que implementa un generador de números pseudo-aleatorios fuertes criptográficamente. Al igual que otras clases basadas en algoritmos, SecureRandom proporcionar algoritmos que son independientes de su implementación.

El algoritmo de generación de números aleatorios proporcionado por el proveedor de SUN es conocido como SHA1PRNG y sigue el estándar IEEE P1363.

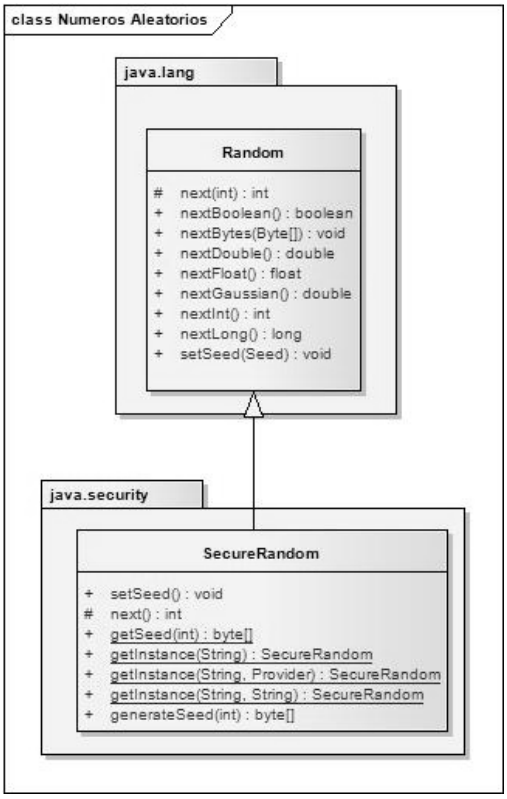


Figura 20: Clases Random y SecureRandom.

4.10.2 Clases criptográficas adicionales

La clase *javax.crypto.Cipher* se utiliza para el cifrado y descifrado de datos. Como todas las clases '*engine'*, se inicia creando una instancia mediante una llamada al método *getInstance()* pasando como parámetro el nombre de la transformación o algoritmo a utilizar. El nombre de la transformación se forma por una cadena de caracteres que contiene el nombre del algoritmo criptográfico seguido del modo y el esquema de relleno (*padding*). Para los algoritmos de cifrado por bloques, es posible especificar el tamaño del bloque dentro del mismo parámetro del modo.

Un objeto *Cipher* puede inicializarse en uno de cuatro estados posibles que son: *ENCRYPT_MODE*, *DECRYPT_MODE*, *WRAP_MODE* y *UNWRAP_MODE*. Para inicializarlo se ocupa alguno de los métodos init() y en él se especifica el modo de operación, la llave o certificado, los parámetros del algoritmo, así como una fuente de aleatoriedad. El cifrado y descifrado de datos se puede realizar en un solo paso o en múltiples pasos. Para cifrar en un solo paso, se utiliza alguno de los métodos *doFinal()* mientras que para realizarlo en múltiples pasos se hacen uno o más llamadas *update()* seguida de una llamada *doFinal()*. Después de llamar este método, el

objeto *Cipher* se restablece y se puede ocupar para cifrar o descifrar datos nuevamente, dependiendo del estado especificado al momento de la llamada *init*.

Para transferir las llaves de un lado a otro de manera segura, se requieren de algoritmos de encapsulado de llaves. En Java se utiliza la clase *Cipher* en modo *WRAP_MODE* y después se llama al método *wrap()*.

Algunos de los nombres de algoritmos, modos, y esquemas de relleno implementados y proporcionados por la plataforma de seguridad de Java son:

- **Algoritmos:** AES, Blowfish, DES, DESede (Triple DES), PBEWithMD5AndDES, PBEWithHmacSHA1AndDESede, RC2, RC4. RC5, RSA.
- **Modos:** NONE, CBC, CFB, ECB, OFB, PCBC
- **Esquemas de relleno:** NoPadding, OAEPWith<digest>And<mgf>Padding, PKCS5Padding, SSL3Padding.

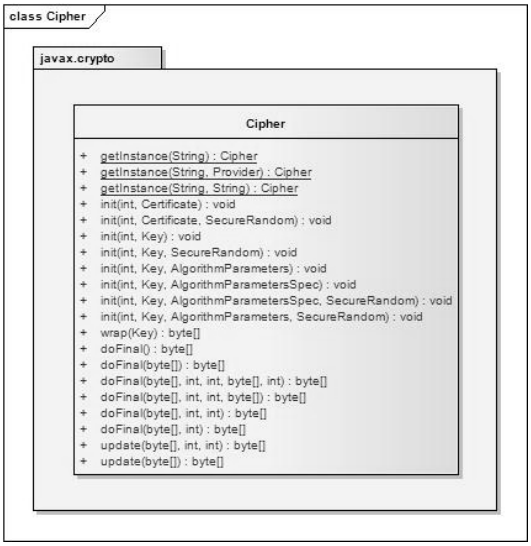


Figura 21: Clase Cipher.

Una clase *javax.crypto.KeyGenerator* es utilizada para generar llaves secretas utilizadas en algoritmos simétricos. Como es el caso de todas las clases '*engine'*, un objeto *KeyGenerator* se crea mediante la llamada a alguno de los métodos estáticos *getInstance()*, proporcionando el nombre del algoritmo simétrico para el cual se ocupará la llave. Al momento de inicializar un objeto se puede establecer un tamaño de llave y un origen de aleatoriedad, así como algunos parámetros propios de cada algoritmo. Para generar la llave, se utiliza el método *generateKey()*.

Los nombres de algoritmos que se pueden utilizar al momento de instanciar un objeto de este tipo pueden ser: AES, Blowfish, DES, DESede, HmacMD5 y HmacSHA1.

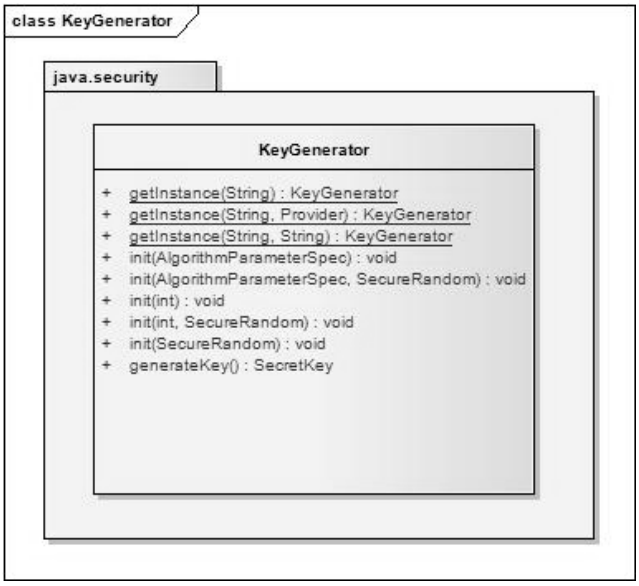


Figura 22: Clase *KeyGenerator*

La clase *javax.crypto.SecretKeyFactory* representa una fábrica de llaves secretas. Se utiliza para convertir llaves criptográficas "opacas" del tipo *java.security.Key* en una representación de especificación, que es la representación transparente del material de la llave.

Solo trabaja sobre llaves simétricas, mientras que por su parte el objeto *java.security.KeyFactory* se encarga del mismo procesamiento para los pares de llaves (pública y privada).

Los algoritmos que se pueden especificar al momento de solicitar una instancia del tipo SecretKeyFactory pueden ser: AES, DES, DESede, PBEWith<digest>And<encryption> o PBEWith<prf>And<encryption>.

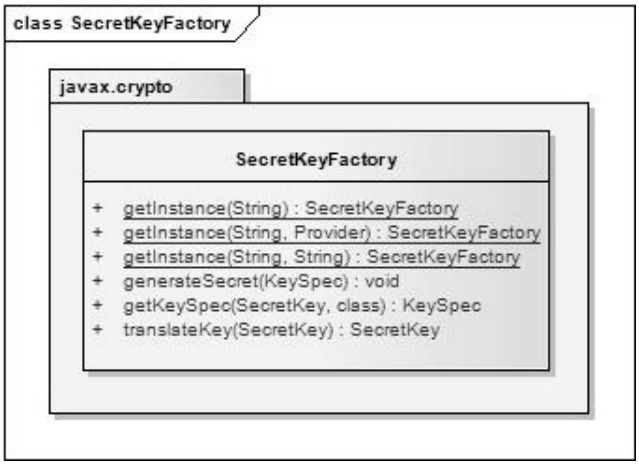


Figura 23: Clase SecretKeyFactory

La clase *javax.crypto.KeyAgreement* proporciona servicios de protocolo de acuerdo de llaves. Las llaves utilizadas para establecer el secreto compartido se crean mediante las clases *KeyPairGenerator*, *KeyGenerator*, *KeyFactory* o como resultado de una fase intermedia en el protocolo de acuerdo de llaves.

En este caso, cada entidad involucrada en el acuerdo de llaves tiene que crear un objeto *KeyAgreement*. Al ser una clase '*engine'*, también utiliza el método *getInstance()* para crear el objeto, seguido de una llamada al método *init()* para inicializarlo con los parámetros como llave, origen de aletoriedad y parámetros propios del algoritmo a utilizar.

Cada protocolo de acuerdo de llaves está compuesto por un número de fases que requieren ser ejecutadas por cada parte involucrada. Para ejecutar la fase siguiente se llama al método *doPhase().*

Después de que se ejecutan todas las fases requeridas para el acuerdo de llaves, cada parte puede calcular el secreto compartido llamando al método *generateSecret().*

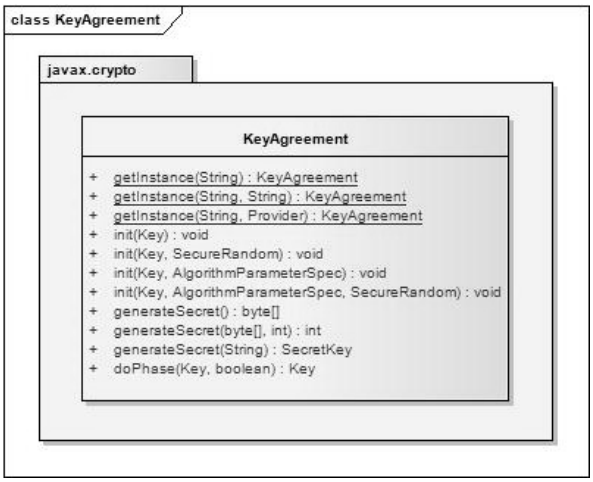


Figura 24: Clase KeyAgreement

La clase javax.crypto.Mac proporciona las funciones de un código de autenticación de mensaje (MAC). Al igual que todas las clases 'engine', utiliza un método getInstance() para instanciar un objeto del tipo dado. Posteriormente para inicializarlo, se hace uso de alguno de los métodos init() proporcionados por el objeto, ya sea con una llave secreta o mediante lla inicialización de un conjunto de parámetros que dependen del propio algoritmo MAC a utilizar. Al igual que el cifrado/descifrado de datos, el calculo del MAC puede utilizarse en uno o múltiples pasos.

Los algoritmos disponibles para instanciar objetos Mac son: HmacMD5, HmacSHA1 y PBEWith<mac>.

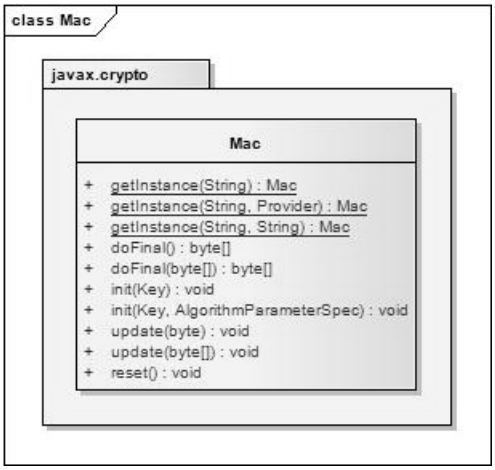


Figura 25: Clase Mac

Algunas de las implementaciones de proveedores de seguridad ampliamente utilizadas son la API de Bouncy Castle y Cryptix.

4.11 Modelos de confianza

Uno de los aspectos más importantes al momento de desarrollar aplicaciones confiables es la capacidad de establecer modelos de confianza entre entidades. Un modelo de confianza es un mecanismo utilizado por la arquitectura de seguridad para verificar la identidad de una entidad y sus datos asociados como pueden ser el nombre, llave pública, etc. Generalmente se logra mediante un proceso de autenticación de entidades. Un ejemplo de modelo de confianza es la estructura del certificado X.509

Los certificados X.509 asocia una llave pública con un nombre distinguido (DN). Los nombres distinguidos son muy utilizados en los servicios de directorio X.500 y en el caso de los certificados digitales se utilizan para identificar los nombres de las entidades del sujeto que representa el certificado y de la autoridad emisora del mismo. La información que contiene un nombre distinguido es:

- CN = Nombre común.
- OU = Unidad organizacional.
- O = Nombre de la organización.
- L = Nombre de la localidad indicando la ciudad.
- S = Nombre del estado
- C = País

Por otro lado, existen distintas versiones de certificados X.509 que son básicamente las siguientes:

- X.509 v1: Se trata de la versión inicial pero ya fue sustituido por la versión 3.
- X.509 v3: Soporta un concepto conocido como extensiones, de tal forma que cualquiera puede definir una extensión e incluirla en el certificado.
- X.509 v4: Incluye el soporte de certificados de atributos.

Todos los certificados X.509 incluyen la siguiente información:

- **Versión:** Versión del certificado digital X.509.
- **Número de serie:** Se refiere a un número de serie asignado por la entidad que emitió el certificado de tal forma que lo distingue de todos los demás certificados que la misma entidad emite.

- **Identificador del algoritmo de firma digital:** Algoritmo utilizado por la autoridad certificadora para firmar el certificado.
- **Emisor:** Nombre X.509 de la entidad que firmó el certificado. Generalmente se trata de la autoridad certificadora.
- **Periodo de validez:** Tiempo durante el cual el certificado es válido.
- **Nombre del sujeto:** Nombre de la entidad cuya llave pública se ha certificado.
- **Información de la llave pública del sujeto:** La llave pública del sujeto junto con el nombre del algoritmo que especifica a que criptosistema de llave pública pertenece, así como los parámetros de llave utilizados por el algoritmo.
- **Firma digital:** Firma digital de los campos anteriores.

Todos los datos del certificado se codifican utilizando 2 estándares que son Abstract Syntax Notation (ASN.1) y Distinguished Encoding Rules (DER). En ocasiones se utiliza la codificación Base64 en cuyo caso el certificado se limita por las líneas:
----- BEGIN CERTIFICATE -----
al inicio del certificado y
----- END CERTIFICATE -----
al final del mismo.

Un certificado se puede obtener de diversas maneras. Es posible generar un certificado auto-firmado[28] utilizando la herramienta en línea de comandos 'keytool' o generando una solicitud conocida como Certificate Signing Request (CSR) utilizando la misma herramienta 'keytool'.

En muchas ocasiones se requiere validar una cadena de certificados. Para esto, los estándares de infraestructura de llave pública (PKIX) define un algoritmo para la validación de cadenas de certificados X.509. En este caso, la API CertPath de Java proporciona un conjunto de clases e interfaces para integrar estas funciones en las aplicaciones que se muestra en la Figura 26.

[28] Self-signed certificate: Un certificado auto-firmado es aquel que se firma utilizando la llave privada que corresponde a la llave pública contenida en el mismo certificado.

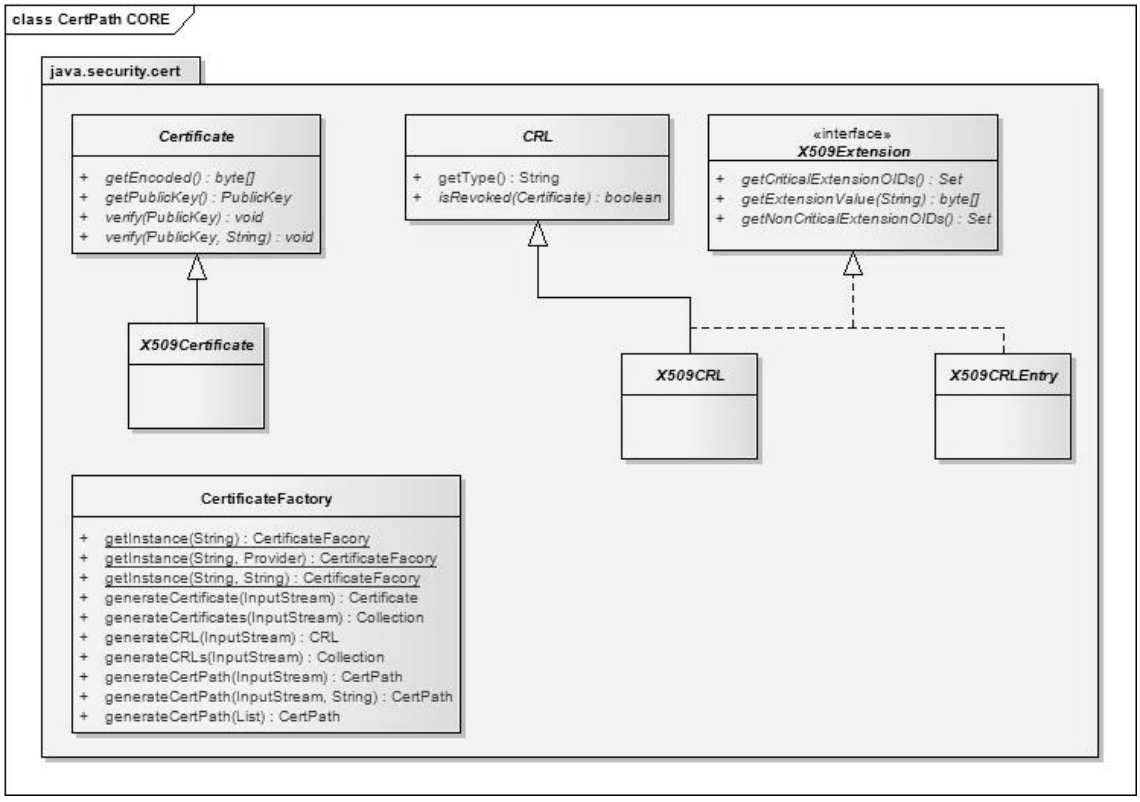


Figura 26: Principales clases e interfaces de la API Cert

La clase clave utilizada para la API de rutas de certificación de Java es *CertPath*. En este caso para crear una instancia del objeto *CertPath* utilizamos un objeto *CertificateFactory*.

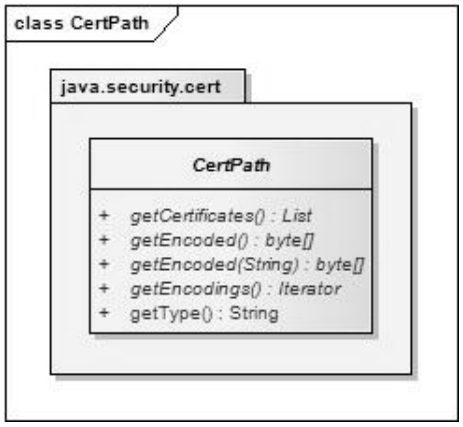


Figura 27: Clase CertPath

La API de ruta de certificación de Java incluye clases e interfaces para la validación de rutas de certificación. En este caso se utiliza una instancia de la clase 'engine' CertPathValidator para validar los certificados de un objeto CertPath, El resultado de la validación se regresa en un objeto CertPathValidatorResult.

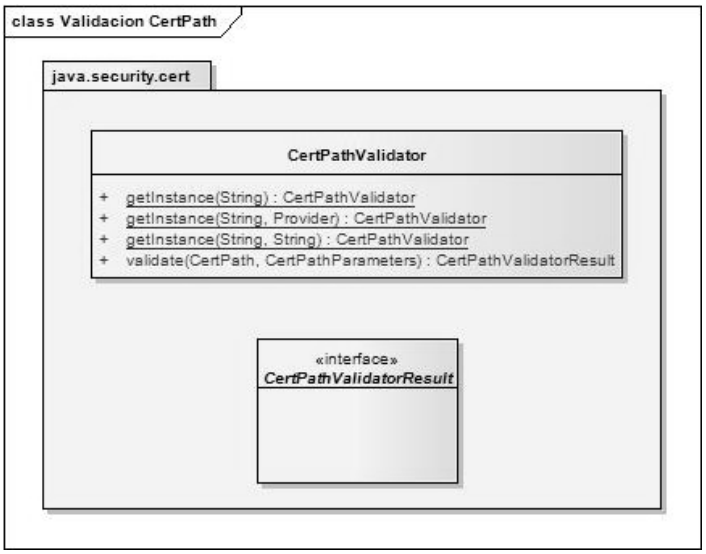


Figura 28: Clase CertPathValidator e interface CertPathValidatorResult.

Esta API también incluye clases para la construcción y el descubrimiento de rutas de certificación. Para crear un objeto *CertPath* se utiliza una instancia de la clase 'engine' *CertPathBuilder*. El objeto *CertPath* se obtiene a través del método *getCertPath* de la instancia *CertPathBuilderResult* regresada después de la llamada al método *build()*.

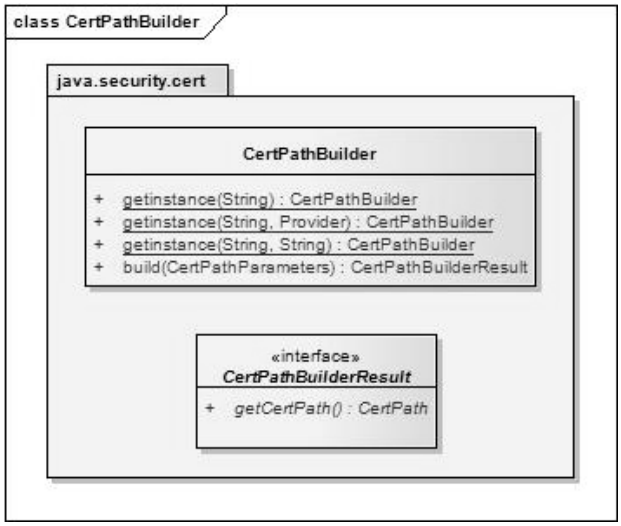


Figura 29: Clase CertPathBuilder e interface CertPathBuilderResult.

La API de ruta de certificación de Java también incluye una clase 'engine' *CertStore* utilizada para obtener certificados y listas de revocación de certificados desde un repositorio. Se utiliza junto con las clases *CerthPathBuilder* y *CertPathValidator* para especificar el repositorio que se utilizará para buscar los certificados y las listas de revocación de certificados. A diferencia de la clase *KeyStore* la cual proporciona acceso a una cache de llaves privadas y certificados confiables, la clase *CertStore* se diseñó para proporcionar acceso a un repositorio de certificados no-confiables, así como a listas de revocación de certificados.

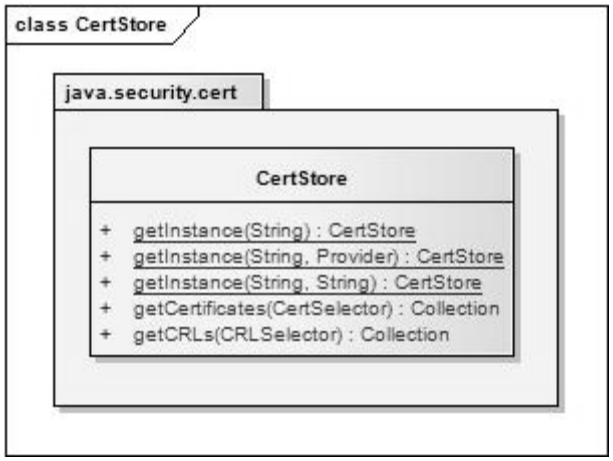


Figura 30: Clase CertStore

Para la gestión de código, el lenguaje de programación Java utiliza un formato de archivo que empaqueta un conjunto de clases conocido como formato jar (Java archive). Es similar a un archivo ZIP y define una estructura estándar y entradas de archivo que describen el contenido del archivo jar, controles de integridad e información adicional necesaria para verificar la integridad del contenido que se utilizará para realizar decisiones de confianza.

Los elementos de seguridad de un archivo JAR son:

- **Directorio META-INF:** Contiene todos los archivos de control utilizado para describir el contenido del archivo, incluyendo la información utilizada para proteger la integridad de los contenidos JAR. Es el directorio donde se encuentran todos los archivos mencionados a continuación.
- **Archivo manifiesto:** El archivo JAR contiene un archivo llamado MANIFEST.MF. Este archivo está compuesto por secciones que corresponden a entradas que corresponden a varios archivos que están en el archivo jar. Las secciones se separan entre sí mediante una línea vacía. No todos los archivos dentro del jar se listan en este archivo, pero si todos aquellos que se van a firmar digitalmente. Cada entrada dentro del archivo MANIFEST.MF

está compuesto por información de atributos de cada archivo como son el nombre del archivo o URL, el nombre del algoritmo hash utilizado, así como su valor hash en código Base64.

- **Archivo de instrucciones de firma:** Siempre que un archivo JAR se firma, se crea un archivo de instrucciones de firma y su correspondiente archivo de bloque de firma. Pueden existir múltiples archivos de instrucciones de firma, uno por cada firmante. Cada archivo de instrucciones de firma tiene una extensión .SF y tiene un formato similar al MANIFEST.MF con excepción de que los hash calculados corresponden a los hash de las entradas del archivo MANIFEST.MF.
- **Archivo de bloque de firma:** Este archivo tiene el mismo nombre que el correspondiente archivo de instrucciones de firma pero con una extensión diferente. La extensión varía dependiendo del algoritmo de firma digital utilizado.

De esta manera, la integridad de los archivos que se encuentran en un archivo JAR se protege mediante el cálculo de sus hashes.

Para validar un archivo JAR en su totalidad, el valor hash que se encuentra en cada una de las entradas del archivo de instrucciones de firma se compara con el hash calculado de la entrada correspondiente del archivo manifiesto. Después, el valor hash que se encuentra en el archivo manifiesto se compara con el hash calculado en el archivo correspondiente a la entrada mencionada. Al momento de cargar las clases, si la clase que se encuentra en el archivo JAR se verifica correctamente, el cargador de la clase establecerá los firmantes de la misma y se tomaran en consideración por la política de seguridad al momento de realizar una verificación de seguridad.

El subsistema de seguridad de la máquina virtual de Java requiere cierta consistencia en las llaves que se utilizan para firmar las clases dentro de un archivo JAR. Una restricción importante es el requisito conocido como mismo-paquete mismo-firmante. Esta restricción establece que las clases firmadas digitalmente de un mismo paquete, deben estar firmadas por los mismos firmantes.

La herramienta utilizada para firmar digitalmente los archivos JAR es conocido como jarsigner, y es una herramienta en línea de comandos disponible en el J2SE.

4.12 Servicio de Autenticación y Autorización de Java (JAAS)

4.12.1 Autenticación.

JAAS es una interfaz de aplicaciones que proporciona la facilidad de autenticar a los usuarios y tomar información de la autenticación para realizar decisiones de control de acceso (autorización). En las versiones iniciales de Java 2, las decisiones de acceso se basaban en las características del origen del código, si éste era firmado digitalmente y/o por quién, sin embargo, un control de acceso centrado en estas características es inusual, ya que las medidas tradicionales de seguridad se centran en el usuario. JAAS se diseñó para proporcionar una interfaz estándar de programación para autenticar usuarios y asignar permisos.

El modelo de autenticación de JAAS se basa en el mecanismo PAM[29] y soporta una arquitectura que permite que los administradores de sistemas decidan e instalen aquellos mecanismos de autenticación que cumplan con los requerimientos de seguridad establecidos. Su arquitectura también permite la independencia de la tecnología de autenticación que cada aplicación decida ocupar.

JAAS utiliza el término '*subject*' para referirse a una entidad, la cual puede representar un usuario del sistema o un servicio de cómputo. Casa sujeto o '*subject'* tiene asociados un conjunto de '*principals'*, los cuales representa el nombre asociado a un sujeto. Se requiere esta abstracción ya que generalmente una entidad o usuario utiliza distintos nombres para cada recurso de cómputo al que accede, como puede ser nombre de usuario LDAP, de base de datos, de red, ssh, correo, etc. En JAAS, el sujeto se representa por la clase *javax.security.auth.Subject* mientras que los *principal* utilizan la interfaz *java.security.Principal*.

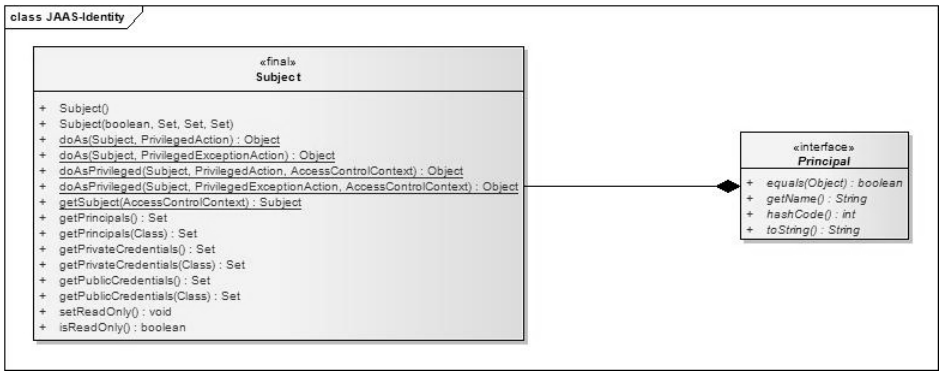


Figura 31: Clases utilizadas para la representación de entidades en JAAS.

[29] PAM (Pluggable Authentication Modules): Mecanismo que proporciona una interfaz entre las aplicaciones de usuario y diferentes métodos de autenticación.

Además de la información contenida en la interfaz *Principal*, la mayoría de los servicios requieren asociar otros atributos y datos a la clase *Subject*. JAAS los llama credenciales[30]. En JAAS, estas credenciales pueden ser cualquier tipo de objeto. Para hacer uso de estas credenciales, un sujeto debe autenticarse de manera correcta. JAAS divide las credenciales de cada *Subject* en dos conjuntos que son credenciales públicas y credenciales privadas y se acceden mediante los métodos *getPublicCredentials()* y *getPrivateCredentials()* respectivos.

A continuación se describe el proceso de autenticación del framework. Al momento que una aplicación intenta autenticar un usuario, JAAS define un contexto conocido como 'login context' representado por la clase *javax.security.auth.LoginContext*.

Esta clase proporciona los métodos básicos utilizados para autenticar sujetos. El objeto *LoginContext* consulta una instancia del tipo *javax.security.auth.login.Configuration* para determinar los mecanismos de autenticación o módulos *login*. Estos módulos son implementaciones de la interfaz *javax.security.auth.spi.LoginModule*. Cada módulo puede utilizar una tecnología de autenticación diferente. El administrador del sistema instala estos módulos dependiendo de los requerimientos de seguridad de cada aplicación teniendo la posibilidad de configurar métodos de autenticación alternos (Figura 32) o una pila de métodos de autenticación (Figura 61).

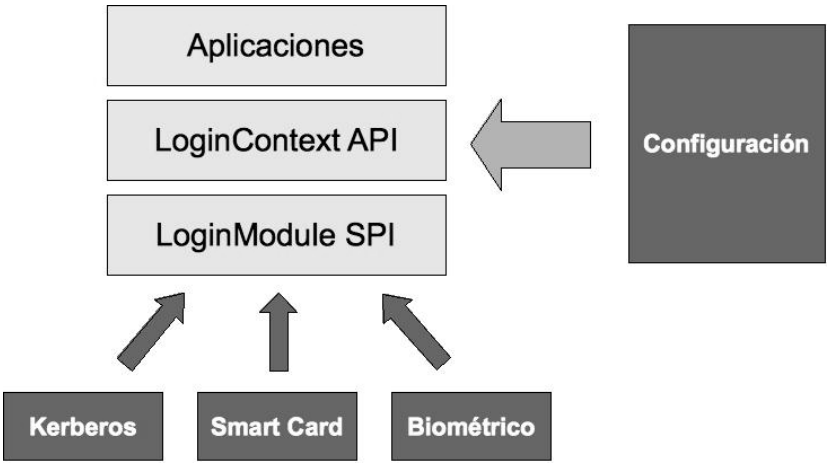


Figura 32: Autenticación modular.

[30] Credencial: Una credencial contiene información utilizada para autenticar al subject con los servicios adicionales. Algunos ejemplos pueden ser las contraseñas, tickets del protocolo Kerberos y certificados de llave pública.

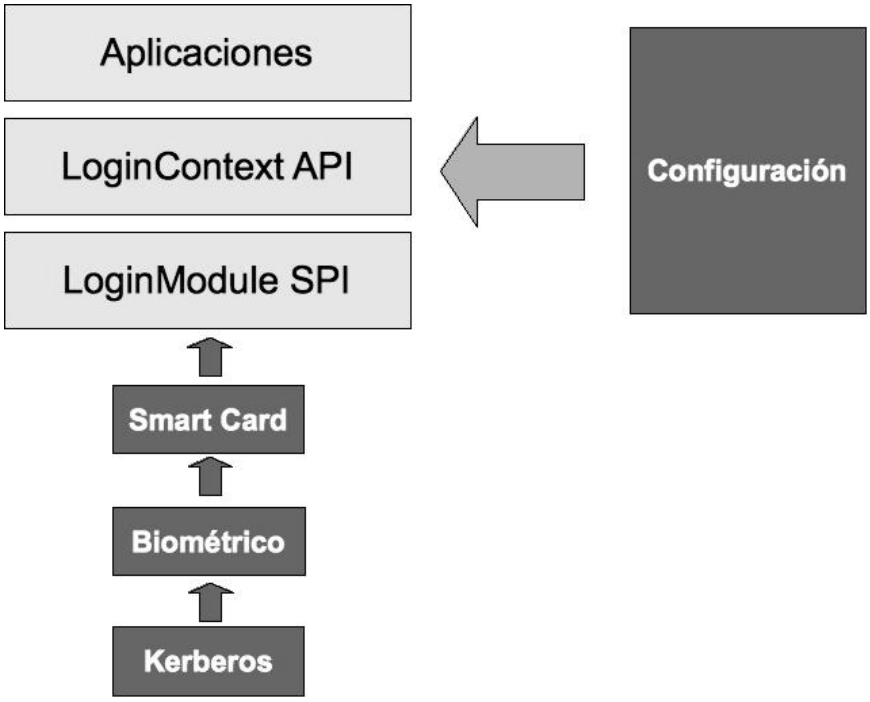


Figura 33: Autenticación apilada: Se deben cumplir más de un método de autenticación para accesar al sistema.

De acuerdo con los diagramas anteriores, un sujeto se autentica en los módulos login (*LoginModule*) en el orden especificado por un módulo de configuración (*Configuration*). En éste módulo se especifica el orden de autenticación y su prioridad.

En el caso de las configuraciones apiladas, la arquitectura de funcionamiento se diseñó de tal forma que si alguno de los módulos no pudo completar con la autenticación, el sujeto continúa el proceso con los módulos restantes, de tal forma que oculta información a posibles atacantes.

Una vez que el contexto de registro (*LoginContext*) realizó el proceso de autenticación para cada módulo configurado, éste reporta el estado de autenticación (si es válida o no) a la aplicación que lo llamó.

Para lograr lo anterior, el contexto de registro realiza la autenticación en dos fases. Ambas fases deben terminar exitosamente antes de regresar un estado de autenticación ya sea positivo o negativo.

Las fases de autenticación son:

1. **Fase 1:** *LoginContext* invoca cada módulo configurado y lo instruye a verificar la entidad de cada sujeto. Si todos los módulos necesarios completan esta fase satisfactoriamente, se pasa a la segunda fase.
2. **Fase 2:** *LoginContext* invoca cada módulo por segunda ocasión, instruyéndolos a completar el proceso de autenticación con una llamada '*commit'*. En esta etapa, cada módulo asocia los '*Principal*' con el *'Subject'*.

3. Si el proceso de autenticación falla en cualquiera de las fases anteriores, *LoginContext* invoca cada módulo configurado y lo instruye a abortar el proceso en su totalidad. En estos casos, cada módulo elimina cualquier estado o información relevante que haya asociado al momento de intentar la autenticación.

Es importante mencionar que ni *LoginContext* ni cada módulo *LoginModule* son responsables de las políticas de reintento de autenticación. Estos se programan a nivel de aplicación.

La arquitectura JAAS se puede utilizar en cualquier tipo de aplicación, ya sea web o de escritorio. Para esto, los módulos deben ser capaces de solicitar y desplegar información a los usuarios. JAAS utiliza un objeto del tipo *CallbackHandler* para proporcionar ésta funcionalidad.

Las aplicaciones proporcionan una implementación de *CallbackHandler* al momento de instanciar un *LoginContext*.

Cada módulo login recibe este objeto *CallbackHandler* de tal manera que sea posible llamar a los callback que lo conforman. El diagrama de secuencia de la Figura 62 muestra de manera más sencilla la interacción entre las clases involucradas en el proceso de autenticación JAAS.

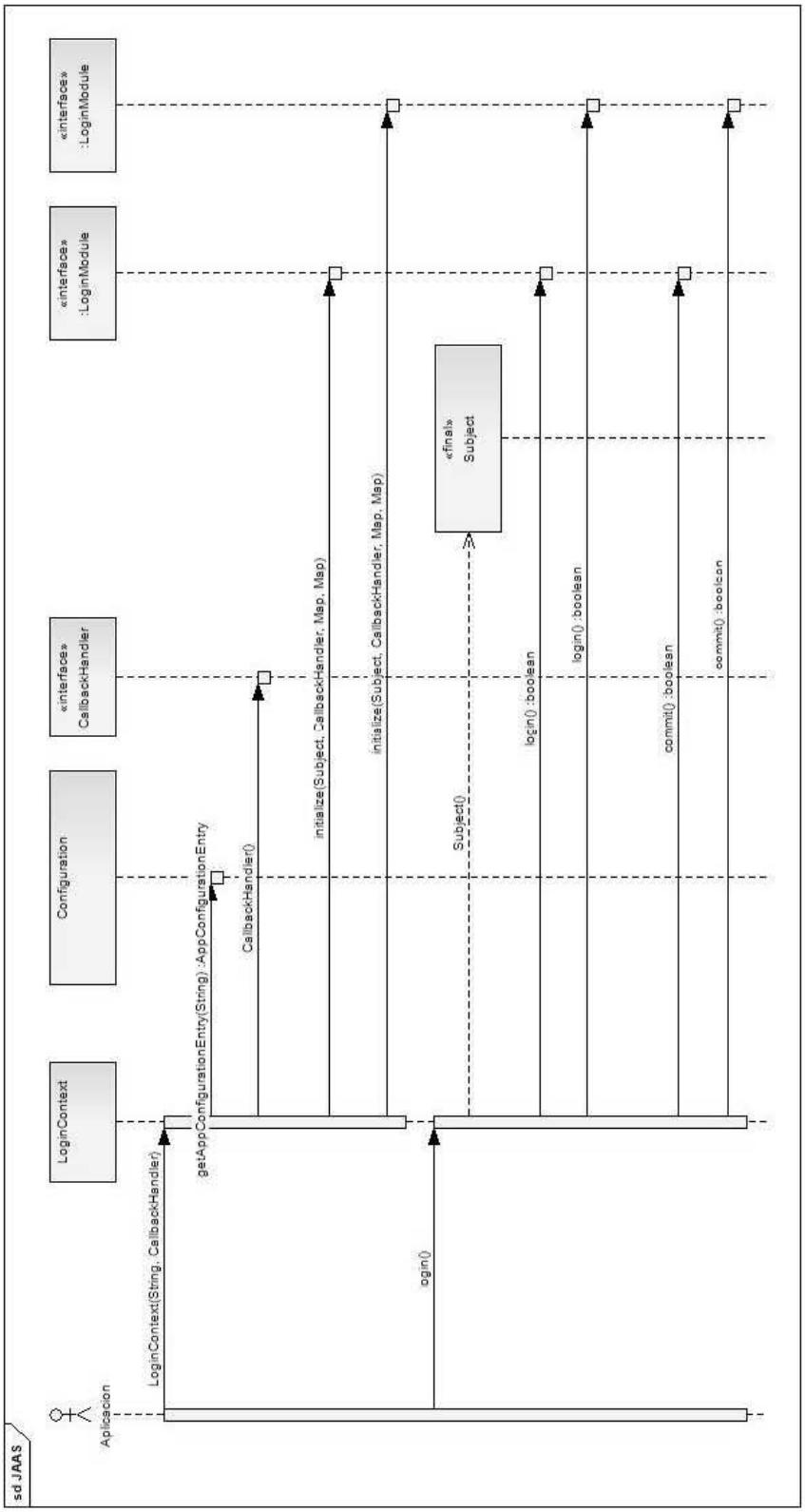


Figura 34: Proceso de Autenticación JAAS.

4.12.2 Autorización.

Una vez realizada la autenticación, la máquina virtual, a través de JAAS, utiliza esta información como base en su mecanismo de autorización o control de acceso. Como se mencionó anteriormente, la clase *Subject* solo representa un contenedor de información importante para cada usuario o servicio, mientras que la clase *Principal* representa entidades autenticadas asociadas a cada *Subject*, por lo tanto, el conjunto de permisos se otorgan a cada *Subject* basados en el conjunto de *Principal*s que contiene. Este conjunto de permisos se pueden configurar en una política de control de acceso externa.

JAAS extiende la arquitectura de control de acceso de la máquina virtual Java proporcionando la posibilidad de realizar un control de acceso basado en las características de los usuarios o servicios autenticados. Para esto, se pueden utilizar un conjunto de métodos estáticos de la clase *Subject* conocidos como *javax.security.auth.Subject.doAs()*. Estos métodos asocian dinámicamente un *Subject* con el contexto actual *AccessControlContext*. De esta forma, todo el código privilegiado se ejecuta en nombre del *Subject* asociado. Gracias a JAAS, el controlador de acceso puede basar sus decisiones en las características del código ejecutado y en el conjunto de *Principal*s asociados al *Subject*. Este funcionamiento se logra construyendo un *AccessControlContext* con una implementación de la interfaz *java.security.DomainCombainer* y asociándolo al hilo de ejecución actual invocando el método correspondiente *AccessController.doPrivileged()*.

Cuando una clase implementa la interfaz *DomainCombainer*, ésta tiene la posibilidad de aumentar y actualizar dinámicamente los dominios de protección asociados al *AccessControlContext* actual. La interfaz *DomainCombainer* se agregó a Java en la versión 1.3 con el propósito específico de proporcionar el soporte de control de acceso basado en principals.

La arquitectura del servicio JAAS está diseñado siguiendo dos patrones de seguridad importantes conocidos como “Authentication Enforcer” y “Authorization Enforcer”. En la estrategia propia de este servicio, el AuthenticationEnforcer se implementa como un cliente JAAS que interactúa con los LoginModules para la realización de la autenticación. Todos los módulos Login se configuran utilizando un archivo de configuración

4.12.3 Archivos de configuración.

Como se mencionó anteriormente, la configuración de los módulos de autenticación depende de la implementación de la clase abstracta *javax.security.auth.login.Configuration*. La implementación por default de Sun Microsystems obtiene la configuración a partir de un archivo de configuración. Este archivo consiste de una o más entradas, cada una de estas indica la tecnología de autenticación utilizada por una o más aplicaciones. Su estructura es:

```
<nombre utilizado por una aplicación para referirse a esta entrada>{
        <LoginModule><flag><opciones del LoginModule>;
        <LoginModules opcionales, flags y opciones del LoginModule>;
};
```

De esta manera un ejemplo de una entrada del archivo de configuración es:

```
Login {
        com.sun.security.auth.module.UnixLoginModule REQUIRED;
        com.abc.AbcLoginModule REQUIRED;
};
```

En este ejemplo, Login, es el nombre que utilizaríamos en nuestra aplicación al momento de instanciar la clase *LoginContext*. Cada entrada LoginModule está formado por las siguientes subintradas:

1. El nombre de una subclase de *LoginModule*. Algunas implementaciones proporcionadas por Sun Microsystems a través del paquete JRE son *UnixLoginModule*, *KeyStoreLoginModule* y *Kbr5LoginModule*.

2. Flag el cual es un valor que indica si el módulo es *REQUIRED*, *REQUISITE*, *SUFFICIENT* u *OPTIONAL*.

    a. *REQUIRED*: El usuario DEBE autenticarse en éste módulo. Si no es así, el proceso continua con los módulos siguientes solo para evitar proporcionar información a los posibles atacantes.

b. *REQUISITE*: El usuario DEBE autenticarse en éste módulo, sin embargo a diferencia del anterior, si no es posible autenticarse, el proceso no continúa en los módulos siguientes y regresa el control a la aplicación.

c. *SUFFICIENT*: No representa un requisito autenticarse en el módulo asociado, sin embargo, si lo logra, se regresa el control a la aplicación sin continuar con los módulos siguientes; si no, continúa con los módulos siguientes.

d. *OPTIONAL*: Como su nombre lo indica, es opcional. Independientemente si falla o no, el procedimiento de autenticación continúa en los módulos siguientes.

De esta manera, para que el proceso de autenticación termine exitosamente, todos los módulos REQUIRED y REQUISITE deben cumplirse. Si por el contrario no existe ningún módulo con estas etiquetas, al menos uno marcado como SUFFICIENT u OPTIONAL debe cumplirse.

3. Opciones adicionales específicas para el módulo.

Para especificar la localización de este archivo de configuración se puede hacer a través de la línea de comandos mediante el argumento *–Djava.security.auth.login.config* o modificando el valor de las variables *login.config.url.n* del archivo *java.security*, donde 'n' representa un número mayor a 0.

4.13 Comunicación Segura.

El protocolo de comunicación segura de amplio uso en la actualidad es el protocolo SSL. Este protocolo se desarrolló por Netscape en 1996 con el propósito de implementar una versión segura del protocolo HTTP, sin embargo se diseñó de tal forma que cualquier protocolo basado en TCP/IP pueda implementar una versión segura del mismo como son LDAPS, IMAPS, POPS, entre otros.

Este protocolo utiliza una combinación de algoritmos de llave pública y de llave simétrica para autenticar el lado del servidor y opcionalmente el lado del cliente, así como asegurar toda la comunicación entre ambas partes.

SSL incluye básicamente cuatro tipos de mensajes que son: mensajes de handshake, de alerta, de cambio de especificación de cifrado, mejor conocido como ChangeCipherSpec, y los datos de la propia aplicación. Una conexión SSL mantiene dos tipos de estado, un estado de sesión y un estado de conexión. El primero contiene los parámetros de seguridad negociados por ambas partes como son los certificados, los algoritmos de cifrado y un valor maestro “secreto”. Un estado de conexión se crea para cada conexión individual y mantiene parámetros de seguridad para tal conexión como son valores de la clave de cifrado y secuencia de números.

El protocolo SSL incluye dos sub-protocolos conocidos como protocolo de registro SSL y protocolo handshake SSL. El protocolo de registro SSL define el formato utilizado para la transmisión de datos, mientras que el protocolo de handshake utiliza el protocolo anterior para el intercambio de mensajes entre la máquina cliente y la máquina servidor. Dentro de este intercambio de mensajes se realizan las siguientes acciones:

1. Autenticación del servidor con el cliente.
2. Selección de algoritmos de cifrado soportados por el cliente y el servidor.
3. Autenticación opcional del cliente con el servidor.
4. Utilización de técnicas de cifrado de llave pública para la generación de los secretos compartidos.
5. Establecimiento de una conexión SSL cifrada.

La Figura 63 muestra un diagrama de actividad del handshake del protocolo SSL.

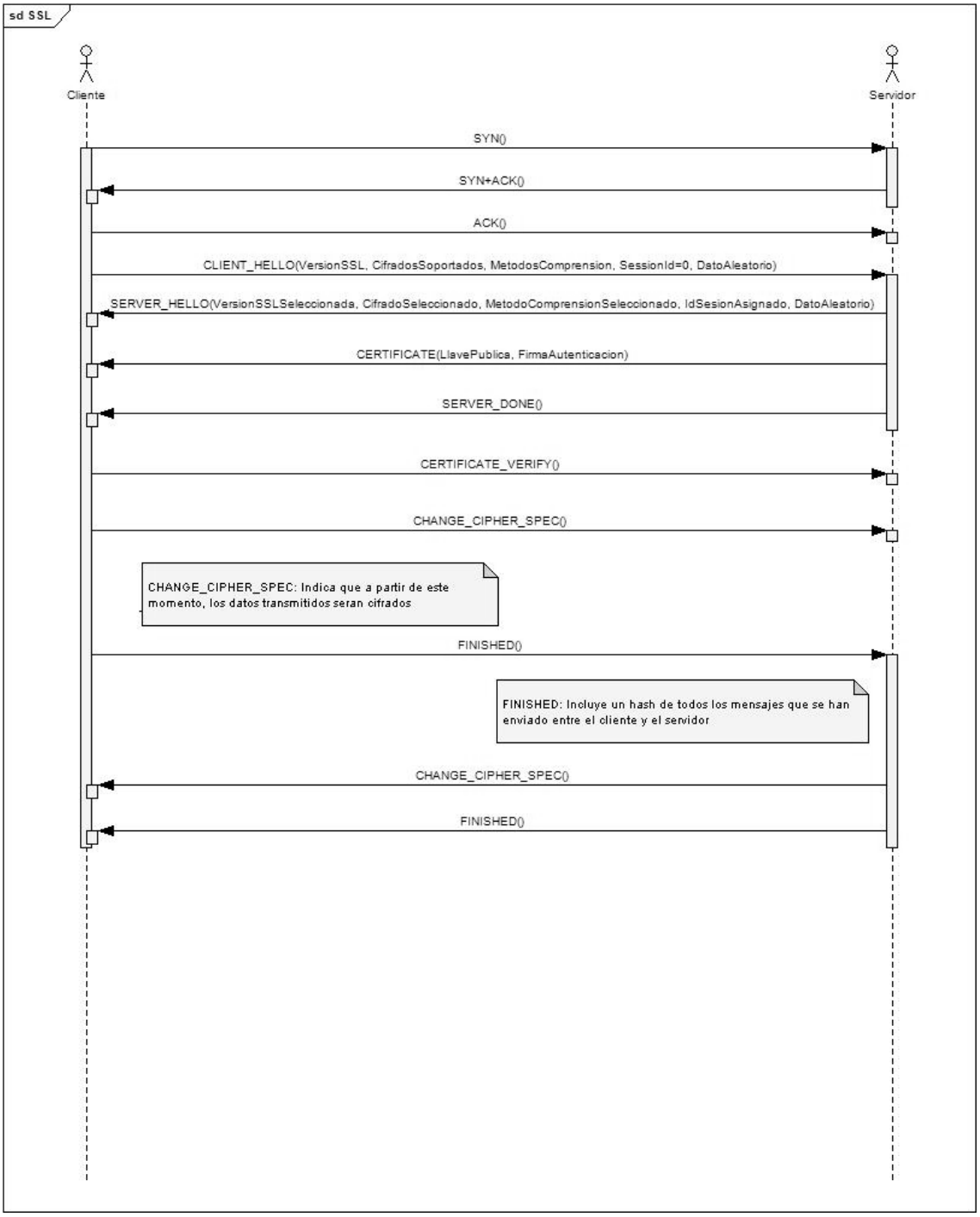


Figura 35: Handshake del protocolo SSL.

Es importante aclarar que el protocolo TLS[31] no es lo mismo que SSL. La especificación de TLS 1.0 está ampliamente basada en SSL 3.0 y se desarrolló como un intento por crear un protocolo estándar oficial de IETF[32], sin embargo ambos tienen pequeñas diferencias que los hacen imposibles de interoperar. En particular, la plataforma JAVA proporciona una comunicación segura a través de la tecnología conocida como JSSE[33]. JSSE define un API para el manejo de sockets SSL. Dentro de este API se pretende ocultar y/o abstraer todos los detalles de protocolo, aunque existe la opción de controlarla hasta cierto punto.

Las clases principales de JSSE se encuentran en los paquetes javax.net y javax.net.ssl y en el siguiente diagrama se muestra el conjunto de estas clases utilizadas para la creación de una comunicación segura.

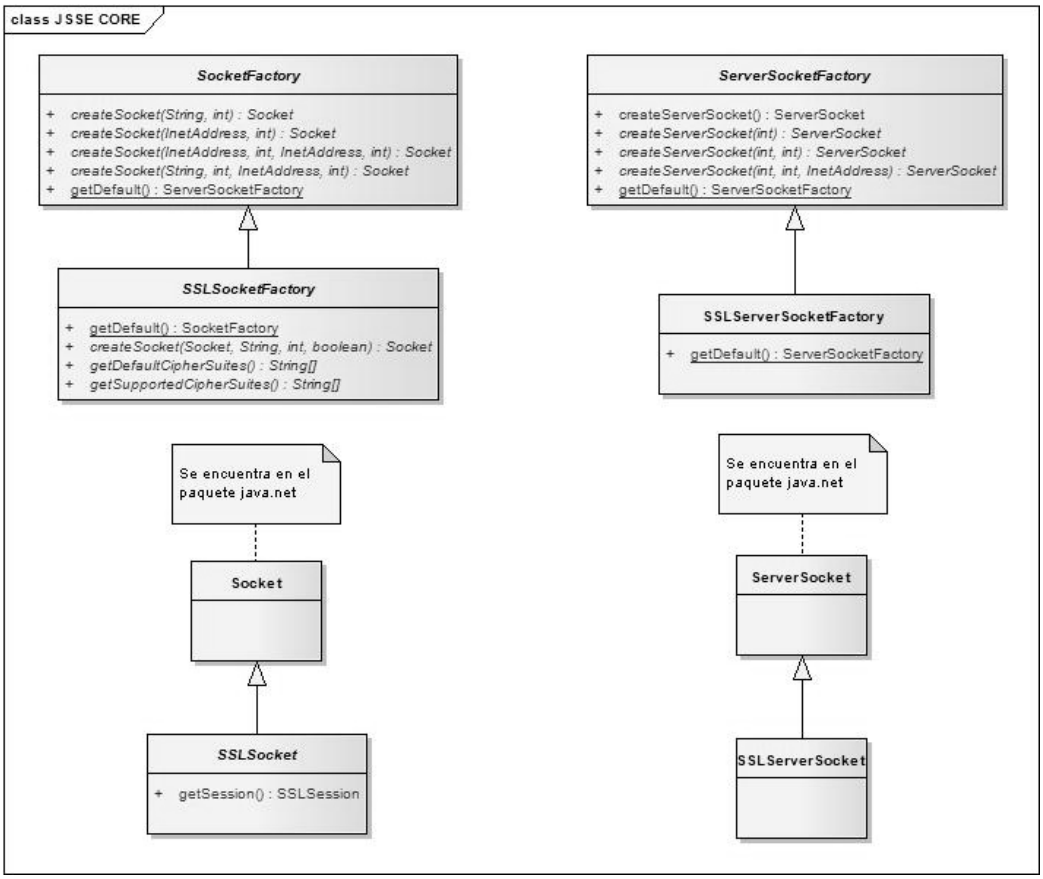


Figura 36: Clases principales de SSL para la creación de una comunicación segura.

[31] Transport Layer Security
[32] Internet Engineering Task Force
[33] Java Secure Socket Extension

Una comunicación segura depende de un par de sockets SSL cliente y SSL servidor que se crean a través de los métodos *createSocket()* y *createServerSocket()* de las clases *SSLSocketFactory* y *SSLServerFactory* respectivamente, previa creación de una instancia de las mismas a través de la llamada al método estático *getDefault()*.

La versión de JSSE de Sun Microsystems contiene un proveedor estándar conocido como SunJSSE. Este paquete proporciona los servicios siguientes:

- Implementación de un generador de llaves que soportan el algoritmo RSA[34].
- Un generador de pares de llaves para la creacuón de un par de llaves públicas y privadas apropiado para el algoritmo RSA.
- Un 'keystore' que soporta el formato PKCS12[35].
- Algoritmos de firma digital que soporta MD2withRSA, MD5withRSA y SHA1withRSA.
- Generadores de gestión de llaves y gestión de confianza que manipulan certificados X.509.
- Implementaciones de *SSLContext* para los protocolos SSL, SSLv3, TLS y TLSv1.

Como se muestra en el diagrama de actividad anterior, todo el proceso de comunicación segura a través de SSL requiere de un lugar donde se almacenen y verifiquen las credenciales necesarias para la comunicación entre las partes, entendiendo por credenciales a todos los certificados digitales, llaves privadas y secretas requeridas por el protocolo. Estos se representan mediante un *keystore* y un *truststore,* que en realidad se representan utilizando los mismos formatos y se administran por la misma herramienta '*keytool*'.

La diferencia radica en su utilización. El *keystore* se emplea por el servidor SSL para almacenar la llave privada y el certificado digital que lo identifica, mientras que el *truststore* es utilizado por el cliente SSL para verificar la validez del certificado enviado por el servidor, lo que significa que contiene los certificados de las autoridades certificadoras que permiten validar el certificado digital enviado por el servidor SSL. En caso de que el servidor SSL requiera autenticación del cliente (paso opcional en el protocolo SSL), el cliente deberá tener su *keystore* con su certificado digital y llave privada, y el servidor deberá tener su propio *truststore*.

Cuando se utilizan los sockets SSL genéricos a través de las clases *SSLSocketFactory* y *SSLServerSocketFactory*, estamos trabajando en realidad con muchas opciones por default como

---

[34] RSA: Sistema de cifrado por bloques de llave pública. Fue descrito en 1977 por Ron Rivest, Adi Shamir y Len Adleman en el MIT. El nombre del algoritmo debe su nombre a las iniciales de los apellidos de sus creadores.

[35] PKCS12: Estándar de formato de archivo para el almacenamiento de llaves privadas con su certificado de llave pública protegido mediante llave simétrica.

son las versiones de protocolos usadas (TLSv1, SSLv3) y las implementaciones del administrador de llaves y de confianza (keystore y truststore respectivamente). Si deseamos cambiar alguna de estas opciones, necesitamos utilizar la clase *SSLContext*.

La clase *SSLContext* está encargada de la implementación del protocolo SSL. Todo el paquete JSSE se implementa de manera similar al paquete JCA, es decir, la implementación real está definido en un paquete externo a la máquina virtual mediante proveedores de implementación, de tal manera que es posible elegir el proveedor que más convenga a nuestras necesidades y que cumpla con nuestros objetivos. *SSLContext* es una clase 'engine' que permite crear una instancia de la clase *SSLContext* basada en el proveedor elegido. Una vez creada ésta, podemos inicializarla con valores de administradores de llaves (*key manager*), administradores de confianza (*trust manager*) determinados. El siguiente diagrama muestra a grandes rasgos la relación que existe entre las clases involucradas en la creación de un servicio SSL personalizado.

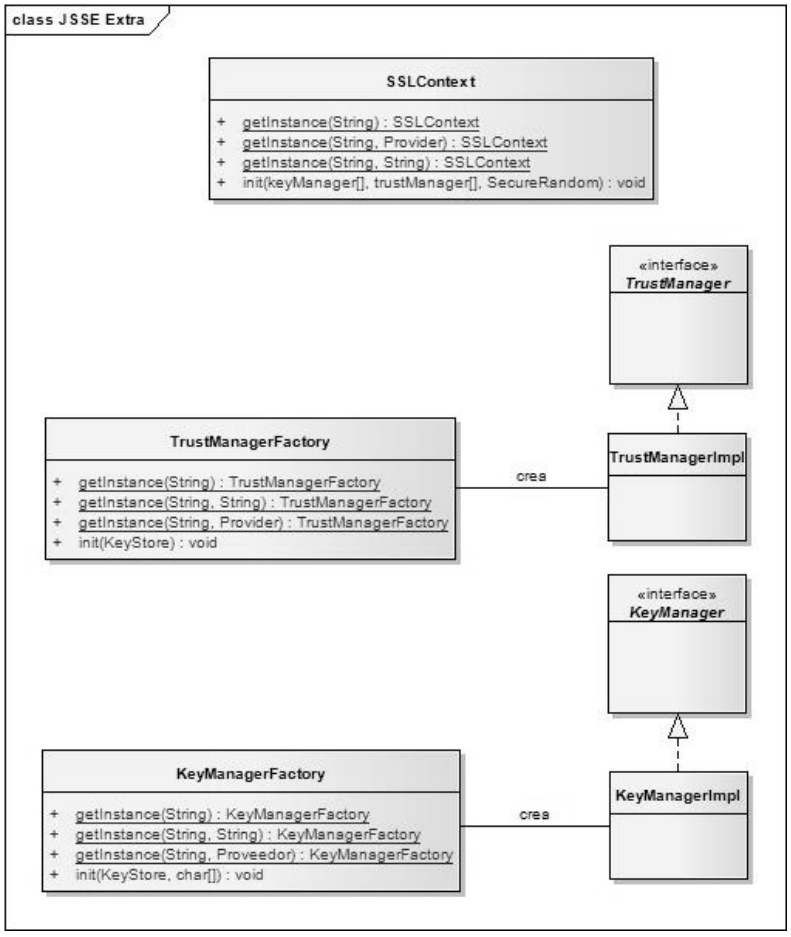


Figura 37: Clases involucradas en la creación de una sesión SSL personalizada.

4.14 Técnicas de Programación Segura.

En este capítulo se han mencionado muchas de las características de seguridad incorporadas en el lenguaje de programación Java, sin embargo, todas estas características deben de implementarse adecuadamente en el desarrollo de cualquier aplicación. En este aspecto el equipo CERT[36] ha publicado recomendaciones de codificación segura para los lenguajes Java, C y C++. A continuación se mencionan algunas de las reglas y prácticas recomendadas para una programación segura en Java.

4.14.1 Manejo de Excepciones de Seguridad.

Una práctica común en el manejo de excepciones consiste en cachar una excepción y lanzar en su lugar otra distinta, esto con el objeto de ocultar el origen de la primera excepción. Este manejo de las excepciones es conocida como 'swallow'. Sin embargo, en el caso de las excepciones de seguridad *java.security.AccessControlException* y *java.lang.SecurityException*, esta práctica no se recomienda, ya que puede crear huecos importantes de seguridad.

4.14.2 Métodos y campos.

El lenguaje de programación incorpora cuatro niveles de acceso a los miembros de clase, ya sea variables o métodos, los cuales son: *public*, *private*, *protected* y *default*. Un error muy común es cuando se declara una variable con un control de acceso *'public'*. Esta práctica provoca que la variable en cuestión sea modificada de manera directa por cualquier persona o código, por lo que no existe un control del rango o tipo de valor que de acuerdo al contexto de la variable, pueda tener. Si es necesario agregar comprobaciones o características adicionales antes de modificar la variable, esta práctica no lo permite, por lo que se recomienda crear un método 'set' y un método 'get' que permita acceder a la variable y cambiar el modificador de acceso de la misma a *'private'*.

Los modificadores de acceso *'protected'* y *default* tampoco se recomiendan para las variables de clase, ya que estos permitirían la modificación de las mismas ya sea a través de métodos del mismo paquete o de una subclase de la misma.

[36] Computer Emergency Response Team o Equipo de respuesta a emergencias informáticas.

Es por esto que se recomienda poner especial atención en cada uno de los modificadores de acceso de variables y métodos, revisar si requieren de métodos 'set' y/o 'get', así como verificar si el acceso de alguna de estas variables no requiere de un permiso en particular.

En el caso de las variables constantes, es posible declararla *'public'* siempre y cuando se agreguen los modificadores *'final'* y *'static'*.

4.14.3 Campos estáticos.

Un campo estático (*static*) es aquel que se comparte por todos los objetos instanciados por la misma clase. Generalmente se utilizan para crear variables constantes ampliamente utilizadas en nuestras aplicaciones.

El uso incorrecto de las variables estáticas resulta muy peligroso, ya que éstas, a diferencia de las variables de instancia, proporcionan menor protección. Esto se debe en gran medida a que para acceder a un campo o variable estática no se requiere de una referencia al objeto, sino simplemente del nombre de la clase. Es importante verificar la no existencia de variables *no-final public static* y en caso de que existan, modificarse por *private static* y crear los métodos 'set' y/o 'get' con los modificadores *'public static'*.

4.14.4 Estado e inmutabilidad de objetos.

Un concepto que requiere de especial atención en el paradigma de programación a objetos y en especial en el lenguaje de programación Java, es el de mutabilidad de objetos. Para esto, es importante mencionar que éste lenguaje utiliza paso de variables por valor para ambos tipos de variables, ya sea primitivas o de referencia.

Decimos que un objeto es mutable, cuando a través de una referencia a la instancia de un objeto, podemos alterar el contenido de la misma. Por otro lado, decimos que un objeto es inmutable, cuando, aún teniendo la referencia a la instancia del objeto, no es posible alterar los contenidos del mismo.

La mayoría de los objetos tienen un estado interno, privado, establecido por sus variables de instancia y con métodos *'set'* o *'get'* utilizados para consultar y modificar su estado. Los métodos *'get'*, generalmente se implementan mediante una simple declaración de un *return* como

se muestra en el ejemplo siguiente.

```
public MiClase{
        private boolean status = false;
        ...
        public boolean getStatus(){
                return status;
        }
}
```

En este ejemplo no existe problema de mutabilidad, sin embargo, si el código fuera como el que se muestra a continuación, los problemas de seguridad podrían presentarse.

```
public MiClase {
        private boolean[] status = null;
        ...
        public boolean[] getStatus(){
                return status;
        }
}
```

En el ejemplo anterior, cuando se realiza una llamada al método *getStatus*, obtenemos una referencia a la variable privada *'status'*, y el objeto puede cambiar el valor de la variable sin el consentimiento de la clase *MiClase*. La razón de que esto sea posible es debido a que un arreglo de *boolean*, a diferencia de una variable del tipo *boolean*, es mutable, ya que un arreglo no es un elemento primitivo sino un objeto.

Por otro, para el mismo método, la manera más sencilla de implementar un método 'set' es mediante el código siguiente:

```
public void setStatus(boolean[] s){
        status=s;
}
```

Sin embargo, debido a que la misma variable *'s'* es mutable, a pesar de que aparentemente la clase *MiClase* toma posesión de la variable *'s'*, el objeto que proporciona el valor de la variable, en ningún momento se despoja de la misma, por lo que aún puede modificar el valor de la misma variable sin el uso del método *'set'*. La mejor práctica al implementar este método, es copiar o clonar los objetos mutables antes de almacenarlos.

Por todo lo anterior, es importante distinguir los objetos inmutables de los mutables y resaltar la importancia de hacer objetos inmutables cada que sea posible.

Para poder implementar una clase inmutable debemos considerar las siguientes recomendaciones de implementación:

1. Todos los campos de la clase deben tener un control de acceso '*private'*;
2. No deben tener métodos *get* y *set*.
3. Asegurarse que los métodos no pueden sobrescribirse por una subclase. Esto se puede lograr declarando la clase completa como '*final*' o declarando sus métodos como '*final*'.
4. Si alguna variable no es de tipo primitivo o inmutable, es necesario realizar un clon completo de la misma, es decir, no utilizar la referencia del objeto creado.

Un caso de uso que requiere de especial atención resulta al momento de implementar las contraseñas. Generalmente, una contraseña la visualizamos como una cadena de caracteres y utilizamos el objeto *String* para implementarla. Dado que el objeto *String* es inmutable, no existe manera de borrarla una vez que se ha dejado de ocupar, por lo que su eliminación depende totalmente del colector de basura de la máquina virtual. Una implementación más segura se realizaría utilizando un objeto *char[]*, ya que esto permitiría borrar el contenido del arreglo una vez que la variable se ha dejado de ocupar.

4.14.5 Código privilegiado.

Como se ha resaltado anteriormente, en muchas ocasiones existe la necesidad de ejecutar código privilegiado en un momento dado. Esto se realiza con la ayuda del método *AccessController.doPrivileged* que permite clasificar un código determinado como código privilegiado. Esta pieza de código privilegiado lo convierte en una región de seguridad crítica, donde es necesario tener atención especial.

Una de las recomendaciones más importantes en este segmento de código, es mantenerlo lo más pequeño posible, de esta manera, reducimos la posibilidad de cometer errores y aumentamos la facilidad al momento de auditarlo.

Algunos de los puntos más importantes al momento de implementar un código privilegiado es colocar el mismo dentro de un método que no sea *'public'*, de otra manera estaríamos permitiendo que el acceso al recurso protegido sea accesible por cualquier código externo. Por otro lado, cualquier variable que sea utilizada por el mismo código privilegiado debe ser del control del mismo código, es decir, la implementación del código debe asegurar que el acceso al recurso compartido sea lo más limitada posible.

4.14.6 Serialización.

La serialización es una característica que permite almacenar el estado de un objeto como flujo de bytes, con el propósito de transportar el objeto a otra máquina y deserializarlo o reconstruirlo.

Los objetos se serializan y deserializan mediante las clases *ObjectOutputStream* y *ObjectInputStream* a través de los métodos *writeObject* y *readObject*, respectivamente. También es posible personalizar la manera en cómo se serializan y reconstruyen a través de una implementación de la interfaz *Serializable*. En este caso, implementamos los métodos writeObject y readObject. Al momento de implementarlos es importante pensar que el método readObject en realidad funciona como un constructor público que toma como entrada los valores de cada uno de los campos no-*transient* del objeto serializado y crea una nueva instancia del objeto. Bajo estas circunstancias, los desarrolladores deben asegurarse que las mismas restricciones de seguridad aplicadas en un constructor público se apliquen al momento de reconstruir el objeto serializado.

4.14.7 Clases internas (Inner Classes)

El uso de este tipo de clases tiene algunas implicaciones de seguridad. Por ejemplo, cuando creamos una clase A y declaramos un campo '*private'*, accesible solo desde la misma clase y además declaramos dentro de la misma clase, una clase interna B que accede a nuestra variable '*private'*, al momento de la compilación se crea un método con acceso default, es decir, accesible mediante cualquier clase del mismo paquete, que permite acceder a nuestra variable '*private'*. Gracias a éste método creado al momento de compilación, nuestra clase interna B puede acceder a la variable '*private'*. De esta manera, cualquier clase dentro del mismo paquete podrá tener acceso a nuestra variable '*private'* a través del método creado.

Otra implicación de seguridad se debe a la misma naturaleza de estas clases. Cuando declaramos una clase interna B con un nivel de acceso '*protected*' dentro de la clase A, después de compilarse, el archivo *class* B se define como una clase '*public*', pero un atributo en el mismo archivo *class* describe correctamente su nivel de acceso. De la misma manera, si la clase interna B es un miembro '*private*' de A, el archivo *class* de B se define con un acceso '*default*' con un atributo que declara el modo de protección de acceso verdadero. Esto solo trae problemas para la implementación de la máquina virtual, ya que ésta debe preocuparse de realizar las verificaciones de acceso adicionales para que los niveles de acceso se cumplan de manera adecuada.

4.14.8 Métodos nativos

Los métodos nativos (*native*), por definición están fuera del sistema de seguridad de la plataforma Java, es decir, ni el gestor de seguridad ni el controlador de acceso pueden controlar su comportamiento. Por lo tanto, es importante examinar los parámetros que reciben, sus valores de retorno, así como el nivel de acceso a los mismos.

4.14.9 Objetos firmados.

Existen algunas situaciones en las cuales la autenticidad de un objeto y su estado deben de asegurarse. Algunos ejemplos son:

- Cuando un objeto funciona como elemento de autenticación o autorización debe de cumplir con características importantes de seguridad para que no pueda ser falsificado.
- Cuando un objeto es transportado entre máquinas virtuales y se requiere verificar su autenticidad.
- Cuando el estado de un objeto se almacena fuera de la máquina virtual.

La clase *java.security.SignedObject* define las interfaces para firmar un objeto. También es posible anidar estos objetos para crear una secuencia lógica de firmas que semejan una cadena de autorización y delegación.

El objeto a firmarse debe de implementar la interfaz *Serializable*, ya que cuando se crea el objeto *SignedObject*, el objeto se serializa antes de generar la firma digital. De esta manera, un *SignedObject* está conformado por un objeto serializado, su firma digital y el nombre del algoritmo utilizado para generar la firma. El objeto firmado y serializado corresponde a una copia completa del objeto original, de hecho el objeto *SignedObject* es inmutable. La Figura 38 muestra el concepto general de los objetos firmados, mientras que Figura 67 muestra los métodos proporcionados por la clase *SignedObject*.

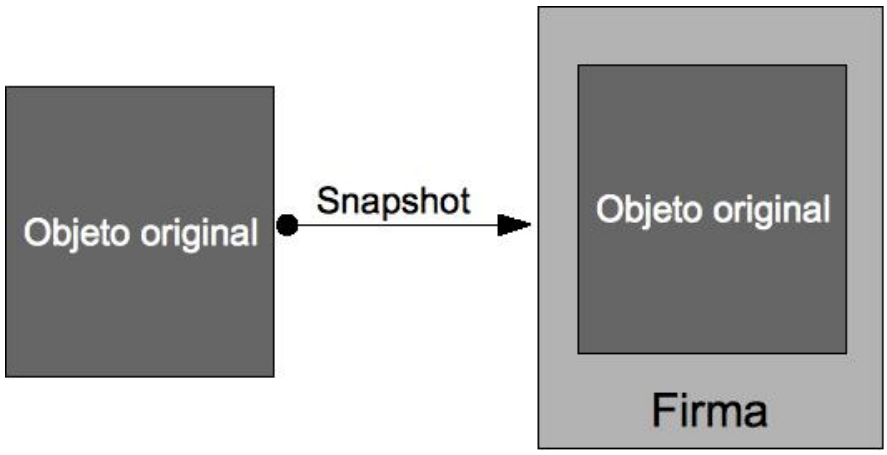


Figura 38: Concepto de objeto firmado

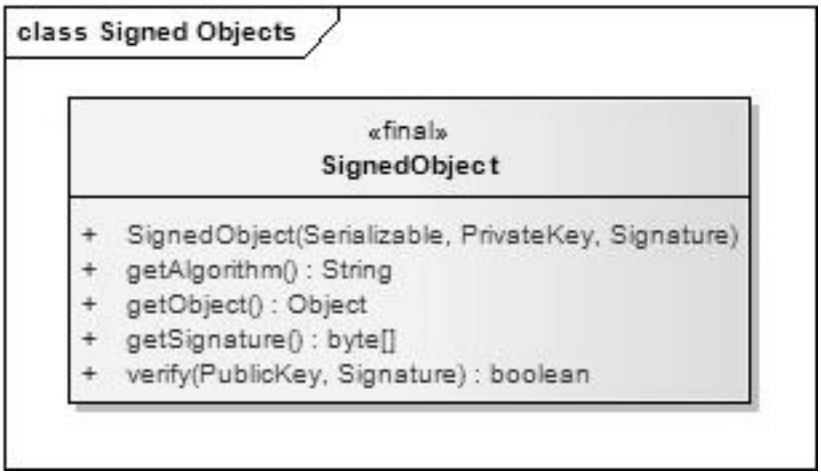


Figura 39: Diagrama de la clase SignedObject.

4.14.10 Objetos cifrados

Mientras que la clase *SignedObject* proporciona servicios de autenticidad e integridad, la clase *SealedObject* protege la confidencialidad de un objeto. Ambas clases pueden combinarse para proporcionar los servicios de confidencialidad e integridad al mismo tiempo.

Al igual que la clase *SignedObject*, *SealedObject* toma como parámetro una clase que implemente la interfaz *Serializable*, después al objeto en formato serializado, se le aplica el algoritmo de cifrado elegido. La Figura 68 muestra los métodos proporcionados por esta clase.

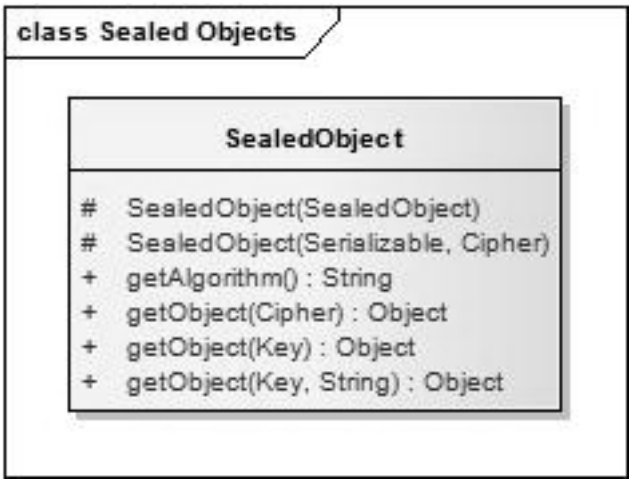


Figura 40: Clase SealedObject

4.14.11 Objetos vigilantes

A partir de la versión 1.2 de Java se introdujeron en la API una interfaz y una clase nuevas llamadas *java.security.Guard* y *java.security.GuardedObject*, respectivamente, utilizadas para proporcionar un control de acceso a nivel de objeto. Un objeto *GuardedObject* se utiliza para controlar el acceso a otro objeto. Esto lo logra encapsulado el objeto a vigilar con un objeto que implementa la interfaz *Guard*. Cuando un objeto se encapsula de esta manera, el acceso a este objeto se controla por el método *getObject*(). Este método invoca el método *checkGuard()* del objeto *Guard* encapsulado. En caso que el acceso no esté permitido, el método lanzará una *SecurityException*. La Figura 41 muestra como al solicitar una referencia al objeto protegido, ésta solo se logrará si el objeto *Guard* lo permite.

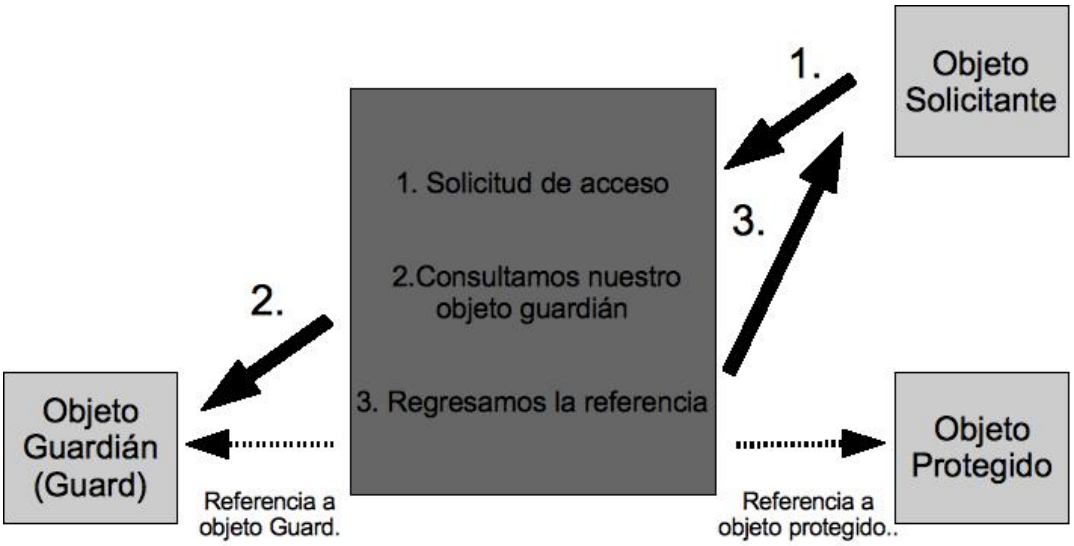


Figura 41: Funcionamiento e interacción de los elementos GuardedObject y Guard

Debido a que la clase base *Permission* implementa la interfaz *Guard*, todos los permisos de este tipo, incluyendo las subclases, puede utilizarse como objetos *Guard*.

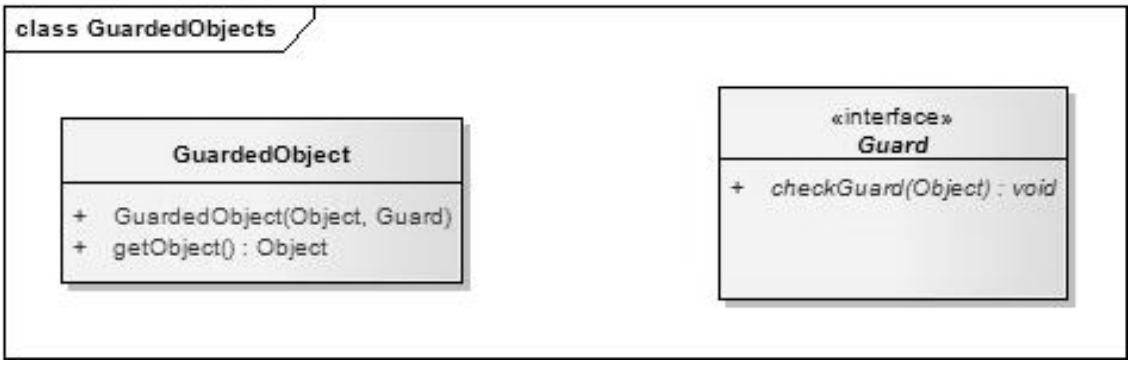


Figura 42: Clase GuardedObject e interfaz Guard

Uno de los motivos más fuertes para la creación de la clase *GuardedObject* es debido a que en muchas ocasiones el proveedor de cierto servicio no se encuentra en el mismo contexto de ejecución que el consumidor del mismo. En estos casos una verificación de seguridad basado en el contexto de seguridad del proveedor del mismo es inapropiado ya que ésta debiera hacerse basado en el contexto de seguridad del consumidor. La clase *GuardedObject* se diseñó de tal forma que el proveedor del servicio pueda crear un objeto representando el recurso y un objeto *GuardedObject* que contiene el objeto recurso y de esta manera proporcionar el objeto *GuardedObject* al cliente junto con un objeto *Guard.*

4.14.12 Ofuscado de código.

Una de las amenazas más importantes para las aplicaciones Java es la ingeniería inversa. Mediante este mecanismo es posible decompilar el bytecode de las clases compiladas Java y obtener el código fuente de la aplicación. Este proceso permite entre otras cosas, modificar el código y los datos, determinar el flujo de la ejecución del programa, conocer los algoritmos implementados, entre otras cosas.

Para evitar la posibilidad de aplicar estas técnicas de ingeniería inversa, existen algunas herramientas y técnicas que permiten proteger el código fuente. Algunas opciones son:

1. Autenticación de código:
2. Cifrado y descifrado de código:
3. Ofuscado de código:

El ofuscado de código es el proceso de transformar un ejecutable de manera tal que afecte los mecanismos de ingeniería reversa, es decir, mediante éste proceso se rompe con las relaciones existentes entre un código ejecutable y su código fuente lo que provoca que la decompilación del código no sea efectivo. A pesar de todos estos cambios, el programa ofuscado funciona de manera idéntica al ejecutable original. Para la plataforma Java, algunos ejemplos de decompiladores de código son Jad, DJ Java decompiler y JD-GUI.

4.15 J2EE y su arquitectura de seguridad.

4.15.1 Arquitectura J2EE.

La plataforma J2EE es básicamente un ambiente de cómputo distribuido basado en Java que define una arquitectura de aplicación multi-capa, especialmente diseñada para las aplicaciones empresariales. Esta plataforma se representa mediante las siguientes capas lógicas:

- ***Capa de Cliente:*** Representa la interfaz de usuarios de la plataforma J2EE y puede tratarse de una aplicación Java, un applet, un navegador web, un servicio web o un dispositivo móvil.

- ***Capa web o de presentación***: Se encarga de manipular las solicitudes y respuestas, la gestión de sesiones, entrega de contenido independiente del dispositivo, así como la invocación de componentes de negocio. Desde el punto de vista de la seguridad, se encarga de las sesiones de login de usuario y establece los controles de acceso 'single sign-on' para los componentes de aplicación principales. Los componentes de esta capa generalmente se tratan de Servlets, JSP's o JSF´s y se encuentran en el contenedor web, el cual se encarga de entregar las interfaces de usuario a los clientes.
- ***Capa de aplicación o negocio:*** Esta capa generalmente se encarga de todas las funciones del negocio, así como de las transacciones hacia los recursos 'back-end'. Los componentes J2EE que generalmente se encuentran en esta capa, se refieren a los EJB, beans de sesión, beans de entidad, así como beans para el manejo de mensajes.
- ***Capa de integración:*** Esta capa representa la conexión y comunicación con la capa de recursos. Los componentes de esta capa pueden ser JMS, conectores J2EE y componentes JDBC.
- ***Capa de recursos:*** Esta capa generalmente se compone de bases de datos, sistemas de información empresarial, aplicaciones 'mainframe', entre otras.

Cada uno de los componentes mencionados en estas capas, generalmente se encuentran en un contenedor web o en un contenedor EJB.

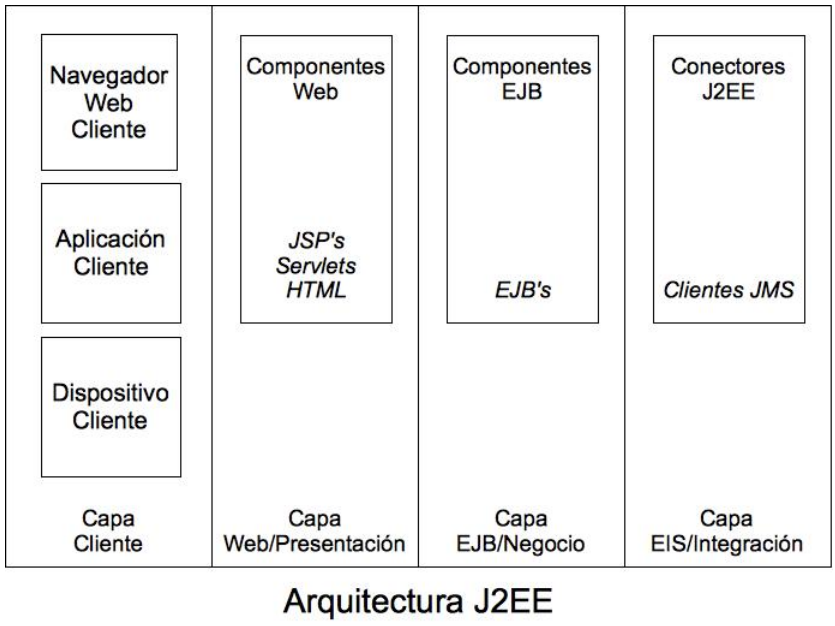


Figura 43: Plataforma J2EE y sus capas lógicas.

4.15.2 Infraestructura de Seguridad J2EE

La plataforma J2EE proporciona una solución de infraestructura de múltiples capas para el desarrollo de aplicaciones de cómputo distribuidas, portables y escalables. Esta plataforma también está diseñada para proporcionar un modelo de seguridad completo que aborda los principales requerimientos de seguridad en una infraestructura de aplicación.

La manera en que se afronta la seguridad en la plataforma es proporcionando seguridad en cada componente de la infraestructura, cumpliendo con los requerimientos de integridad y confidencialidad de datos y transporte, basándose en gran medida, en las capacidades de seguridad del propio lenguaje Java ya descritas con anterioridad.

Hablar de la seguridad en cada una de las capas que componen la plataforma J2EE lleva cientos de páginas más que por cuestiones de tiempo y espacio no se tocaran a fondo en este trabajo, sin embargo, es importante mencionar algunas de las características más importantes y algunas de las recomendaciones propuestas para cada una de las capas.

En el caso de J2EE, el framework de seguridad generalmente es declarativa, es decir, que nosotros solo indicamos el nivel de seguridad que deseamos que tenga un recurso en particular, y el encargado de proporcionar tal característica es el contenedor o servidor de aplicaciones. Esta declaración de seguridad se realiza en un archivo en particular y generalmente en formato XML, sin embargo, si se requiere de un control de acceso más complicado o detallado, siempre es posible recurrir al estilo programable.

La mayoría de los servidores de aplicación J2EE proporcionan los siguientes servicios de seguridad:

1. Mediante los dominios de seguridad o ‘realms’ es posible crear grupos lógicos de usuarios, grupos y listas de control de acceso que permiten proteger los recursos del servidor.
2. Mecanismos de autenticación que permiten identificar al usuario que solicita acceso a los recursos administrados por el servidor de aplicaciones J2EE.
3. Autorización de usuarios y grupos a través de las listas de control de acceso, lo que permite el aseguramiento de las políticas de seguridad y restricciones de acceso a recursos y usuarios específicos.
4. Integridad y confidencialidad de datos a través del uso de protocolos de comunicación segura como SSL/TLS.

5. Auditoría y registro de eventos de seguridad importantes como son los intentos de logging fallidos, certificados digitales rechazados, intentos de acceso no autorizados, entre otros.
6. Soporte de métodos de autenticación y autorización basados en el framework modular JAAS.
7. Soporte de servicios de seguridad de terceros mediante agentes proveedores de seguridad modulares.
8. Implementación de la arquitectura de seguridad extensible de Java, como son JSSE, JCE.
9. Soporte de acceso 'single sign-on' en todas las aplicaciones J2EE dentro de un mismo dominio de seguridad.

En la Figura 44 se muestran los descriptores de implementación utilizados para los distintos componentes J2EE.

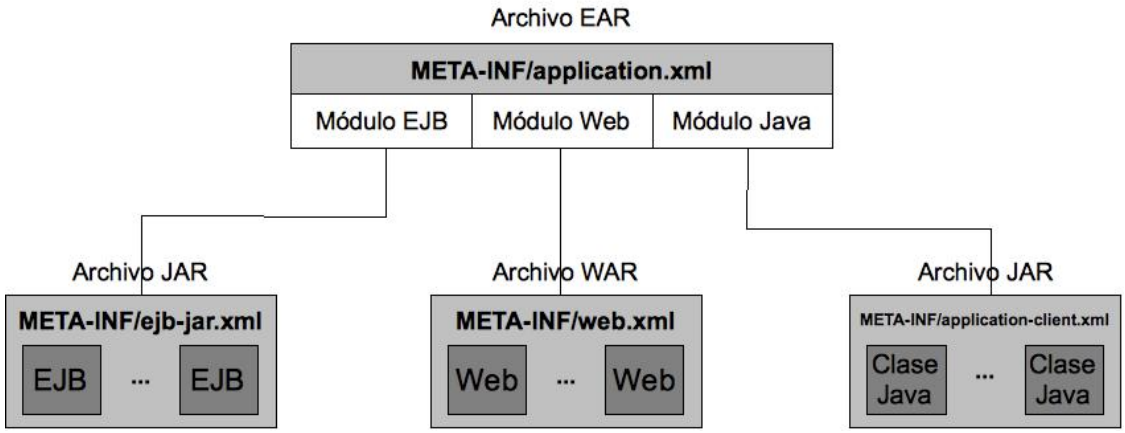


Figura 44: Descriptores de implementación J2EE.

4.15.3 Seguridad en la capa de transporte

La plataforma J2EE facilita la comunicación segura adoptando mecanismo de confidencialidad e integridad en la capa de transporte basados en los protocolos SSL/TLS. Las propiedades de seguridad de estos protocolos se configuran a nivel del contenedor de aplicaciones. La seguridad de la comunicación basada en los protocolos SSL/TLS se puede especificar para los componentes web o para los componentes EJB. Generalmente es la responsabilidad del ensamblador de aplicaciones identificar los componentes con llamadas a métodos cuyos parámetros o valores de retorno deben ser confidenciales e íntegros. Se utiliza el archivo conocido como descriptor de implementación de componente para representar esta información.

4.15.4 Seguridad en la capa web o capa de presentación.

La capa web representa los componentes de presentación, los cuales interactúan entre los clientes web y la lógica de negocio de la aplicación. En esta capa se incluyen los archivos JSP's, Servlets, applets Java, archivos HTML estáticos, así como clases de soporte asociadas a la aplicación web. Un archivo de aplicación web se empaqueta como archivo *war* y contiene todos los archivos 'class' y recursos adicionales de la aplicación, así como el archivo descriptor XML que contiene información de configuración de la aplicación.

El contenedor web proporciona los mecanismos de seguridad para garantizar la seguridad y el control de acceso de los componentes web. Dentro de las características de seguridad están la autenticación, autorización, mecanismos de login de usuario, manejo seguro de sesiones, seguridad en la capa de transporte y acceso a servicios 'single sign-on'.

Mecanismos de Autenticación disponibles para la capa web:

- **Autenticación básica HTTP:** Utiliza un par de campos usuario/contraseña. En este caso, el contenedor de aplicación solicita al navegador web del usuario que capture estos datos a través de una ventana propia del navegador que envía estos datos de regreso al contenedor de la aplicación. En este caso, se configura un elemento *<login-config>* en el archivo web.xml del componente web, incluyendo los subelementos *auth-method* y *realm-name*.

  Ejemplo:

```
<web-app>
...
  <security-constraint> ...</security-constraint>
<login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>miDominio</realm-name>
</login-config>
...
</web-app>
```

- **Autenticación basada en formas:** Utiliza una página web personalizada para la captura de los campos de usuario/contraseña. También permite la configuración de una página de error personalizada. Al igual que la autenticación básica, utiliza un elemento *<login-config>* en el archivo web.xml del componente web, así como los subelementos *<form-login-config>*, *<form-login-page>* y *<form-error-page>* como se muestra en el fragmento de código que sigue.

```
<web-app>
...
<security-constraint> ...</security-constraint>
<login-config>
        <auth-method>FORM</auth-method>
        <realm-name>miDominio</realm-name>
        <form-login-config>
            <form-login-page>login.jsp</form-login-page>
            <form-error-page>login.jsp</form-error-page>
        <form-login-config>
```

Este mecanismo de autenticación se puede utilizar en los clientes GUI incluyendo los applets AWT y Swing, sólo que estos applets deben utilizar componentes que proporcionen este tipo de autenticación.

Al igual que la autenticación básica, la basada en formas utiliza la codificación BASE64 para la transmisión de los campos de usuario/contraseña, lo cual se considera inseguro, ya que no es como tal un algoritmo de cifrado. Es por esto, que la mejor opción es utilizar este mecanismo sobre una capa SSL segura. Esta configuración también se especifica en el descriptor de implantación del componente web de la siguiente forma:

```
<web-app>
...
<security-constraint>
<user-data-constraint>
        <description>
        Transmisión segura utilizando SSL
        </description>
        <transport-guarantee>
            CONFIDENTIAL
        </transport-guarantee>
<user-data-constraint>
</security-constraint>
<login-config>
        <auth-method>FORM</auth-method>
        <realm-name>miDominio</realm-name>
        <form-login-config>
            <form-login-page>login.jsp</form-login-page>
            <form-error-page>login.jsp</form-error-page>
        <form-login-config>
</login-config>
```

- **Autenticación mutua o certificado de cliente:** En este tipo de autenticación el contenedor web acepta solicitudes HTTP utilizando el protocolo SSL en ambas direcciones y utilizando el certificado de llave pública del cliente emitido por una Autoridad Certificadora confiable. Para configurarlo se utiliza el mismo elemento <login-config> del archivo web.xml de la siguiente manera.

```
<web-app>
...
<security-constraint> ...</security-constraint>
<login-config>
        <auth-method>CLIENT_CERT</auth-method>
        <realm-name>miDominio</realm-name>
</login-config>
...
</web-app>
```

- **Autenticación DIGEST:** Es un mecanismo similar a la autenticación básica, con la diferencia de que en vez de utilizar la codificación BASE64, se utiliza una función hash para la transmisión de la contraseña. A pesar de eso, no se considera una autenticación segura ya que la transmisión no es cifrada. Se configura mediante el mismo elemento *<login-config>* en el archivo web.xml como se muestra a continuación.

```
<web-app>
...
<security-constraint> ...</security-constraint>
<login-config>
        <auth-method>DIGEST</auth-method>
        <realm-name>miDominio</realm-name>
</login-config>
...
```

- **Autenticación usando JAAS:** La mayoría de los contenedores J2EE y proveedores de contenedores implementan el soporte de mecanismos de autenticación y autorización basados en JAAS. Para esto, es necesario instalar y configurar los módulos de login JAAS. Estos módulos generalmente se configuran como *realms* o dominios. En estos casos, el encargado del procedimiento de autenticación es el propio módulo, no el contenedor.
- **Autenticación Single Sign-On:** Se basa en gran medida en las características del modelo JAAS, generalmente proporcionando una pila de módulos de autenticación. Su configuración depende del contenedor de aplicaciones y del conjunto de módulos JAAS.
- **Autenticación basada en agentes:** Es un proceso de autenticación implementada por un módulo externo al contenedor de aplicaciones.

Mecanismos de Autorización disponibles para la capa web.

El contenedor web adopta un mecanismo de autorización basado en roles para restringir el acceso a componentes web y recursos asociados. La autorización en la capa web sigue los mecanismos de seguridad J2EE utilizando los modelos de seguridad declarativa y programable.

- **Autorización declarativa:** Se configura en el archivo web.xml utilizando el elemento <*security-constraint*> así como el subelemento <*auth-constraint*> definiendo los roles autorizados para acceder a los recursos protegidos.

  Es importante mencionar que esta restricción de seguridad solo aplica a la solicitud web original y no en llamadas hechas desde los métodos de servicio de un *RequestDispatcher*, como las realizadas por las directivas <*jsp:include*> y <*jsp:forward*>. Esto significa que se debe tener especial precaución en las llamadas 'forward' hacia recursos protegidos, a menos que se verifiquen los permisos de acceso.

- **Autorización programable:** Generalmente se utiliza este tipo de autorización cuando la autorización declarativa no es suficiente para expresar los requerimientos de seguridad de la aplicación web. Para esto se utilizan métodos del objeto *HttpServletRequest* como son *getRemoteUser*(), *isUserInRole*() y *getUserPrincipal*().

4.15.5 Seguridad en la capa de negocio o capa EJB.

Representa los componentes de negocio. Generalmente consisten en *beans* de sesión, *beans* de entidad o *beans* orientados al mensaje. Cada *bean* se empaqueta y se implementa como un archivo *"jar"* o "*ear*" junto con un archivo XML descriptor de implementación.

El modelo se seguridad de esta capa es un subconjunto de la seguridad de la plataforma J2EE y es muy similar a las estrategias seguidas por otros componentes J2EE, sin embargo, la especificación de EJB no define esquemas de autenticación soportados y deja la responsabilidad al contenedor J2EE para implementarlos. Las opciones de seguridad EJB se centran en el control de acceso y autorización. Al igual que en la capa web, el servidor J2EE adopta mecanismos de seguridad declarativa y programable para la capa de negocio.

- **Autorización declarativa:** En este caso, las reglas de control de acceso y las políticas de seguridad de un EJB se definen en un archivo descriptor del componente externo que se asocia al componente EJB. Permite declarar los roles de seguridad que se asocian con permisos en métodos del componente EJB. Este descriptor puede contener cero o más elementos *<method-permission>* definidos dentro de un elemento *<assembly-descriptor>* para proporcionar el mapeo entre los roles y el control de acceso a métodos. Un elemento *<method-permission>* puede contener uno o más elementos *<role-name>* y uno o más elementos *<method>.* Los elementos *<role-name>* contienen valores de nombres de roles definidos en los elementos *<role-name>* dentro de los elementos *<security-role>.* Los elementos *<method>* identifican los métodos EJB para los que es necesario definir permisos de control de acceso. Una descripción de autorización declarativa para un EJB sería como sigue:

```
<ejb-jar>
...
<assembly-descriptor>
...
<security-role>admin</security-role>
        <method-permission>
                <role-name>*</role-name>
                <method>
                        <ejb-name>CSPSecureService</ejb-name>
                        <method-name>create</method-name>
                </method>
        </method-permission>
</assembly-descriptor>
```

- **Autorización programable:** Permite la invocación dinámica de métodos de negocio basados en el rol de seguridad del cliente remoto. Este mecanismo permite realizar un control de acceso detallado utilizando reglas de acceso dinámicas que se utilizan en los casos donde la autorización declarativa no es suficiente para expresar los requerimientos de seguridad del componente EJB. Generalmente se hace mediante llamadas a métodos de la interface *EJBContext* como son *isCallerInRole* y *getCallerIdentity*,

En el contexto de seguridad del ambiente J2EE, es muy común que los componentes EJB se basen en los mecanismos de autenticación proporcionados por el contenedor web.

El contenedor web utiliza la interface de usuario para obtener las credenciales de seguridad y es responsabilidad del contenedor de aplicaciones mantener los límites de acceso a tales credenciales entre las distintas llamadas y componentes del contenedor, así como también de mantener el contexto de seguridad que encapsula toda la información de autenticación. Cuando un cliente web invoca un método EJB, el contenedor propaga el contexto se seguridad mediante métodos específicos ya mencionados.

4.15.6 Seguridad en Servicios Web.

Actualmente existen esfuerzos importantes para el desarrollo de estándares industriales para la seguridad de los servicios web principalmente basados en la tecnología XML. Los estándares de seguridad más destacados en este sentido son los siguientes:

- XML Encryption: Esta especificación proporciona las bases para el intercambio seguro de datos. Se encarga de proporcionar los servicios de confidencialidad de datos. De acuerdo a la especificación es posible cifrar/descifrar un documento xml a diferentes niveles, como puede ser a nivel de dato, elemento o documento.
- XML Signature: Esta especificación define las bases para el intercambio seguro de documentos XML. Su propósito es proporcionar los servicios de integridad de datos, autenticación de mensajes y no repudio. Se diseñó con la idea de tener la posibilidad de firmar cualquier tipo de contenido digital u objetos de datos
- WS-Security: Es un estándar desarrollado por OASIS para la seguridad de los servicios web. Nace debido a la necesidad de especificar mecanismos de seguridad en los mensajes SOAP incorporando un conjunto de extensiones SOAP estándares requeridos para la seguridad de los servicios web e implementar servicios de autenticación, integridad, confidencialidad de mensajes, así como propagación de tokens de seguridad. Así mismo define mecanismos para soportar diversos mecanismos de cifrado y firma digital, y estándares de seguridad como PKI y Kerberos.
- XML Key Management System (XKMS): Esta especificación define un protocolo basado en XML utilizado en la distribución y registro de llaves criptográficas basadas en una infraestructura de llave pública (PKI) para su uso en los servicios web.

Capítulo 5: Ejemplo de un desarrollo con UMLSEC y JAVA.

Resumen: En este capítulo se presentan fragmentos de código aplicados al desarrollo de una aplicación siguiendo las recomendaciones descritas en los capítulos anteriores, desde el uso de diagramas UML utilizando el profile UMLSec, su asociación al momento de implementarlo en código Java y algunos fragmentos de código utilizando el API de seguridad y criptografía de Java. Se plantea el desarrollo de una aplicación siguiendo el proceso unificado de desarrollo, por lo que el capítulo se divide en las distintas etapas que conforman el proceso. Para cada etapa, se describen las actividades de seguridad recomendadas, los artefactos creados y en el caso de los diagramas UML, se hace uso del profile *UMLSec* para resaltar requerimientos de seguridad al momento de la implementación. Para la etapa de implementación, se plantean algunas de las prácticas de programación segura recomendadas utilizando el lenguaje Java.

5.1 Descripción del sistema.

Como parte de un programa de reducción de costos de una institución gubernamental, se presenta el proyecto de sustitución de tecnología de software licenciada por alternativas de código abierto. En la oficina de administración de proyectos se hace uso de software Microsoft Project Server del lado de servidor y Microsoft Project Professional del lado del cliente. Como alternativa a esta tecnología se propone la sustitución de funcionalidad con software de código abierto que en su caso es el sistema llamado ProjectOpen[37] para el lado del servidor y GanttProject[38] del lado del cliente.

No es objetivo de este trabajo comparar aspectos de seguridad de la tecnología licenciada contra la tecnología de código abierto, ni de las ventajas del uso de código abierto sobre código licenciado o viceversa, sólo se hace referencia a este proyecto para ejemplificar el proceso de desarrollo utilizando algunas de las recomendaciones de un desarrollo de software seguro.

[37] http://www.project-open.com/
[38] http://www.ganttproject.biz/

ProjectOpen es un sistema de administración de proyectos open-source de acceso web, desarrollado sobre un framework conocido como OpenACS[39]. Esta desarrollado en el lenguaje Tcl[40] y se ejecuta sobre un servidor web Aolserver[41]. Por su parte, GanttProject es una herramienta de escritorio para la administración de proyectos desarrollada con el lenguaje de programación Java.

En las secciones siguientes se hará referencia a este proyecto para ejemplificar aspectos de un desarrollo seguro utilizando los diagramas UMLSec que ayudarán a describir características generales del sistema, y en la parte de implementación se hace uso de fragmentos de código del GanttProject para presentar algunas de las prácticas recomendadas de codificación utilizando el lenguaje de programación Java. . El proyecto consiste en analizar, diseñar e implementar la funcionalidad necesaria, no proporcionada por las versiones de software open-source disponibles, de tal forma que pueda considerarse una solución viable a la sustitución de software licenciado en uso.

5.2 Análisis de Riesgos.

Cualquier intento de proporcionar seguridad en un sistema se vuelve inconcebible si no existe un análisis de riesgos aceptable. El objetivo de un análisis de riesgos es identificar los activos de una organización, sus metas, así como comprender las amenazas de seguridad y sus vulnerabilidades. Inicialmente está información no es producto de un análisis de software o de requerimientos funcionales de un sistema en particular, si no del análisis de la información organizacional que en su momento se reflejará en el análisis de requerimientos de seguridad de los sistemas de software. Se recomienda realizar este análisis de manera periódica y requiere de un monitoreo constante.

En el desarrollo ejemplo que se describe en este capítulo, no se contempla este análisis ya que no es parte de la información disponible para su desarrollo, sin embargo, se menciona debido a su importancia, ya que determina gran parte de los parámetros de seguridad de los sistemas.

[39] http://www.openacs.org/
[40] http://www.tcl.tk/
[41] http://www.aolserver.com/

5.3 Análisis de Requerimientos.

Éste análisis ya se realiza sobre los requerimientos de un sistema a implementar, donde se incluyen requerimientos funcionales y no funcionales del sistema. De la misma manera que un análisis de requerimientos funcionales nace de un modelo de dominio y/o negocio, así como de entrevistas con los interesados en el sistema, los requerimientos de seguridad parten del análisis de riesgos descrito anteriormente, pero modelados en el contexto del sistema actual y considerando la tecnología y arquitectura a utilizarse.

A continuación pondremos en contexto algunos pasos propuestos por el método SQUARE para la ingeniería de requerimientos y se mencionarán los elementos del profile UMLSec que apliquen para el análisis de requerimientos. Desafortunadamente, debido a que se trata de un método alterno al utilizado en el desarrollo real de la aplicación, carecemos de información suficiente que permita realizar un análisis más a fondo.

De acuerdo a lo citado por el método SQUARE, uno de los elementos más importantes en la captura de requerimientos está en saber definir los requerimientos de seguridad. Establecer como requisito de seguridad que el canal de comunicación entre el cliente y el server debe ser cifrado, o decir que se utilizará una infraestructura de llave pública para establecer relaciones de confianza entre cliente y servidor, no son propiamente requerimientos de seguridad, si no, mecanismos de seguridad. Generalmente los requerimientos de seguridad se deben ligar a los servicios de seguridad mencionados en capítulos anteriores, por lo que para nuestro ejemplo, algunos de los requerimientos de seguridad válidos e importantes pueden ser los siguientes:

Requerimientos de Seguridad:

1. Mantener la confidencialidad de información de recursos humanos disponibles en el servidor de proyectos.
2. El proceso de autenticación para el cliente y servidor se requiere que sea sencillo y seguro.
3. Garantizar el no repudio en la creación de líneas base[42] de un cronograma.
4. Garantizar la integridad de los datos construidos por el cliente, como son cronograma, línea base y asignación de recursos.

[42] En el área de administración de proyectos, la línea base se define como el estado de un cronograma de actividades en un tiempo dado y sirve para comparar el avance real alcanzado de acuerdo a lo planeado.

Una vez establecidos los requerimientos funcionales y de seguridad, el siguiente paso es la generación de artefactos o documentación necesaria. Esta actividad es una de las más importantes del desarrollo de sistemas y como se menciona en el proceso SQUARE, la creación de documentación para caracterizar al atacante como un actor del sistema es una de etapas más importantes del proceso. Como se define en el proceso unificado, los artefactos de modelado utilizados en esta etapa son los diagramas de casos de uso, glosario de términos y prototipo de interfaz de usuario, adicionalmente, y como parte del proceso SQUARE se recomienda la creación de diagramas de casos de abuso, así como árboles de ataque. Como se mencionó en capítulos anteriores, en esta etapa se recomienda utilizar los patrones de ataque para identificar y modelar los diagramas de casos de abuso.

Siguiendo con el análisis de nuestro desarrollo, partimos de un patrón de ataque para la creación de diagramas de casos de abuso.

Manipulación de elementos de datos del cliente.	
Identificador del patrón de ataque:	39
Severidad Típica:	Media
Descripción:	En circunstancias donde una aplicación almacena información importante (cookies, URLs, archivos de datos) y tales datos son manipulables. Si tales componentes son reinterpretados por el cliente o por el servidor como componentes de autenticación o datos y el atacante puede manipularlos de alguna forma.
Pre-requisitos del Ataque:	El atacante dispone de acceso al sistema o puede acceder a datos del cliente a través de otro usuario que tiene acceso al sistema. Para que un atacante pueda ejecutar este ataque con éxito, algún tipo de dato se almacena en el cliente y puede manipularse sin ningún mecanismo de detección.
Probabilidad para explotar el ataque:	Muy alta.
Métodos de ataque:	Modificación de recursos.
Habilidades requeridas por el atacante:	Medio: Si los datos del cliente son ofuscados, Alto: Si los datos del cliente son cifrados.
Recursos necesarios:	El atacante no requiere de recursos de hardware especiales para realizar este ataque.
Técnicas probadas:	Generalmente es mediante prueba y error.
Soluciones y Mitigaciones:	Proteger los datos del cliente con servicios de confidencialidad e integridad. Asegurarse que todos los datos de las sesiones utilizan una buena fuente de aleatoriedad. Realizar validaciones por parte del cliente para asegurarse que los valores enviados por el cliente son consistentes con lo que se espera.
Requerimientos de seguridad relevantes:	Verificación de integridad de información sensible en el cliente antes de utilizarse.

El patrón de ataque anterior muestra un escenario de abuso probable aplicado a nuestro sistema ejemplo y ayudará en gran medida en las etapas de análisis posteriores. En la figura siguiente se muestra un ejemplo de un diagrama de caso de abuso basado en el patrón de ataque mostrado en la tabla anterior.

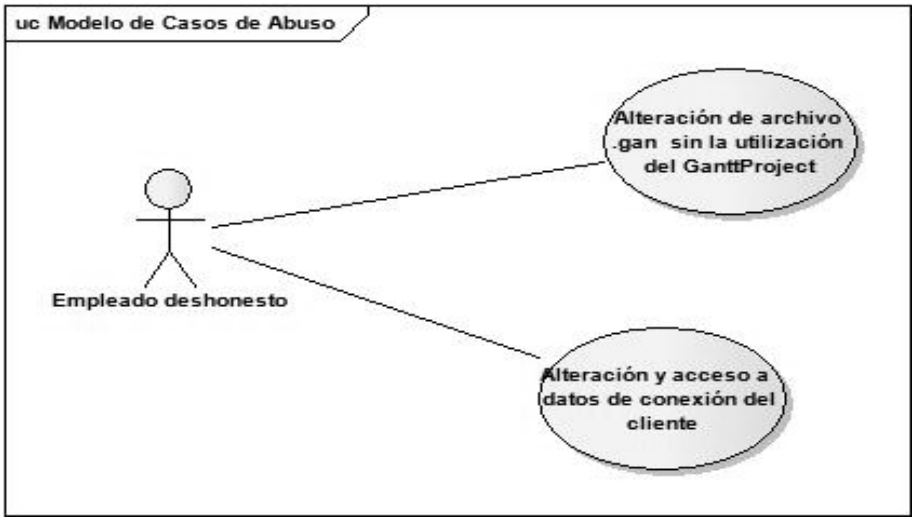


Figura 1: Ejemplo de un diagrama de caso de abuso.

Una vez identificado gran parte de los requerimientos de seguridad del sistema se recomienda realizar un análisis de priorización de requerimientos. Esto se debe a que cada requerimiento de seguridad conlleva a un costo en el proyecto y generalmente existen restricciones de presupuesto para su realización. Ya se han mencionado algunos de los métodos de priorización de requerimientos conocidos y la selección del método depende de diversos factores, aunque por lo general, el método utilizado es decisión de los 'stakeholders'. En nuestro desarrollo ejemplo, no se muestra esta etapa del priorización, sin embargo es importante considerarla debido a su trascendencia.

Para el término de esta etapa los artefactos creados serían los siguientes:

1. Lista de requerimientos funcionales y no-funcionales.
2. Lista de requerimientos de seguridad descritos y priorizados.
3. Modelo de casos de uso: Incluye diagramas de casos de uso de los requerimientos funcionales del sistema, así como sus especificaciones.
4. Modelo de casos de abuso: Incluye diagramas de casos de abuso basados en patrones de ataque, así como árboles de ataque.
5. Interfaz de usuario prototipo.

5.4 Análisis y Diseño del sistema.

En esta etapa del ciclo de desarrollo se realizan las mayores decisiones sobre la arquitectura de un sistema y también es donde se desarrollan la mayor cantidad de los artefactos de modelado. Se hace un uso extensivo de los diagramas UML y es en esta etapa donde se recomienda realizar gran parte de los análisis de seguridad, ya que los defectos hallados en esta etapa, contribuirán a una reducción de costos importante. Es este caso, es en esta etapa donde se mostrará la mayor cantidad de elementos del profile UMLSec descrito en este trabajo.

Como en nuestro desarrollo ejemplo, gran cantidad de desarrollos no se realizan desde cero, si no que son adaptaciones, mejoras o, sustitución de tecnología, por lo que los sistemas con los que se interactúa se consideran actores y son identificados en la etapa anterior. En nuestro ejemplo, uno de los requerimientos de funcionalidad y seguridad es mantener la autenticación actual. Debido a que la red de la institución es una red Windows, utilizan el Directorio Activo de Windows como servicio de autenticación. En este caso, el directorio activo es considerado un actor del sistema y se convierte en una pieza trascendental en el mismo, ya que se encarga de proporcionar el mecanismo de autenticación. En este sentido, debido a que el alcance del desarrollo no incluye un análisis de seguridad en la configuración del Directorio Activo, sólo se asume que está hecha con las recomendaciones de seguridad adecuadas. Obviamente, una configuración de Directorio Activo segura depende de una instalación del sistema operativo adecuada, por lo que los sistemas host donde se instalará el sistema se convierten en dependencias externas importantes para la seguridad integral del sistema a desarrollar. Por lo tanto, en este desarrollo, las dependencias externas a considerar son las siguientes:

1. Sistema operativo Windows donde reside el directorio activo que ofrece la autenticación del sistema.
2. Configuración adecuada del directorio activo de Windows.
3. Sistema operativo Unix para la instalación del sistema ProjectOpen instalado y configurado correctamente.
4. ProjectOpen puede trabajar sobre instancias de bases de datos PostgreSQL y Oracle por lo que la instalación y configuración de tales servicios también determinará el nivel de seguridad del sistema en desarrollo.

En el desarrollo que se describe, el análisis se debe realizar en dos sentidos, uno que consiste en la comprensión de las reglas de negocio utilizadas en la actualidad y sus posibles mejoras, y por otro lado, la comprensión de los sistemas de código abierto que se utilizarán como alternativas.

El sistema mencionado funciona como una arquitectura cliente-servidor, y se pretende utilizar dentro de la red local de la organización. Debido a que el proyecto consiste en una sustitución de tecnología, la arquitectura del mismo se definió al momento de elegir los sistemas alternos. En este caso, la arquitectura del sistema se describe mediante el siguiente diagrama de implementación.

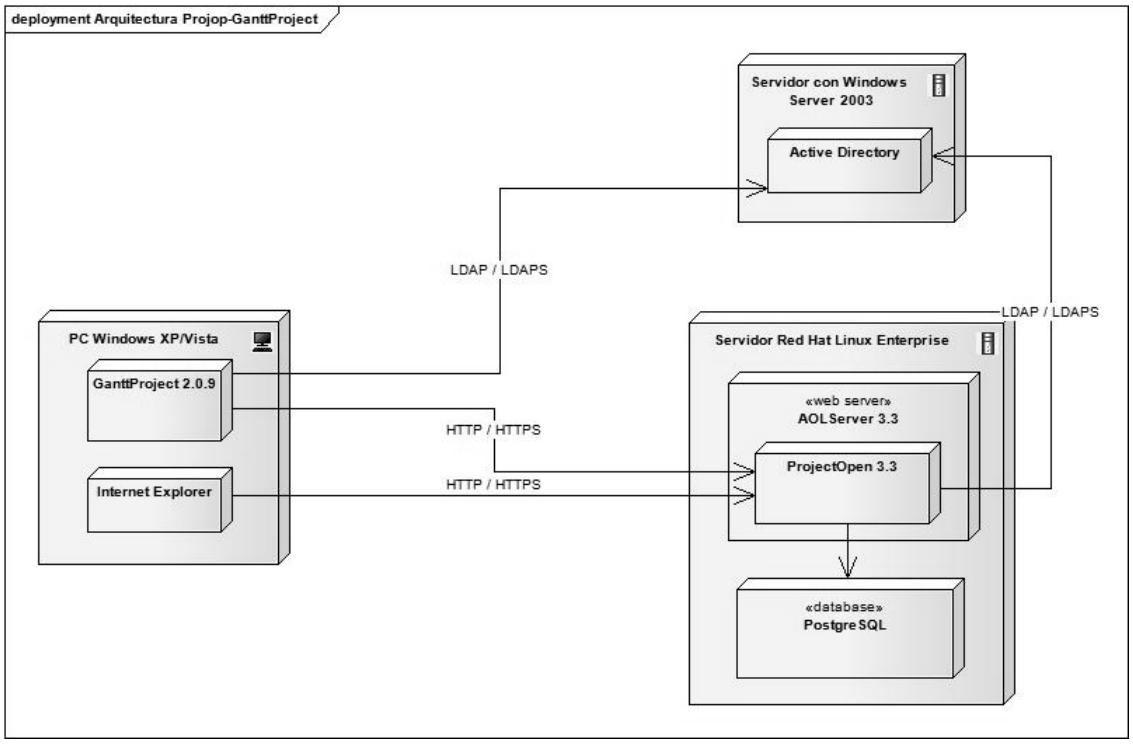


Figura 2: Arquitectura General del Proyecto de Integración Project Open - GanttProject.

A partir del conocimiento de la arquitectura general del sistema y de un análisis de dependencias externas, es posible realizar un análisis de amenazas como el que se sugiere a continuación. Este análisis toma como base el método descrito en el libro "*The Security Development Lifecycle Process* (Howard & Lipner, 2006)", también conocido como método STRIDE. Éste es un esquema de clasificación de caracterización de amenazas conocidas de acuerdo al tipo de '*exploits*' requerido y utiliza diagramas de flujos de datos (DFD) para analizar todos los elementos que afectan un sistema, para después clasificarlos de acuerdo al tipo de amenaza al que pueden ser sujetos. El método STRIDE deriva su nombre de las iniciales de cada una de las posibles amenazas que afectarían algún servicio de seguridad, cuya relación se muestra en la tabla siguiente (Shostack, 2007):

Tabla 1: Relación de Amenazas STRIDE con las propiedades de seguridad.

Propiedad	Amenaza	Definición
Autenticación	**S**poofing	Hacerse pasar por alguien o por algo.
Integridad	**T**ampering	Modificación de datos o código.
No-repudio	**R**epudiation	Negar la realización de una acción.
Confidencialidad	**I**nformation disclosure	Exposición de información a personas no autorizadas.
Disponibilidad	**D**enial of Service	Denegación o degradación de servicios de usuarios.
Autorización	**E**levation of Privilege	Ganar capacidades sin una propia autorización.

El método de amenazas STRIDE comienza con un diagrama de flujos de nivel 0 como el que se muestra en la Figura 3. Este diagrama solo muestra el sistema como proceso central y los actores principales del mismo.

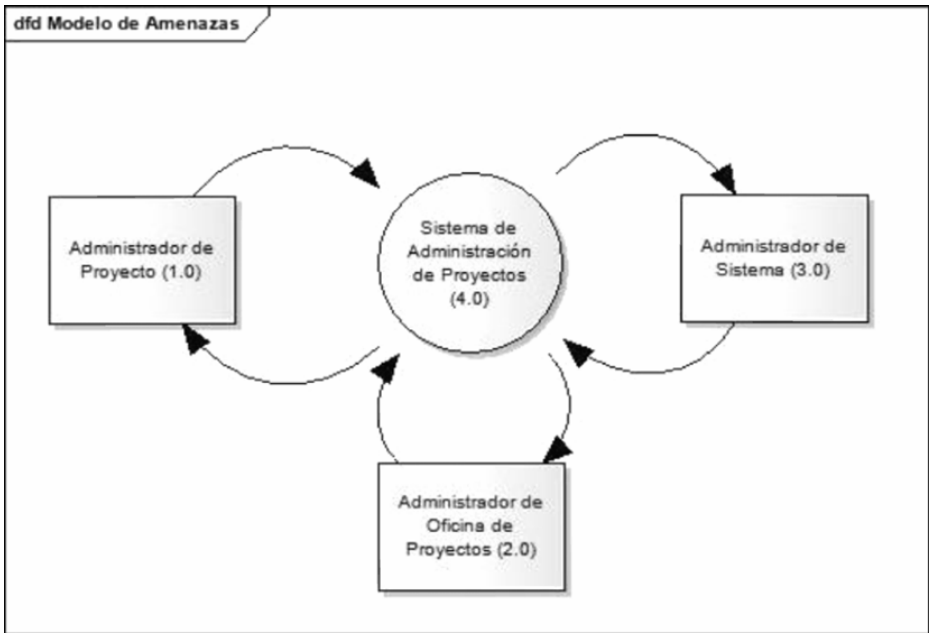


Figura 3: Diagrama de Flujo de Datos de Nivel 0.

Después del desarrollo del diagrama de flujo de datos de nivel 0, se procede con la realización de diagramas de flujo de nivel 1, 2, y así sucesivamente hasta tener una idea clara del sistema en desarrollo. En la Figura 4 se muestra el diagrama de flujo de datos del sistema de nivel 1, donde el sistema se ha dividido en dos, que son el sistema cliente y el sistema servidor, además de identificar las unidades de almacenamiento de datos, como son archivos de configuración, archivos de intercambio de datos y bases de datos principales.

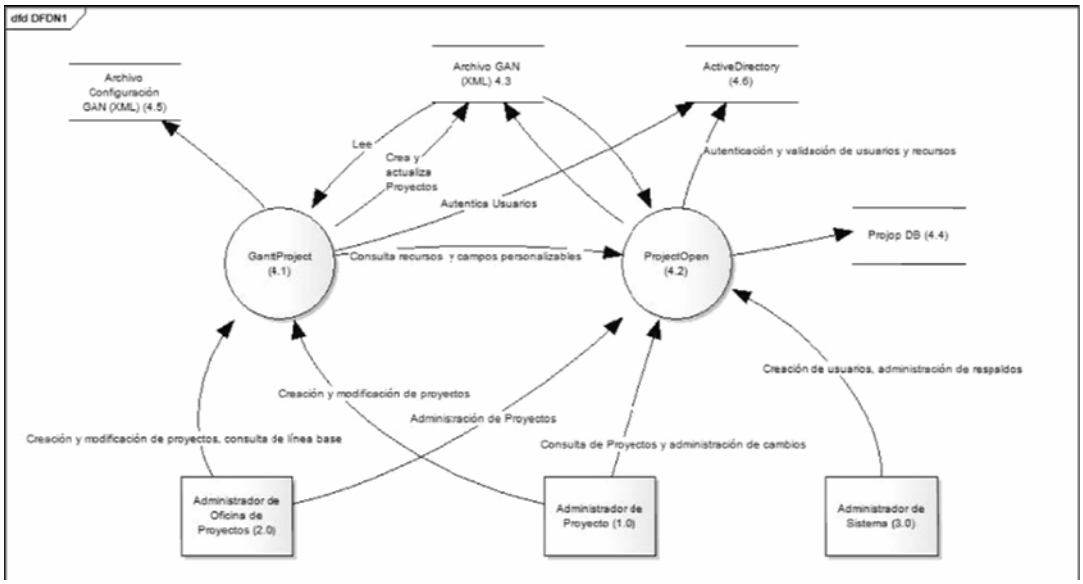


Figura 4: Diagrama de Flujo de Datos de Nivel 1

Siguiendo con el análisis, en la Figura 5 se muestra el diagrama de flujo de datos de nivel 2 para el sistema cliente GanttProject (4.1)

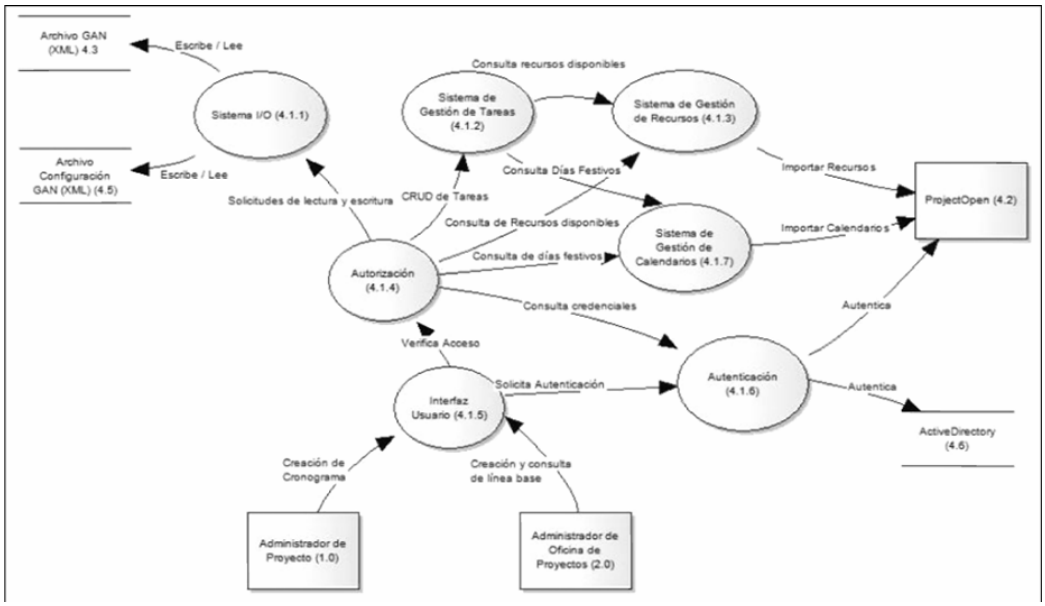


Figura 5: Diagrama de Flujo de Datos de Nivel 2 para el sistema cliente GanttProject (4.1)

Una vez obtenido un modelo del sistema de un nivel aceptable, el método STRIDE identifica los tipos de amenazas de acuerdo al tipo de elemento del diagrama de flujo de datos, siguiendo la siguiente tabla de relación.

Tabla 2: Mapeo de amenazas STRIDE con los tipos de elementos de un diagrama de flujo de datos.

Elemento DFD	Spoofing.	Tampering.	Repudiation.	Information Disclosure.	Denial of Service.	Elevation of Privilege.
Flujo de Datos.		x		x	X	
Almacenes de datos.		x	x	x	x	
Procesos.	x	x	x	x	x	x
Entidades externas.	x		X			

Aplicando esta relación de amenazas a los tipos de elementos de DFD de nivel 2, se genera la siguiente tabla de amenazas al sistema.

Tabla 3: Amenazas del sistema modelado.

Tipo de Amenaza (STRIDE)	Números de Elementos del DFD.
Spoofing	***Entidades Externas:*** (1.0), (2.0), (4.2) ***Procesos:*** (4.1.1), (4.1.2), (4.1.3), (4.1.4), (4.1.5), (4.1.6), (4.1.7)
Tampering	***Procesos:*** (4.1.1), (4.1.2), (4.1.3), (4.1.4), (4.1.5), (4.1.6), (4.1.7) ***Almacenes de datos****:* (4.3), (4.5), (4.6) ***Flujos de datos:*** (1.0→4.1.5), (2.0→4.1.5), (4.1.5→4.1.4), (4.1.5→4.1.6), (4.1.4→4.1.1), (4.1.4→4.1.2), (4.1.4→4.1.3), (4.1.4→4.1.7), (4.1.4→4.1.6), (4.1.1→4.5), (4.1.1→4.3), (4.1.2→4.1.3), (4.1.2→4.1.7), (4.1.3→4.2), (4.1.7→4.2), (4.1.6→4.2), (4.1.6→4.6).
Repudiation	***Entidades Externas:*** (1.0), (2.0), (4.2) ***Almacenes de datos****:* (4.3) ***Procesos:*** (4.1.1), (4.1.2), (4.1.3), (4.1.4), (4.1.5), (4.1.6), (4.1.7).
Information Disclosure.	***Procesos:*** (4.1.1), (4.1.2), (4.1.3), (4.1.4), (4.1.5), (4.1.6), (4.1.7). ***Almacenes de datos:*** (4.3), (4.5), (4.6) ***Flujos de datos:*** (1.0→4.1.5), (2.0→4.1.5), (4.1.5→4.1.4), (4.1.5→4.1.6), (4.1.4→4.1.1), (4.1.4→4.1.2), (4.1.4→4.1.3), (4.1.4→4.1.7), (4.1.4→4.1.6), (4.1.1→4.5), (4.1.1→4.3), (4.1.2→4.1.3), (4.1.2→4.1.7), (4.1.3→4.2), (4.1.7→4.2), (4.1.6→4.2), (4.1.6→4.6).
Denial of Service.	***Procesos:*** (4.1.1), (4.1.2), (4.1.3), (4.1.4), (4.1.5), (4.1.6), (4.1.7). ***Almacenes de datos:*** (4.3), (4.5), (4.6) ***Flujos de datos:*** (1.0→4.1.5), (2.0→4.1.5), (4.1.5→4.1.4), (4.1.5→4.1.6), (4.1.4→4.1.1), (4.1.4→4.1.2), (4.1.4→4.1.3), (4.1.4→4.1.7), (4.1.4→4.1.6), (4.1.1→4.5), (4.1.1→4.3), (4.1.2→4.1.3), (4.1.2→4.1.7), (4.1.3→4.2), (4.1.7→4.2), (4.1.6→4.2), (4.1.6→4.6).
Elevation of Privilege.	***Procesos:*** (4.1.1), (4.1.2), (4.1.3), (4.1.4), (4.1.5), (4.1.6), (4.1.7).

Una vez obtenida una lista de las amenazas del sistema modelado, el paso siguiente es determinar el riesgo del sistema. Existen gran cantidad de métodos de análisis de riesgos, sin embargo no es el objetivo de este trabajo describir los métodos conocidos. En este sentido, el método de definición de riesgo utilizados en conjunto con el método STRIDE son el DREAD, que considera el daño potencial de una amenaza (***D***), su reproducibilidad (***R***), explotabilidad (***E***), usuarios afectados (***A***) y la facilidad de descubrirlos (***D***).

Una vez identificados los riesgos, se proponen medidas de defensa o de mitigación considerando algunas de las siguientes estrategias:

1. No hacer nada: Solo aplicables para riesgos de nivel bajo.
2. Eliminar la característica: Es la única forma de reducir el riesgo a cero. La decisión de eliminar una característica solo es el resultado de un balanceo entre las características de usuario y los riesgos de seguridad potencial.
3. Apagar la característica: Es una solución menos drástica que la anterior y solo se utiliza cuando el riesgo se va a reducir tiempo después.
4. Advertir al usuario: Solo se aplica a amenazas muy específicas
5. Contrarrestar la amenaza con tecnología: Esta es la estrategia de mitigación más común.

Para nuestro sistema ejemplo vamos a mitigar con tecnología los elementos {Archivo GAN, Almacén de datos, 4.3}, {Sistemas I/O, Proceso, 4.1.1} y {Autenticación, Proceso, 4.1.6} del sistema Ganttproject.

En esta etapa ejemplificamos el uso del profile UMLSec para el modelado de sistemas, así como la implementación utilizando el lenguaje de programación Java. A continuación se presentarán algunos modelos que ayuden a comprender el uso del profile y en la sección siguiente se mostrará el código Java que lo implementa.

Elemento "Archivo GAN (4.3) y Sistemas I/O (4.1.1)"

Tomando en cuenta el requerimiento de seguridad "*Garantizar la integridad de los datos construidos por el cliente, como son cronograma, línea base y asignación de recursos*", así como el patrón de ataque "*Manipulación de elementos de datos del cliente*" presentamos un modelo para el subsistema de creación del archivo "gan" del cliente Ganttproject. Este modelo deberá contemplar servicios de integridad y confidencialidad de los datos descritos en los requerimientos de seguridad y del modelo de amenazas.

En el diagrama de clases de la Figura 6 se muestran los objetos involucrados para la creación de un archivo “GAN”. Este archivo esta en formato XML y contiene todos los datos de un proyecto como son las propiedades del cronograma, propiedades de los recursos, asignaciones de recursos a tareas, días no laborables y propiedades de despliegue propias de un proyecto Ganttproject. Debido a la naturaleza de un archivo XML este puede estar sujeto a distintos tipos de ataques además del ya mencionado, por lo que se debe garantizar que las modificaciones realizadas al mismo se realicen de manera segura, ya que este archivo se utiliza para actualizar la información almacenada en el servidor de administración de proyectos. En este caso partimos del modelo de clases disponible en la aplicación original descargada del sitio del proyecto y realizamos las modificaciones necesarias para agregar los requerimientos de seguridad. El diagrama de clases siguiente muestra el diseño original del subsistema de almacenamiento del archivo Gantt.

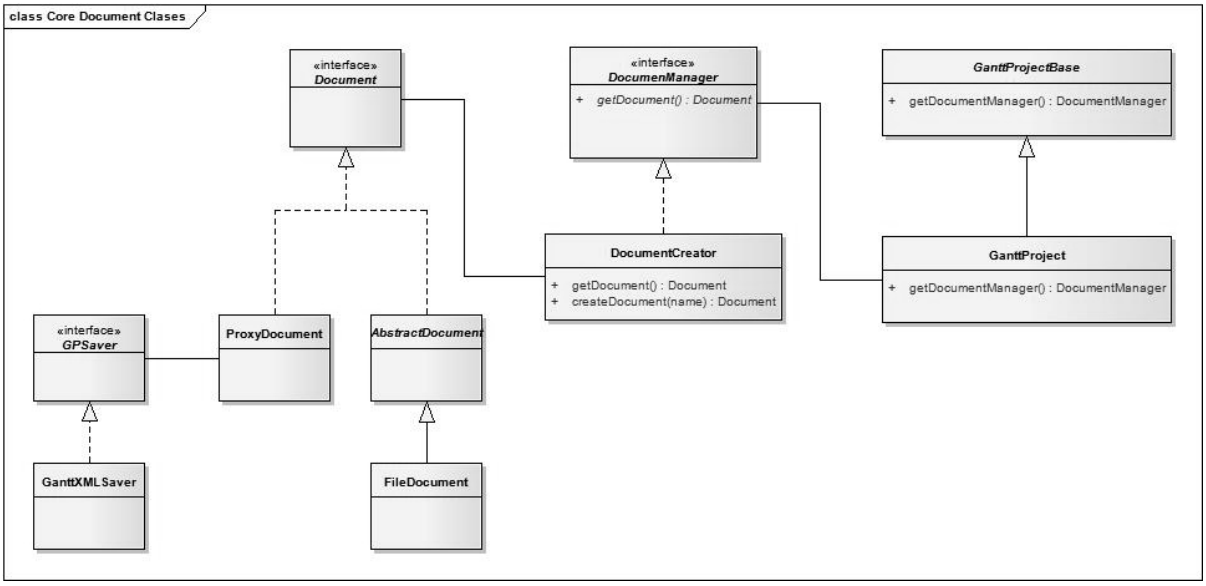


Figura 6: Sistema para el almacenamiento de archivos GAN.

Este modelo oculta detalles de atributos y métodos declarados para cada clase, sin embargo, una de las vulnerabilidades más importantes consideradas es la ausencia de un mecanismo de integridad de datos que garantice que el archivo es generado por el sistema cliente adecuado y por la persona debida. En el siguiente modelo, se muestra el mismo sistema agregando los estereotipos UMLSec necesarios y modificando el diseño para agregar las características de seguridad mencionadas. (Wutka, 1996)

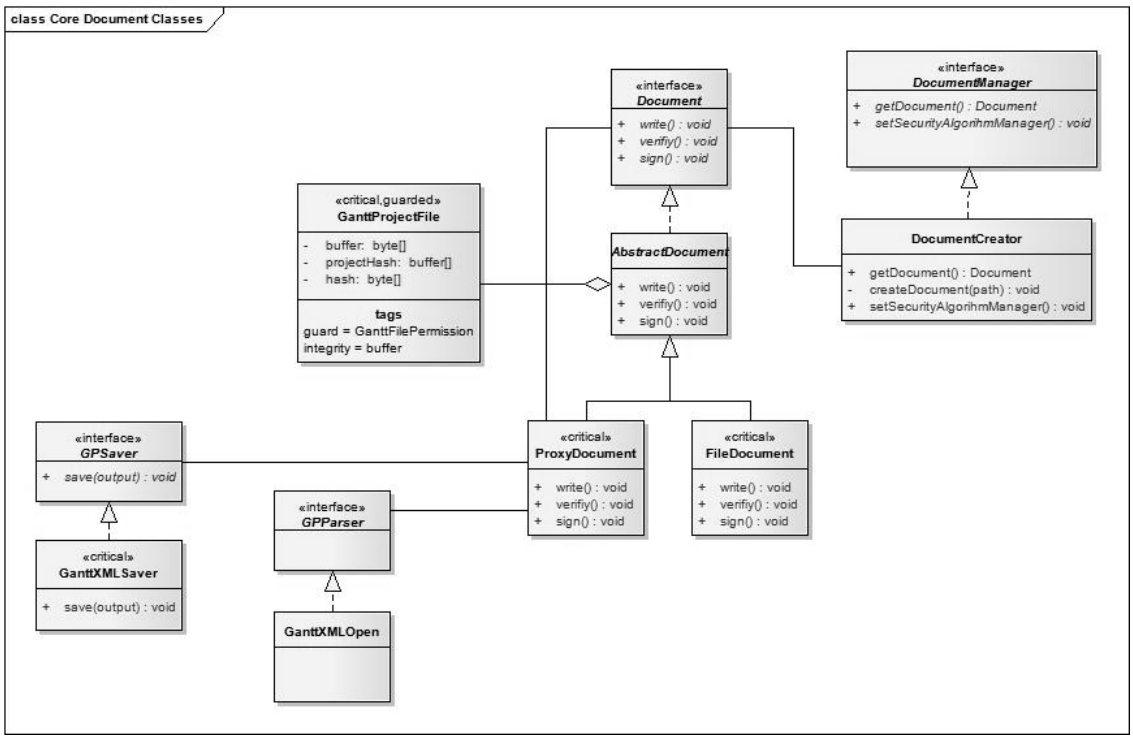


Figura 7: Sistema para el almacenamiento de archivos GAN con UMLSec

Es importante destacar el uso de los estereotipos y valores etiquetados utilizados en el diagrama anterior, ya que permiten identificar las clases críticas del sistema, así como características de seguridad y codificación importantes a nivel de clase, métodos y atributos. La relación de cada elemento del profile UMLSec con su implementación se describe con más detalle en la sección siguiente, sin embargo, el uso de un profile UML que describa características de seguridad en el modelado de sistemas, permite mantener la consistencia de los requerimientos de seguridad en las etapas de análisis, implementación y pruebas, disminuyendo en cierta medida el problema de refinamiento de sistemas descrito anteriormente.

Elemento "Autenticación (4.1.6)"

El sistema original no contiene ningún módulo de autenticación, por lo tanto es necesario adaptar tal subsistema. Como parte de los requerimientos funcionales y de seguridad, está el uso del directorio activo de Windows, así como la base de datos del sistema de Projop, como sistemas de autenticación de usuarios dentro del sistema. En este sentido un mecanismo de autenticación seguro debe contemplar distintas políticas de gestión de contraseña como son longitud, periodo de validez, uso de combinación de caracteres alfanuméricos y especiales en la contraseña, módulos de recuperación de contraseñas, entre otras características que para nuestro sistema forman parte

de las funciones de las dependencias externas del sistema. Con esto, queda claro la necesidad de un análisis de seguridad a nivel organizacional y que un sistema no puede estar seguro si se encuentra dentro de un ambiente o ecosistema inseguro. Para éste sistema se opta por el desarrollo de módulos de autenticación basados en el framework JAAS, ya que proporciona patrones de diseño adecuados para tal funcionalidad. El modelo UML del subsistema de autenticación propuesto queda de la siguiente manera:

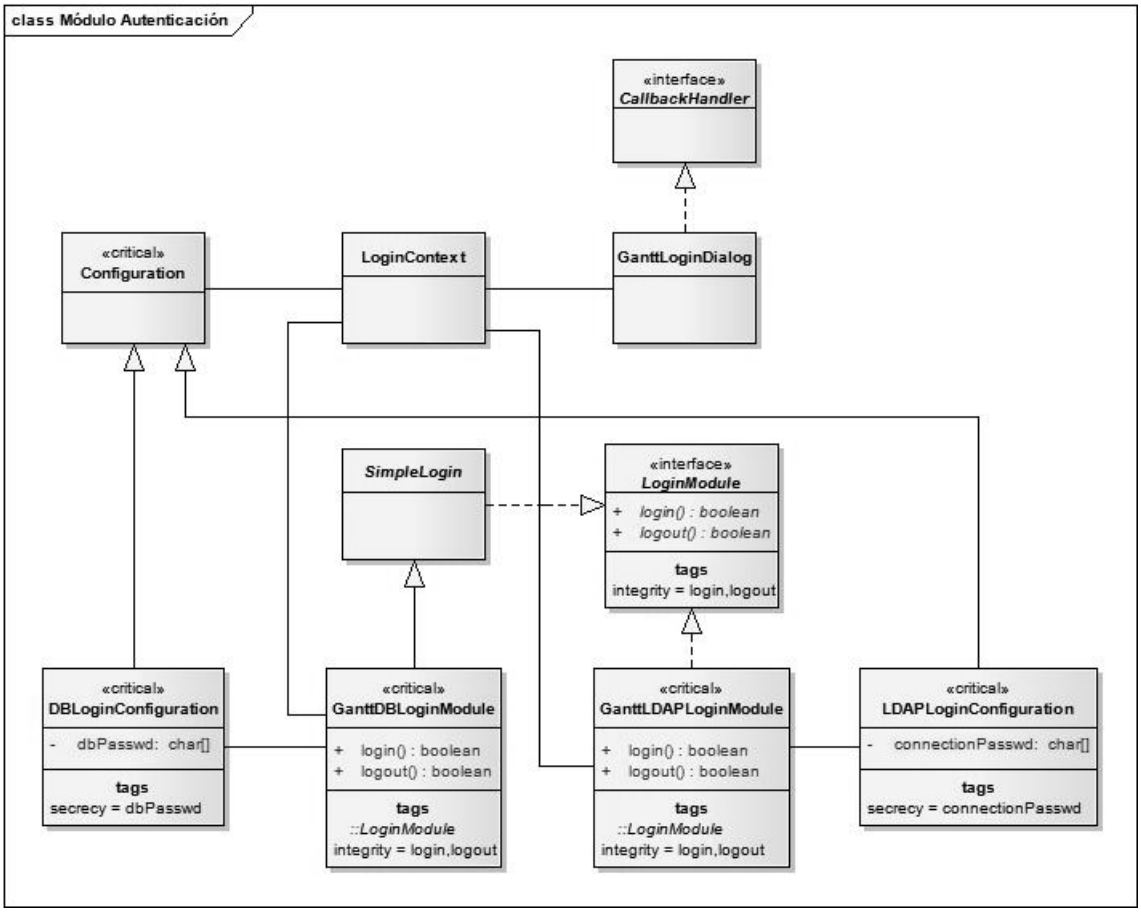


Figura 8: Modelo de Autenticación con UMLSec

El modelo anterior se basa completamente en el framework de JAAS, implementando los módulos adecuados para proporcionar la autenticación con el Directorio Activo (*GanttLDAPLoginModule*) y con la base de datos (*GanttDBLoginModule*). El modelo muestra las clases críticas del subsistema y los métodos y atributos que deben cumplir con los requisitos de confidencialidad (*dbPasswd*, *connectionPasswd*) e integridad. Este sistema tiene gran dependencia con sistemas externos, por lo que en la figura siguiente se muestra el modelo de despliegue del subsistema mencionado.

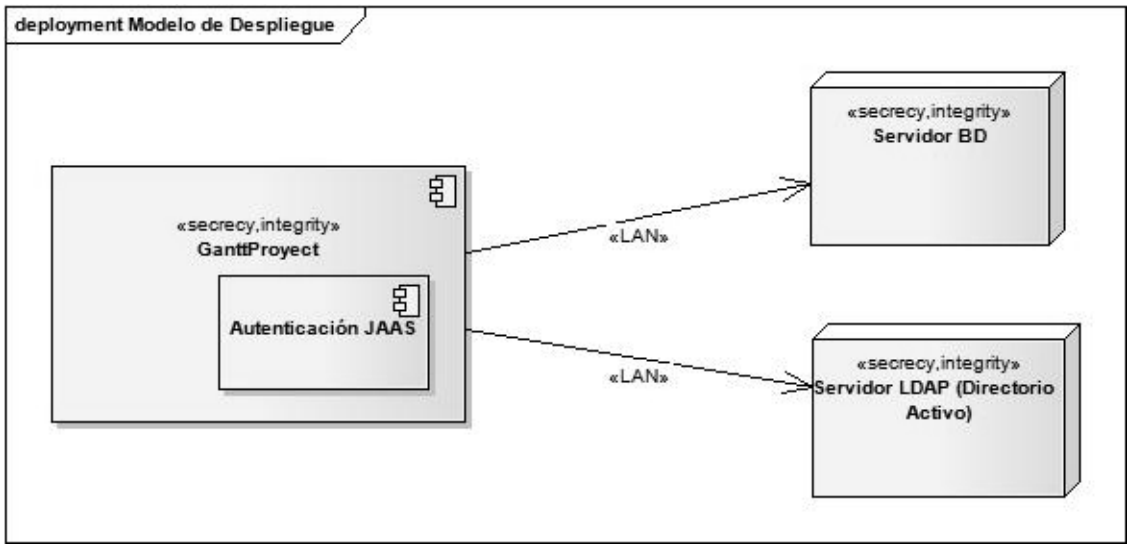


Figura 9: Modelo de despliegue. Sistema de Autenticación con UMLSec.

Los elementos UMLSec del diagrama anterior revelan información importante de las características del sistema. Primero, el estereotipo "*LAN*" especifica que el sistema funcionará dentro de la red local de la organización, de esta manera, determinamos el tipo de adversario al que estará expuesto el sistema. Los estereotipos *"secrecy"* e *"integrity"* indican que los tres sistemas son responsables de la integridad y confidencialidad de los datos transmitidos por el enlace, en específico, de los parámetros de autenticación de usuario y de origen de datos como son el usuario, contraseña, así como los certificados digitales de los servicios involucrados. En este sentido, el sistema es visto como un ecosistema en donde la seguridad en conjunto depende de la seguridad de cada una de las partes individuales.

Es importante resaltar que los estereotipos de UMLSec no sólo sirven para indicar los elementos críticos del modelo de un sistema en las etapas de análisis, diseño e implementación, si no que determinan el tipo de pruebas a realizar y sobre que componentes del sistema, como pueden ser en análisis de caja blanca y enfocar las pruebas de penetración.

Una vez realizado un análisis y diseño del sistema tomando en cuenta sus requerimientos funcionales y de seguridad, el siguiente paso es la implementación o codificación

5.5 Implementación.

Elemento "Archivo GAN (4.3) y Sistemas I/O (4.1.1)"

Así como la etapa de análisis y diseño, la etapa de implementación es una de las más delicadas en el proceso de desarrollo de software, ya que en esta etapa existe una gran probabilidad de que se inserten vulnerabilidades al sistema, por lo que se requiere de un monitoreo constante a lo largo del proceso de desarrollo. En este sentido, el uso del profile UMLSec funciona como lenguaje de comunicación de los requerimientos de seguridad entre los desarrolladores del sistema, sin embargo, también se requiere de un conocimiento considerable del lenguaje y plataforma de desarrollo para poder reflejar de manera adecuada los requerimientos de seguridad establecidos. En la actualidad existen distintas guías y recomendaciones de seguridad para distintos lenguajes, pero desgraciadamente no las suficientes para todos los lenguajes de programación y las plataformas de desarrollo. En este sentido, el lenguaje de programación Java es uno de los lenguajes con más guías y recomendaciones de seguridad. Para nuestro ejemplo, tomaremos el modelo propuesto y especificaremos algunas de las recomendaciones más importantes al momento de implementar el sistema. Para esta etapa se toman como fuente distintas recomendaciones de sitios encontrados en Internet siendo la más importante la publicada por el CERT en el sitio www.securecoding.cert.org llamada "The CERT Sun Microsystems Secure Coding Standard for Java".

De acuerdo con esta guía, una de las reglas más importantes para el desarrollo en Java es la utilización de la clase *SecurityManager*. Sin el uso de esta clase, muchas de las características de seguridad del lenguaje simplemente no se aplican. En este sentido, gran parte de la comunidad de desarrolladores creen que el uso de un lenguaje de programación "seguro" como Java garantiza un desarrollo "seguro", sin embargo, esto no aplica, especialmente si no se utiliza un objeto *SecurityManager*. Mediante su utilización, garantizamos el uso de una política de seguridad, al menos una política por default. Esta política de seguridad establece el conjunto de permisos que se aplican a la máquina virtual en el momento de ejecución de nuestra aplicación. La manera más fácil de habilitar esta opción en una aplicación Java es agregando los parámetros *–Djava.security.manager* y *–Djava.security.policy=={archivo_de_politicas}* a la línea de comandos de nuestra aplicación o proporcionando implementaciones propias de las clases *SecurityManager* y *Policy*. Una vez agregado estas opciones a la línea de comandos de la aplicación Java se han habilitado características de seguridad importantes del lenguaje, sin embargo, su uso no garantiza el desarrollo de una aplicación segura, sino que solo establece los límites de una aplicación dentro de la máquina virtual, evitando que una aplicación Java vulnerable acceda más allá de los límites y acciones establecidas.

Enfocándonos en el modelo del sistema realizado con anterioridad (Figura 7), para reflejar los requerimientos de seguridad establecidos hacemos uso de propiedades del lenguaje Java. Todas aquellas clases identificadas con el estereotipo "*critical*" son clases candidatas al uso del modificador de clase "*final*", de esta manera podemos garantizar que la funcionalidad de esa clase no pueda modificarse a través de una implementación de la misma y alterarse a través de mecanismo de herencia, por lo que para nuestro modelo, las clases *ProxyDocument*, *FileDocument*, *GanttXMLSaver*, *GanttProjectFile* y *DocumentCreator*, al tener el estereotipo "*critical*" se recomiendan sean "*final*" en la declaración de la clase. Siguiendo con los modificadores de acceso de una clase Java y de acuerdo a la recomendación *"SEC07-J Minimize accesibility"* del estándar del CERT es importante comprender y utilizar adecuadamente los modificadores de acceso propios del lenguaje. Muchos de estos modificadores de acceso se establecen al momento del análisis y diseño del sistema y se deben respetar en la implementación siguiendo las siguientes recomendaciones:

1. Un paquete Java además de proporcionar una estructura de nombres, ayudan a definir límites de acceso y seguridad a nivel de clases y métodos.
2. Una interface Java se utiliza para definir funcionalidad y es muy utilizada como alternativa a la herencia múltiple. Una interface siempre definirá métodos públicos por lo que si queremos que una clase mantenga un método privado o protegido, no es buena idea definirlo en una interface.
3. Una clase puede tener dos tipos modificadores de acceso que son el acceso por default (sin modificador *public*) y el acceso público (*public*). Una clase con acceso por default solo es visible dentro del paquete donde se declara mientras que una clase con acceso *public* es visible desde cualquier paquete.
4. Los atributos de una clase pueden tener uno de cuatros modificadores de acceso que son '*private'*, *default*, '*protected'*, y '*public'*. Por definición un atributo de clase debería tener modificador de acceso '*private'*, especialmente si es una clase con estereotipo "*critical*". Si por alguna razón se requiere que clases heredadas accedan al atributo de una manera especial, se debe definir el tipo con el acceso más limitado posible de acuerdo a su función, ya sea acceso por *default* o *protected* aunque en sentido estricto, estos usos no se recomiendan (*OBJ00-J Declare data members private*). El modificador de acceso *public* solo se justifica para la declaración de constantes en cuyo caso deben tener los modificadores de acceso *static* y *final (OBJ31-J. Do not use public static non-final variables)* y en su caso definirse en las interfaces.

Las reglas anteriores se aplican en el diseño e implementación de todo el sistema, no sólo para aquellas clases declaradas con el estereotipo "*critical*". Para las clases con estereotipo "*critical*" y etiqueta "*integrity*" es importante seguir las recomendaciones "*OBJ30-J Identify and handle immutable objects appropriately*", "*OBJ37-J Do not return references to prívate data*", "*OBJ36-J Provide mutable clases with a clone method*" especialmente para los atributos y métodos definidos en los valores de la etiqueta "*integrity*". De acuerdo con estas recomendaciones la implementación de nuestro modelo se muestra en las figuras siguientes. Solo se muestran fragmentos de código de las características del modelo que a nuestra consideración son las más relevantes.

Comenzamos describiendo un fragmento de la implementación de la clase abstracta *AbstractDocument*, donde los puntos más relevantes son el uso cuidadoso de los modificadores de acceso y el modificador *final*. Por otro lado está el uso de un objeto guardián para la protección del objeto *GanttProjectFile* que en nuestro diseño es el encargado de almacenar la información sensible de un archivo o proyecto Gantt, como es el contenido y su hash para proteger la integridad de nuestro trabajo. Como se describió anteriormente, todo objeto guardián requiere de un objeto "permiso" para verificar si es posible o no acceder al objeto protegido. En este ejemplo, se utiliza el método *checkPermission* del objeto *AccessController* para asegurarse que independientemente de la existencia de un *SecurityManager*, la verificación de acceso se debe realizar.

```
package net.sourceforge.ganttproject.document;

import java.security.AccessController;
import java.security.GuardedObject;
import net.sourceforge.ganttproject.permission.GanttProjectBeanPermission;

public abstract class AbstractDocument implements Document {
   private final GanttProjectFile documentBean;
   private final GuardedObject documentBeanGuard;

   public AbstractDocument() {
      documentBean = new GanttProjectFile();
      documentBeanGuard = new GuardedObject(documentBean, new
GanttProjectBeanPermission(GanttProjectBeanPermission.ACCESS_GANTT_BEAN_ACTION));
   }

   public GanttProjectFile getProjectFileBean() {
      AccessController.checkPermission(new
GanttProjectBeanPermission(GanttProjectBeanPermission.READ_HASH_ACTION));
      return (GanttProjectFile)documentBeanGuard.getObject();
   }
```

En el caso de la clase *GanttProjectFile*, también se resalta el uso de los modificadores de acceso y del modificador final, al tratarse de un objeto marcado con el estereotipo "critical" en nuestro modelo UML. Al igual que en la clase anterior utilizamos el método *checkPermission* del objeto *AccessController* para verificar los permisos de acceso al contenido y hash de nuestro proyecto, aunque es posible utilizar el método proporcionado por el *SecurityManager* instalado. Por su parte, para conservar la inmutabilidad del objeto *GanttProjectFile*, después de verificar los accesos se regresa una copia de los atributos, no una referencia a ellos, ya que al tratarse de un arreglo de bytes, si se regresara la referencia sería posible modificar su contenido sin el control de la clase *GanttProjectFile*.

```
package net.sourceforge.ganttproject.document;

import java.security.AccessController;
import net.sourceforge.ganttproject.permission.GanttProjectBeanPermission;

public final class GanttProjectFile {
   private byte[] projectBuffer;
   private byte[] projectHash;

   public GanttProjectFile() { }

   public byte[] getProjectHash() {
      AccessController.checkPermission(new
GanttProjectBeanPermission(GanttProjectBeanPermission.READ_HASH_ACTION));
      return projectHash.clone();
   }

   public byte[] getProjectBuffer(){
      AccessController.checkPermission(new
GanttProjectBeanPermission(GanttProjectBeanPermission.READ_FILE_ACTION));
      return projectBuffer.clone();
   }
```

En las dos clases anteriores se ha verificado la existencia de los permisos adecuados antes de realizar ciertas funciones. A continuación se muestra un fragmento del código que implementa estos permisos. Debido a su función dentro de la plataforma de seguridad de Java, todo objeto *Permission* debe considerar un conjunto de características importantes al momento de implementarse. Dentro de estas características esta el uso del modificador *final* para la clase *Permission* que se implementa, así como la encapsulación de atributos que todo objeto debe tener. Un caso especial se aplica en la implementación de los métodos *implies* de las clases *GanttProjectBeanPermission* y *GanttProjectBeanPermissionCollection* donde es importante verificar que los objetos que se pasan como parámetros realmente pertenezcan al tipo especificado.

```
package net.sourceforge.ganttproject.permission;

import java.security.Permission;
import java.security.PermissionCollection;
import java.util.Enumeration;
import java.util.Vector;

public final class GanttProjectBeanPermissionCollection extends PermissionCollection{
   private Vector<Permission> allPermissions;

   @Override
   public void add(Permission permission) {
                  if(permission instanceof GanttProjectBeanPermission){
                        GanttProjectBeanPermission
newGanttBeanPermission=(GanttProjectBeanPermission) permission;
                        allPermissions.add(newGanttBeanPermission);
                  }
   }

   @Override
   public boolean implies(Permission permission) {
                  if(! (permission instanceof GanttProjectBeanPermission))
                        return false;

                  GanttProjectBeanPermission
newGanttProjectBeanPermission=(GanttProjectBeanPermission)permission;

                  Enumeration<Permission> allPermissionsEnumeration=allPermissions.elements();
                  while(allPermissionsEnumeration.hasMoreElements()){
                        if(!(allPermissionsEnumeration.nextElement() instanceof
GanttProjectBeanPermission))
                              return false;
                        GanttProjectBeanPermission
ganttPermissionBean=(GanttProjectBeanPermission)allPermissionsEnumeration.nextElement();
                        if(ganttPermissionBean.getMaskActions() ==
newGanttProjectBeanPermission.getMaskActions()){
                              return true;
                        }
                  }
                  return false;
   }

   @Override
   public Enumeration<Permission> elements() {
                  return (Enumeration<Permission>)allPermissions.elements();
```

```
package net.sourceforge.ganttproject.permission;

import java.security.BasicPermission;
import java.security.Permission;
import java.security.PermissionCollection;

public final class GanttProjectBeanPermission extends BasicPermission{
   public static final int READHASH=0x1;

   public static final String ACCESS_GANTT_BEAN_ACTION="accessDocumentBean";
   public static final String READ_HASH_ACTION="readHash";
   public static final String READ_FILE_ACTION="readBuffer";

   private int actionsMask;

   public GanttProjectBeanPermission(String nombre) {
      super(nombre);
   }

   @Override
   public boolean implies(Permission perm) {
      if(!(perm instanceof GanttProjectBeanPermission)){
         return false;
      }
      GanttProjectBeanPermission ganttProjectPermission=(GanttProjectBeanPermission)perm;

      if(ganttProjectPermission.getClass()!=this.getClass())
         return false;

      if(this.getName().equals(ganttProjectPermission.getName())){
         if(this.getMaskActions()==ganttProjectPermission.getMaskActions()){
            return true;
         }
      }
      return false;
   }

   public int getMaskActions(){
      return actionsMask;
   }

   @Override
   public PermissionCollection newPermissionCollection() {
```

En todos los fragmentos de código mostrados anteriormente se ha hecho énfasis en las características de accesibilidad e inmutabilidad de los objetos mediante el uso adecuado de los modificadores de acceso, del modificador *final* para clases y atributos, así como del diseño de clases pensado en sus características de inmutabilidad. En este sentido el lenguaje de programación Java cuenta con un API conocida como "*Reflection*" la cual es muy utilizada por los ambientes de desarrollo (IDE's) y debuggers, Dentro de sus funciones está la posibilidad de alterar los modificadores de acceso mencionados anteriormente y de esta manera acceder a valores que de otra forma serían imposible. De acuerdo con lo anterior y siguiendo la recomendación *"SEC32-J No otorgar ReflectPermission con la acción supressAccessChecks"* es importante vigilar que no existan los permisos *ReflectPermission* y *AllPermission* dentro de la política de seguridad instalada en nuestra aplicación.

A continuación se muestra la implementación de los módulos *login* descritos en el modelo de autenticación.

Elemento "Autenticación (4.1.6)"

El diagrama UML del sistema de autenticación mostrado en la sección anterior está basado completamente en el framework JAAS y como se describe en el modelo de la Figura 8, el sistema requiere de la implementación de dos módulos de autenticación. A continuación se muestran fragmento de la implementación de cada uno de estos módulos.

La clase que implementa el módulo *GanttDBLogin* hereda de la clase *SimpleLogin*, la cual implementa las funciones de *login()*, *logout()*, *commit()* y *abort()* necesarias por todo módulo login que utilice el framework JAAS. La clase *GanttDBLogin* es la encargada de inicializar el módulo login y de asegurarse que el usuario dado es válido dentro de la base de datos de ProjectOpen. Al igual que en los ejemplos anteriores, en la implementación resaltamos el uso de los modificadores de acceso *prívate* para los atributos de la clase, así como el modificador *final* para la declaración de la clase y de atributos importantes. En la implementación del método *valídateUser* resaltamos el uso de variables parametrizadas en la construcción de la consulta a la base de datos, ya que este es uno de los métodos de evitar ataques del tipo *SQLInjection*. Este método además de validar al usuario dentro de la base de datos extrae los grupos o roles definidos dentro de la aplicación servidor, creando por cada uno de estos un objeto *GanttDBGroupPrincipal* asociado al objeto *Subject*, el cual se utiliza para las operaciones de autorización. En el método *initialize* se muestra el uso de una función de seguridad que descifra y decodifica los valores de las variables almacenadas en el archivo de configuración de este módulo.

```
package net.sourceforge.ganttproject.auth;

import ...

public final class GanttDBLogin extends SimpleLogin {
   private String dbDriver;
   private String dbURL;
   private String dbUser;
   private char[] dbPassword;
   private final SecurityTools secTools;

   public GanttDBLogin() throws NoSuchAlgorithmException, NoSuchPaddingException {
      secTools = SecurityTools.getInstance();
   }

   protected synchronized Vector<Principal> validateUser(String username, String password) throws
LoginException {
      ResultSet resultSetUser = null;
      ResultSet resultSetRoles = null;
      Connection dbConnection = null;
      PreparedStatement preparedStatementUser = null;
      PreparedStatement preparedStatementRoles = null;

      try {
         Class.forName(dbDriver);
         if (dbUser != null) {
            dbConnection = DriverManager.getConnection(dbURL, dbUser, dbPassword.toString());
         } else {
            throw new LoginException(MensajesError.DB_LOGIN_ERROR);
         }

         preparedStatementUser = dbConnection.prepareStatement("SELECT user_id FROM PARTIES
WHERE username=?");

         preparedStatementUser.setString(1, username);
         resultSetUser = preparedStatementUser.executeQuery();
         if (!resultSetUser.next()) {
            throw new LoginException(MensajesError.DB_LOGIN_ERROR);
         }
         int uid = resultSetUser.getInt(1);
         Vector<Principal> dbLoginPrincipals = new Vector<Principal>();
         dbLoginPrincipals.add(new GanttDBUserPrincipal(username));

         preparedStatementRoles = dbConnection.prepareStatement("" +
               "SELECT GROUPS.group_name FROM GROUP_DISTINCT_MEMBER_MAP,GROUPS,IM_PROFILES "
+
               "WHERE GROUP_DISTINCT_MEMBER_MAP.member_id= ? " +
               "AND IM_PROFILES.profile_id=GROUP_DISTINCT_MEMBER_MAP.group_id " +
               "AND GROUP_DISTINCT_MEMBER_MAP.group_id=GROUPS.group_id");

         preparedStatementRoles.setInt(1, uid);
         resultSetRoles = preparedStatementRoles.executeQuery();
         while (resultSetRoles.next()) {
            dbLoginPrincipals.add(new GanttDBGroupPrincipal(resultSetRoles.getString(1)));
         }
```

```
        } finally {
            try {
                if (resultSetUser != null) {
                    resultSetUser.close();
                }
                if (resultSetRoles != null) {
                    resultSetRoles.close();
                }
                if (preparedStatementUser != null) {
                    preparedStatementUser.close();
                }
                if (preparedStatementRoles != null) {
                    preparedStatementRoles.close();
                }
                if (dbConnection != null) {
                    dbConnection.close();
                }
            } catch (SQLException sqle) {
                throw new LoginException(MensajesError.DB_LOGIN_ERROR);
            }
        }
    }

@Override
    public void initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map
options) {
        super.initialize(subject, callbackHandler, sharedState, options);

        try {
            dbDriver = getOption("dbDriver", null);
            if (dbDriver == null) {
                throw new Error(MensajesError.DB_LOGIN_ERROR);
            }
            dbURL = getEncryptedString(getOption("dbURL", null));
            if (dbURL == null) {
                throw new Error(MensajesError.DB_LOGIN_ERROR);
            }

            dbUser = getEncryptedString(getOption("dbUser", null));
            dbPassword = getEncryptedString(getOption("dbPassword", null)).toCharArray();

            if (dbUser == null || dbPassword == null) {
                throw new Error(MensajesError.DB_LOGIN_ERROR);
            }
        } catch (BadPaddingException bde) {
            throw new Error(MensajesError.DB_LOGIN_ERROR);
        } catch (IllegalBlockSizeException ibse) {
            throw new Error(MensajesError.DB_LOGIN_ERROR);
        } catch (InvalidKeyException ibse) {
            throw new Error(MensajesError.DB_LOGIN_ERROR);
        }

    }
```

A continuación se muestran fragmentos de código de la implementación del módulo *GanttLDAPLoginModule*. Como en todo código Java se resalta el uso de los modificadores de atributos y de clase, además del uso constante de arreglos *char* para almacenar la contraseña de usuario. La lógica de autenticación adecuada y un correcto manejo de las excepciones son puntos críticos en la implementación de un módulo login JAAS, adicionalmente a la limpieza de todos los objetos *Principal* agregados al objeto *Subject* utilizado en los mecanismos de autorización de JAAS.

```
package net.sourceforge.ganttproject.auth;

import ...

public final class GanttLDAPLoginModule implements LoginModule {
   private static final String INITIAL_CONTEXT_FACTORY = "initialContextFactory";
   private static final String CONNECTION_URL = "connectionURL";
   private static final String CONNECTION_USERNAME = "connectionUsername";
   private static final String CONNECTION_PASSWORD = "connectionPassword";
   private static final String CONNECTION_PROTOCOL = "connectionProtocol";
   private static final String AUTHENTICATION = "authentication";
   private static final String USER_BASE = "userBase";
   private static final String USER_SEARCH_MATCHING = "userSearchMatching";
   private static final String USER_SEARCH_SUBTREE = "userSearchSubtree";
   private static final String SECURITY_PROTOCOL = "securityProtocol";
   private DirContext context;
   private Subject loginSubject;
   private CallbackHandler handler;
   private LDAPLoginProperty[] config;
   private String username;
   private final SecurityTools secTools;

   public GanttLDAPLoginModule() throws NoSuchAlgorithmException, NoSuchPaddingException {
      secTools=SecurityTools.getInstance();
   }

public void initialize(Subject subject, CallbackHandler callbackHandler, Map<String, ?> sharedState,
Map<String, ?> options) {
      loginSubject = subject;
      handler = callbackHandler;
     /* Inicialización de variables de autenticación con el Directorio Activo. Se obtienen del archivo de
configuración, previo descifrado según sea el caso */
      try {
      config = new LDAPLoginProperty[]{
               new
LDAPLoginProperty(INITIAL_CONTEXT_FACTORY,getEncryptedString((String)options.get(INITIAL_CONTEXT_FACTORY))
),
               new LDAPLoginProperty(CONNECTION_URL,
getEncryptedString((String)options.get(CONNECTION_URL))),
               new LDAPLoginProperty(CONNECTION_USERNAME,
getEncryptedString((String)options.get(CONNECTION_USERNAME))),
               new LDAPLoginProperty(CONNECTION_PASSWORD,
getEncryptedString((String)options.get(CONNECTION_PASSWORD))),
               new LDAPLoginProperty(CONNECTION_PROTOCOL, (String)options.get(CONNECTION_PROTOCOL)),
               new LDAPLoginProperty(AUTHENTICATION, getEncryptedString((String)options.get(AUTHENTICATION)))
```

```
options.get(USER_SEARCH_MATCHING)),
                new LDAPLoginProperty(USER_SEARCH_SUBTREE, (String)
options.get(USER_SEARCH_SUBTREE)),
                new LDAPLoginProperty(SECURITY_PROTOCOL, (String) options.get(SECURITY_PROTOCOL)) };


/* Código para el manejo de las excepciones */
            ...
}

public boolean login() throws LoginException {
      Callback[] callbacks = new Callback[1];
      callbacks[0]=new GUIJAASCallback();

      try {
         handler.handle(callbacks);
      } catch (IOException ioe) {
         throw new LoginException(MensajesError.LOGIN_ERROR);
      } catch (UnsupportedCallbackException uce) {
         throw new LoginException(MensajesError.LOGIN_ERROR);
      }

      username = ((GUIJAASCallback)callbacks[0]).getUserName();
      if (username == null) {
         return false;
      }

      char[] password=((GUIJAASCallback)callbacks[1]).getPassword();
      if (password == null) {
         return false;
      }

      try {
         boolean result = authenticate(username, password);
         if (!result) {
            throw new FailedLoginException(MensajesError.LDAP_LOGIN_ERROR);
         } else {
            return true;
         }
      } catch (AuthenticationException fe) {
         throw new LoginException(MensajesError.LDAP_LOGIN_ERROR);
      } catch (CommunicationException ce) {
         throw new LoginException(MensajesError.LDAP_LOGIN_ERROR);
      } catch (NamingException ne) {
         throw new LoginException(MensajesError.LDAP_LOGIN_ERROR);
      }
   }

public boolean logout() throws LoginException {
      username = null;
      cleanSubject();
      return true;
   }
```

```
   protected boolean bindUser(DirContext context, String dn, char[] password) throws
NamingException {
      boolean isValid = false;

      context.addToEnvironment(Context.SECURITY_PRINCIPAL, dn);
      context.addToEnvironment(Context.SECURITY_CREDENTIALS, password);

      try {
         context.getAttributes("", null);
         isValid = true;
      } catch (AuthenticationException e) {
         isValid = false;
      }

      if (isLoginPropertySet(CONNECTION_USERNAME)) {
         context.addToEnvironment(Context.SECURITY_PRINCIPAL,
getLDAPPropertyValue(CONNECTION_USERNAME));
      } else {
         context.removeFromEnvironment(Context.SECURITY_PRINCIPAL);
      }

      if (isLoginPropertySet(CONNECTION_PASSWORD)) {
         context.addToEnvironment(Context.SECURITY_CREDENTIALS,
getLDAPPropertyValue(CONNECTION_PASSWORD));
      } else {
         context.removeFromEnvironment(Context.SECURITY_CREDENTIALS);
      }

      return isValid;
   }

   private void cleanSubject(){
      Set principals=loginSubject.getPrincipals();
      if(principals!=null){
         int principalsTotal=principals.size();
         for(Iterator<Principal> principalsIterator=principals.iterator();principalsIterator.hasNext();){
            Principal currentPrincipal=principalsIterator.next();
            if(currentPrincipal instanceof GanttLDAPPrincipal){
               principals.remove(currentPrincipal);
            }
```

Como se mencionó previamente, ambos módulos requieren de parámetros de configuración que se obtienen de un archivo el cual revela información sensible para cualquiera que tenga acceso al mismo. Es por esto que es importante proteger esta información y evitar un acceso indebido. A continuación se muestra el archivo de configuración JAAS de nuestro sistema ejemplo, en donde se muestra la información sensible cifrada.

```
PROJOPLogin {
        net.sourceforge.ganttproject.auth.GanttLDAPLoginModule required
        initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
        connectionURL="vLiE4uK1smNHHHtx9ScbmHOWYoDNYyGO8+M6LTDtgnRXigHO5SOMEyMN
        bKcludX+"
        connectionUsername="kvcczqeon1J2XvbUMtwR0SdLc+gtMMDv60krbjG+IlV3SaIK8WXae4NRv
        7nklWohmwv5A8wOEthLaXSHYpP+qmTWzHKICJVyIW1VUR2279w="
        connectionPassword="bwsk3xMSCrmS8Lwk89UA9Q=="
        connectionProtocol="ssl"
        authentication=simple
        userBase="x8CHaXp8hUvj8NClOVCOtJhwHQCU1D3Pvd8zE2vbR8Y="
        userSearchMatching="(sAMAccountName={0})"
        userSearchSubtree=true
        ;
        net.sourceforge.ganttproject.auth.GanttDBLogin required
        dbDriver="org.postgresql.Driver"
        dbURL="0Hsd6Io6ic9WTzZLc11/AQn8bD8/FvlbZScDS+NXJakL3hU82gts0ZJM7wrGHHQA"
        dbUser="juV4GxrdU19gqp+nPp8VCw=="
```

De esta manera, para invocar el proceso de autenticación el sistema cliente *GanttProject* debería realizarlo con un código parecido al siguiente.

```
    try {
       LoginContext loginContext = new LoginContext("PROJOPLogin",
GanttLoginCallbackHandler.getInstance());
       loginContext.login();
    } catch (LoginException loginException) {
       /* Manejo de Excepciones */
    }
```

Al momento de crear el objeto *LoginContext*, se manda a llamar los métodos *initialize* de cada módulo definido en el archivo de configuración, y al llamar el método *login* del objeto *LoginContext* comienza el proceso de autenticación de cada módulo definido.

En los sistemas complejos es complicado seguir adecuadamente todas las recomendaciones de seguridad proporcionadas por una guía de desarrollo y aunque definir estos requerimientos en un diagrama UML ayuda a poner particular atención a clases y características críticas de un sistema, es importante llevar a cabo una serie de pruebas automáticas y análisis de seguridad sobre un código implementado. En la siguiente sección se describe y ejemplifica esta etapa dentro del desarrollo de sistemas.

5.6 Pruebas.

Además de las pruebas de funcionalidad, las actividades requeridas por el proceso de desarrollo seguro definen procesos de pruebas de caja blanca y caja negra. Los artefactos de seguridad creados en las etapas iniciales del desarrollo como los diagramas de casos de mal uso y los diagramas UML son herramientas importantes para las etapas de pruebas, especialmente para las pruebas de caja blanca, también conocido como análisis estático. En esta etapa, el uso del *profile* UMLSec es una herramienta útil para enfocar las pruebas de seguridad y revisión de código necesarios en los subsistemas y clases asociadas.

Por otro lado, existen herramientas automáticas que se encargan de automatizar las pruebas de análisis estático, donde la más conocidas son las desarrolladas por las empresa Fortify[43] y OunceLabs[44], sin embargo, es importante desarrollar un plan de pruebas a lo largo de todo el desarrollo de software que considere la tecnología y el lenguaje de programación utilizados. De acuerdo a un documento publicado por (Vries, 2006), se define una taxonomía de pruebas automatizadas de software divididas de la siguiente manera:

1. Pruebas de unidad: Estas pruebas operan a nivel de método y de clase. Permiten detectar errores de funcionalidad a gran detalle y en etapas tempranas del proceso de desarrollo.
2. Pruebas de integración: En estas se verifica la integración entre las distintas clases, módulos y capas. En el caso particular de la plataforma J2EE, permite realizar pruebas en las capas de negocio y web.
3. Pruebas de aceptación. Estas pruebas generalmente se realizan por un equipo de QA y cuando se tratan de pruebas de seguridad se realizan por consultores de seguridad.

Para cada tipo de pruebas, existen herramientas de código libre que permiten automatizar la detección de vulnerabilidades a distintos niveles de la aplicación. A continuación se mencionan algunos ejemplos de herramientas Open Source para el análisis estático y de seguridad orientadas al lenguaje Java. En general no son herramientas orientadas únicamente al análisis de seguridad, pero se pueden utilizar para incluir pruebas de seguridad a distintos niveles.

[43] http://www.fortify.com/
[44] http://www.ouncelabs.com/

1. FindBugs[45]: Por sí misma, no es una herramienta creada para el análisis de vulnerabilidades, pero debido a las necesidades de seguridad, en versiones recientes ha incluido reglas para la detección de problemas comunes de seguridad. Esta herramienta es desarrollada por la Universidad de Maryland en conjunto con instituciones como Google, Sun, así como SureLogin[46], empresa dedicada a la creación de herramientas avanzadas para el desarrollo en Java. Las reglas definidas en la actualidad se clasifican en malas prácticas, corrección, vulnerabilidad de código malicioso, precisión en el manejo de hilos, desempeño y pruebas de seguridad básicas como SQLInjection y XSS.
2. Apache Cactus[47]: Es un framework de pruebas simples para código Java utilizados en los contenedores de aplicaciones. Como la gran mayoría de frameworks de pruebas, se basa en la infraestructura JUnit y permite realizar pruebas sobre componentes EJB y Servlets, además de incluir módulos para la realización de pruebas sobre tecnologías especificas como Struts y JSF.
3. JWebUnit[48]: Es un framework de pruebas basadas en Java orientada a aplicaciones web. Proporciona un API que permite navegar sobre una aplicación web combinada sobre un conjunto de afirmaciones para verificar que una aplicación funciona como se espera. Estas verificaciones pueden ser enlaces de navegación, llenado y envío de formas, validación del contenido de las tablas y otras características de las aplicaciones web.
4. WATIR[49] (Web Application Testing on Ruby): Biblioteca OpenSource para automatizar navegadores web e interactuar con un navegador web de la misma forma que lo haría una persona. Permite realizar clicks sobre los enlaces, llenar formas, presionar botones y verificar los resultados de tales acciones.
5. Selenium[50]: Conjunto de herramientas para automatizar pruebas en aplicaciones web. Permite crear pruebas a través de la interacción directa del usuario sobre un navegador y sobre una aplicación en particular. Crea código en diferentes lenguajes de programación para la realización de pruebas posteriores.

También se recomienda la realización de aplicaciones sencillas que permitan realizar verificaciones de seguridad particulares, como podría ser en los desarrollos basados en el framework Struts, donde se recomienda el análisis de los distintos archivos de configuración como son struts-config.xml y web.xml para verificar validación de formas, manejo de sesiones e interfaces de entrada a los sistemas, así como la configuración general del sistema.

[45] http://findbugs.sourceforge.com
[46] http://www.surelogic.com/
[47] http://jakarta.apache.org/cactus/
[48] http://jwebunit.sourceforge.net/
[49] http://wtr.rubyforge.org/index.html
[50] http://seleniumhq.org/

Estas herramientas son muy útiles para la detección rápida de problemas comunes de implementación, sin embargo no sustituyen un análisis detallado de la implementación para verificar que cumpla con los requerimientos de seguridad de los procesos de negocio o aquellos definidos en el diseño y análisis del sistema. En este sentido, el uso de un profile como UMLSec permite focalizar el análisis en aquellas clases o subsistemas que en etapas anteriores se definieron como críticas, además de verificar mediante herramientas diversas que tales requerimientos se cumplen en la implementación e incluso poder certificar aplicaciones de acuerdo a estándares específicos.

En este caso, ya que se trata de una aplicación de escritorio, utilizamos *FindBugs* para ejemplificar el uso de una herramienta para el análisis estático de código y verificar así el tipo de resultado que podemos esperar de este análisis. Al ejecutar esta herramienta se detectaron 19 errores clasificados en errores de corrección, experimentales, vulnerabilidad de código malicioso y dudgy. La figura siguiente muestra el error clasificado como vulnerabilidad de código malicioso, donde se exhibe la ausencia de inmutabilidad de atributos.

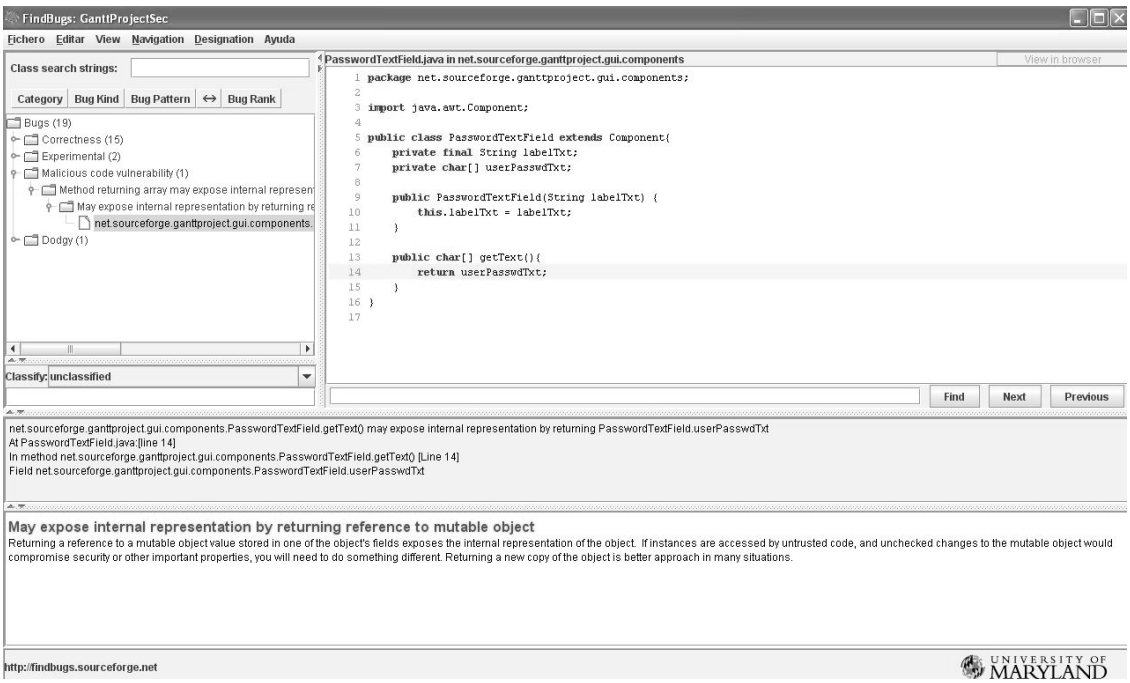


Figura 10: Salida de la ejecución de la herramienta de análisis estático FindBugs

El objetivo del análisis estático y de caja blanca es encontrar gran cantidad de errores de implementación y corregirlos antes de que salgan a producción, logrando tener un sistema de mejor calidad.

Los análisis de caja negra también conocidos como pruebas de penetración toman en cuenta la visión de un atacante externo que no conoce detalles de su implementación y que trata de vulnerarlo con el uso de pruebas específicas que le permitan revelar información sobre la forma en que se implementó el sistema. Estas pruebas son útiles especialmente para verificar que el ecosistema donde se implementa el sistema en desarrollo, cumple con las características de seguridad importantes, ya que se intenta atacar al sistema por su lado más débil, ya en un ambiente productivo.

5.7 Despliegue

Como parte de las etapas finales del proceso unificado y siguiendo las recomendaciones de seguridad sobre código Java esta la firma del código generado y la ofuscación del mismo para evitar la ingeniería inversa sobre los jars generados.

Antes de firmar el código, es importante ofuscarlo, ya que la firma digital de un *jar* se genera a partir del contenido de cada uno de los archivos *class*, y este se modifica con la ofuscación. Hay distintas herramientas OpenSource para la ofuscación de código Java. En este caso utilizamos ProGuard [51] para ofuscar el archivo jar generado.

El resultado de decompilar un archivo class previamente ofuscado se muestra en el siguiente fragmento de código.

```
package net.sourceforge.ganttproject.utils;

import java.io.PrintStream;
import javax.crypto.*;
import javax.crypto.spec.SecretKeySpec;

public class Base64Coder
{
    public Base64Coder()
  {
  }

    public static String a(String s)
  {
        return new String(a(s.getBytes()));
  }
```

[51] http://proguard.sourceforge.net/

```
public static char[] a(byte abyte0[])
  {
      int i = abyte0.length;
      abyte0 = abyte0;
      int j = ((i << 2) + 2) / 3;
      int k;
      char ac[] = new char[k = (i + 2) / 3 << 2];
      int l = 0;
      for(int i1 = 0; l < i; i1++) {
         int j1 = abyte0[l++] & 0xff;
         int k1 = l >= i ? 0 : abyte0[l++] & 0xff;
         int l1 = l >= i ? 0 : abyte0[l++] & 0xff;
         int i2 = j1 >>> 2;
         j1 = (j1 & 3) << 4 | k1 >>> 4;
         k1 = (k1 & 0xf) << 2 | l1 >>> 6;
         l1 &= 0x3f;
         ac[i1++] = a_char_array1d_static_fld[i2];
         ac[i1++] = a_char_array1d_static_fld[j1];
         ac[i1] = i1 >= j ? '=' : a_char_array1d_static_fld[k1];
         i1++;
         ac[i1] = i1 >= j ? '=' : a_char_array1d_static_fld[l1];
      }
```

Existen distintos tipos de ofuscadores desde los básicos que solo modifican la estructura general de un proyecto, cambiando los nombres de los paquetes, clases y métodos o modificando la estructura general de los paquetes; hasta los más avanzados, que además de incluir las funciones anteriores, alteran el flujo, insertan código distractor y cifran las constantes de cadenas, como llaves o mensajes de error, con el propósito de dificultar el revelado de información sensible.

El nivel y tipo de ofuscación dependerá mucho del tipo de proyecto que se trate, ya que si es una biblioteca de funciones, no es posible modificar los nombres de los paquetes, clases y métodos, por su parte, si se trata de una aplicación de escritorio, es importante respetar el nombre del paquete, clase y la llamada al método *main* que inicia la aplicación.

Una vez ofuscado el código, procedemos a firmar digitalmente los artefactos Java generados para el despliegue para lo que se necesita crear una llave que se almacena en un *keystore* específico. Para esto se utiliza la herramienta conocida como *keytool* como se muestra a continuación.

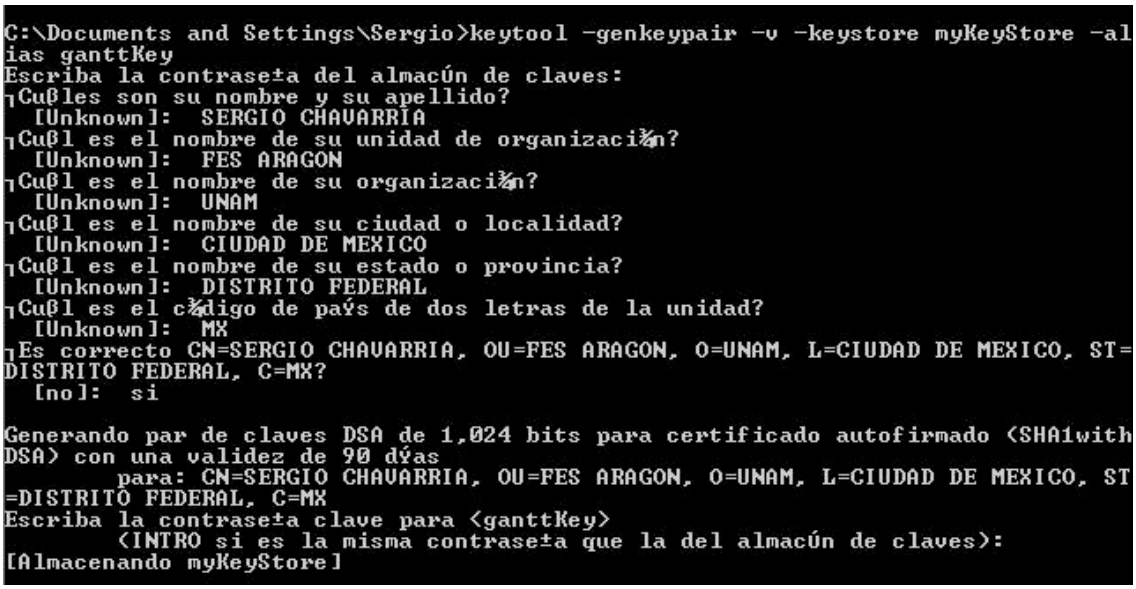


Figura 11: Ejemplo de generación de par de llaves con la herramienta keytool.

Una vez generadas el par de llaves es posible firmar el jar con la ayuda del comando *jarsigner* de la siguiente manera:

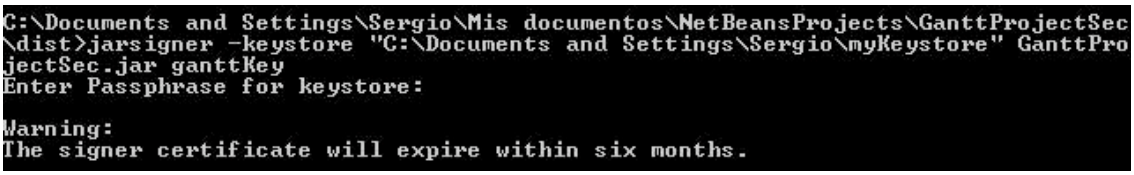


Figura 12: Uso del comando "jarsigner" para el firmado digital de artefactos de despliegue Java.

Un código firmado digitalmente con la herramienta "*jarsigner*" genera entradas en el archivo MANIFEST.MF donde por cada archivo *class* se genera su respectivo hash, además de agregarse dos archivos, que en nuestro caso son llamados GANTTKEY.DSA y GANTTKEY.SF. El primero contiene la firma digital del segundo usando el algoritmo DSA, así como el certificado de la entidad que firmó el archivo. El segundo archivo contiene el hash SHA de cada uno de los archivos *class* del jar y se genera a partir de las 3 líneas que se encuentran en el archivo MANIFEST.MF referentes a cada archivo *class*.

En este caso, solo se han considerado aquellas recomendaciones de seguridad aplicables al desarrollo de sistemas con Java en esta etapa del desarrollo, sin embargo, se requiere verificar que todo componente que forme parte del ecosistema siga recomendaciones de seguridad específicas, como son las recomendaciones de seguridad en sistemas operativos, bases de datos, redes, contraseñas de usuario, políticas, entre otros.

Como se describió a lo largo de este capítulo, la seguridad de la información es un campo tan extenso y complejo que requiere de su integración directa en todo el proceso de desarrollo de software a través del uso de diversos mecanismos que permitan mantener los requerimientos de seguridad especificados.

Capítulo 6: Resultados y Conclusiones.

Resumen: A partir de un análisis de la clasificación de debilidades comunes publicada por el MITRE se comprueba la necesidad de incorporar procesos y herramientas de análisis de seguridad que permitan encontrar y eliminar vulnerabilidades de los sistemas en etapas tempranas del proceso de desarrollo. Tomando como base este catálogo, se clasifican las debilidades por etapa de inserción dentro del ciclo de desarrollo y por lenguaje de programación. Este análisis permite ubicar al lenguaje Java dentro de un contexto de seguridad y se compara con los lenguajes de programación más comunes en la actualidad.

Tener una clasificación de los tipos de errores que crean vulnerabilidades en los sistemas; conocer la etapa de desarrollo donde estos se originan y saber la posición que tiene el lenguaje de programación Java en este análisis permitirá comprobar la necesidad del uso de métodos alternos y complementarios de desarrollo de software, así como resaltar las prácticas de programación segura recomendadas al momento de desarrollar aplicaciones Java.

6.1 Análisis de debilidades comunes.

El MITRE es una organización no lucrativa cuyo objetivo es apoyar en temas de ingeniera de sistemas, tecnología de la información, modernización empresarial y conceptos operacionales para el cuidado de las necesidades nacionales críticas.

Como parte de los temas de investigación realizados por esta organización, en su sitio de internet se incluye un catálogo de las debilidades más comunes en el desarrollo de software o CWE (*Common Weakness Enumeration*) el cual proporciona información para describir y clasificar las vulnerabilidades más comunes, los lenguajes de programación y plataformas en los que se presentan, así como las etapas dentro del ciclo de desarrollo de software donde se insertan, todo esto con el objetivo de permitir una mejor discusión, descripción y selección de servicios y herramientas de seguridad para encontrar estas debilidades en el código fuente y en sistemas operacionales, así como comprender las debilidades relacionadas con la arquitectura y el diseño.

Dentro del catálogo, cada debilidad define la etapa de desarrollo donde se crea y por lo tanto, donde se puede evitar. En este caso, las etapas de desarrollo consideradas son las siguientes:

1. Policy.
2. Requirements.
3. Architecture and Design.
4. Implementation.
5. Testing.
6. Bundling.
7. Distribution.
8. Installation.
9. Patch.
10. Documentation.
11. Porting.
12. System Configuration.
13. Operation.
14. Build and Compilation.

De acuerdo con la información consultada en su versión 1.5 publicada el 27 de julio del 2009 se presentan los siguientes resultados.

De un total de 260 debilidades clasificadas, la distribución de vulnerabilidades de acuerdo a su punto de inserción es la siguiente:

Tabla 1: Distribución de debilidades comunes por etapa del ciclo de desarrollo de software.

ETAPA DEL CICLO DE DESARROLLO	DEBILIDADES
Policy	0
Requirements	0
Architecture and Design	88
Implementation	237
Testing	0
Bundling	0
Distribution	0
Installation	1
Patch	0
Documentation	0
Porting	0
System Configuration	0
Operation	38
Build and Compilation	1

La gráfica de barras siguiente muestra esta distribución de debilidades de una manera más clara.

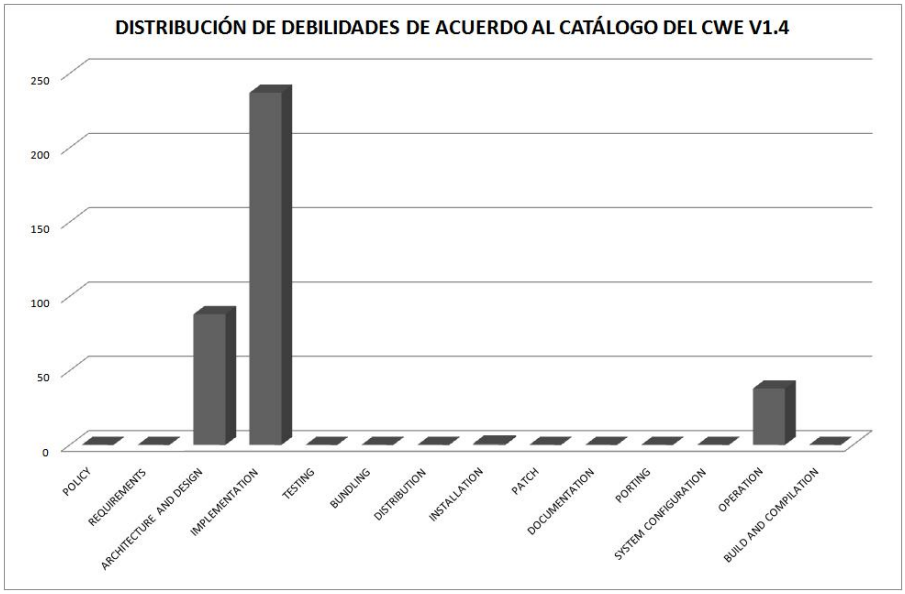


Figura 1: Gráfica de distribución de debilidades comunes por etapa del ciclo de desarrollo.

Tomando en cuenta que muchas de las debilidades se pueden introducir en más de una etapa del ciclo de desarrollo, una gráfica de distribución más realista se muestra a continuación:

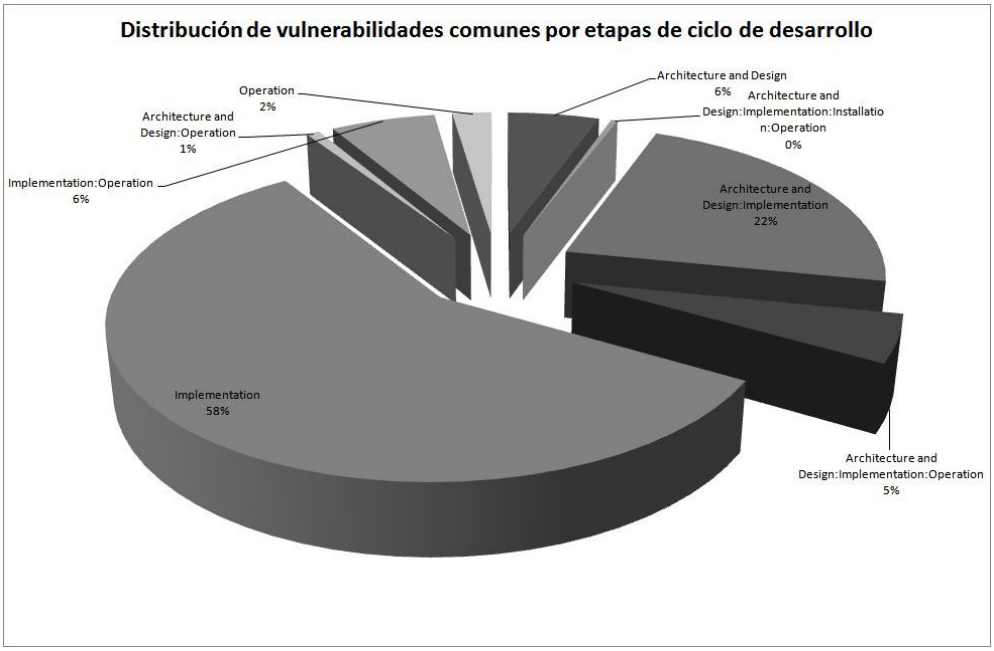


Figura 2: Distribución de vulnerabilidades comunes por etapas de ciclo de desarrollo.

Como se muestran en las gráficas anteriores las etapas críticas del desarrollo de sistemas son la arquitectura y diseño, y la implementación, abarcando entre las dos aproximadamente el 86% de las debilidades más comunes en los sistemas.

Como información adicional, cada debilidad dentro del catálogo de debilidades MITRE muestra información acerca de los lenguajes de programación a los que cada una de las debilidades se aplican. En nuestro análisis, solo se toman en cuenta los lenguajes de programación más comunes como son Java, la plataforma .NET, PHP, C y C++. En este caso, muchas debilidades se aplican a todos los lenguajes de programación y otras son propias de algún lenguaje en particular. La gráfica de barras siguiente muestra como se distribuyen las debilidades comunes por lenguaje de programación. En cada caso se toman en cuenta las debilidades aplicadas a todos los lenguajes más aquellas propias del lenguaje en particular.

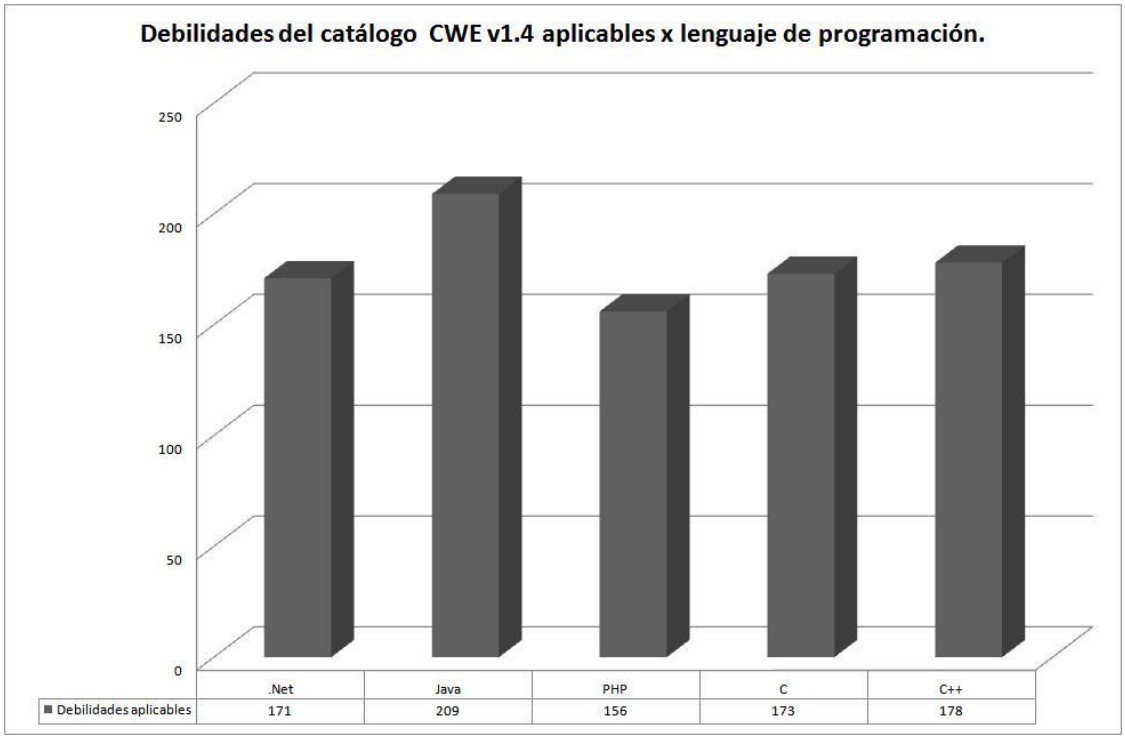


Figura 3: Distribución de debilidades CWE 1.4 por lenguaje de programación.

De acuerdo con este análisis en el lenguaje de programación Java existen mayor cantidad de defectos en los sistemas, lo cual contrasta con lo esperado, ya que al tratarse de un lenguaje de programación con características importantes de seguridad, uno esperaría encontrar una cantidad menor de debilidades en comparación con otros lenguajes.

Ya que este trabajo se enfoca al desarrollo de software utilizando el lenguaje de programación Java, procedemos a realizar un análisis más profundo a aquellas debilidades aplicadas a este lenguaje de programación. De esta manera, de las debilidades que se aplican únicamente al lenguaje de programación Java hacemos una clasificación por tecnología, es decir, las que se aplican a J2EE, Struts, EJB y las que son propias del lenguaje, obteniendo lo siguiente.

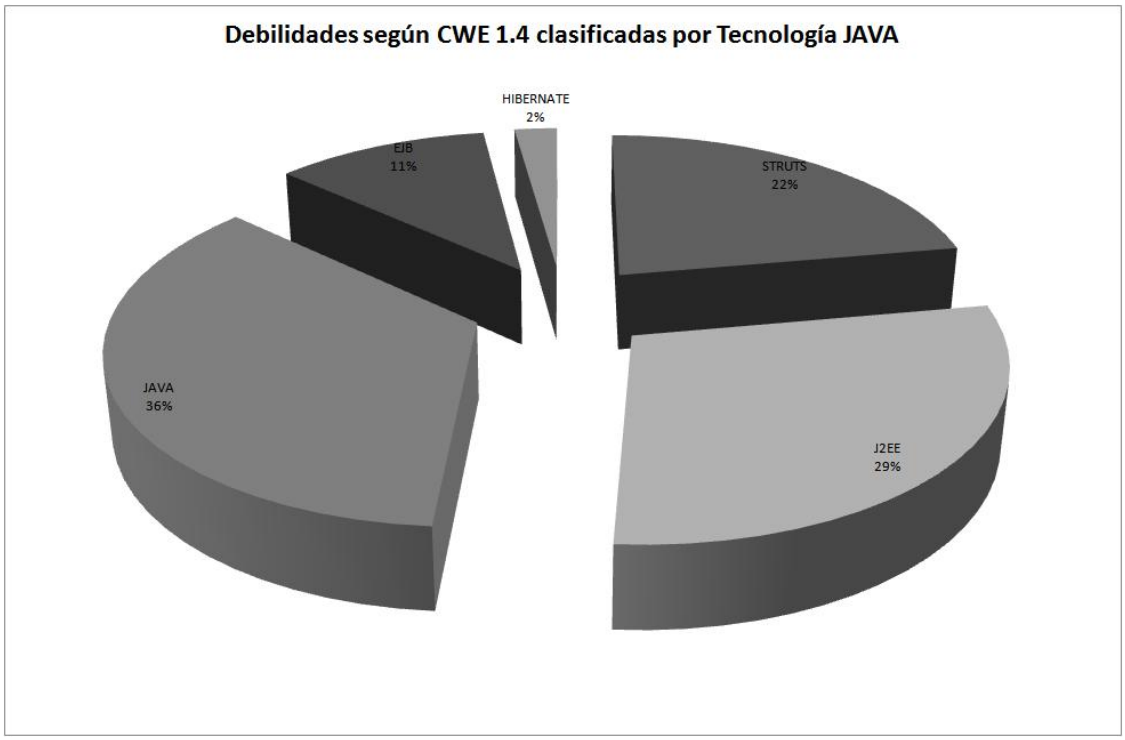


Figura 4: Gráfica de debilidades según CWE v1.4 clasificadas por tipo de tecnología JAVA aplicable.

De acuerdo con lo anterior concluimos que el aumento de debilidades aplicables al lenguaje de programación Java comparado con otros lenguajes de programación se debe al incremento en el desarrollo tecnológico aplicable para este lenguaje, demostrando que el aumento de tecnología y la dificultad de la misma incrementan la probabilidad de incorporar defectos en los sistemas. Así mismo, se demuestra que desarrollar en el lenguaje Java, no significa desarrollar software seguro a pesar de que el lenguaje está catalogado como uno de los más seguros en la actualidad.

El catálogo consultado para este análisis contiene aproximadamente 260 tipos de errores y no es el único existente en la actualidad. Leer, conocer, comprender y evitar cada uno de estos errores al momento de diseñar e implementar una aplicación no es una tarea fácil, por lo que se ha desarrollado una clasificación más sencilla que se muestra a continuación.

6.2 Clasificación taxonómica de errores de seguridad.

Con el objetivo de comprender el tipo de errores que pueden resultar en problemas de seguridad en los sistemas se han desarrollado distintas clasificaciones como la utilizada en el análisis anterior, que permiten comprender el origen de una vulnerabilidad, sin embargo, resultan en clasificaciones muy extensas, que muy probablemente no sean consultadas en su totalidad por los desarrolladores de sistemas. Con el propósito de simplificar esta clasificación y orientar a los desarrolladores en el tipo de errores que pueden desencadenar problemas de seguridad, la empresa *Fortify* en trabajo conjunto con *Gary McGraw*, han desarrollado una clasificación taxonómica de errores de seguridad en el desarrollo de software, publicados en un documento conocida como "*Seven Pernicious Kingdoms: A taxonomy of software security errors*". Esta clasificación divide los errores de seguridad en 7 + 1 reinos que, en orden de importancia, son los siguientes:

1. **Representación y validación de entradas:** Los problemas de representación y validación de entradas son causados por meta caracteres, problemas de codificación y representaciones numéricas. Una correcta validación de las entradas de datos previene una cantidad importante de ataques como pueden ser ataques de buffer overflow, sql injection, cross-site scripting, cache poisoning, entre otros. Es posible abordar este tipo de problemas en distintas etapas del desarrollo de software, como son la etapa de requerimientos, análisis y diseño e implementación. El mejor mecanismo para realizar validaciones de entrada de datos es con una lista blanca (white list), es decir, identificando los tipos de datos y caracteres adecuados para cada campo. En caso de que se requieran de caracteres especiales, se requiere establecer los métodos de escape y codificación necesarios.

2. **Abuso de API:** Básicamente, los errores de este tipo se crean cuando un desarrollador viola o altera el uso de las funciones proporcionadas por un API de desarrollo. Este tipo de errores no se pueden evitar mediante el uso del profile *UMLSec*, si no únicamente a través de la correcta capacitación de los desarrolladores, herramientas CASE y análisis de caja blanca. En el caso del lenguaje de programación Java se han desarrollado guías de codificación segura que ayudan a detectar errores o aciertos al momento de desarrollar una aplicación, aunque se recomienda una constante capacitación en la materia de acuerdo a la tecnología a usar. Ejemplos de errores clasificados en esta etapa son el uso de Sockets en desarrollo de aplicaciones web, manejo incorrecto de excepciones, entre otros.

3. **Características de seguridad:** Este tipo de errores se refiere a la ausencia o mala implementación de los servicios y mecanismos de seguridad en servicios de autenticación, control de acceso, confidencialidad, criptografía y gestión del privilegio. Estos errores solo se pueden evitar mediante una correcta captura de requerimientos de seguridad, su correcta documentación, identificación de amenazas y arquitectura y diseño de sistemas adecuado. Evitar estos errores es complicado y requieren de mucha experiencia en el tema y generalmente requiere de un especialista o consultor en seguridad. Algunos errores conocidos en este reino son la ausencia de controles de acceso, contraseñas hardcodeadas, criptografía débil, entre otras.

4. **Tiempo y estado:** Se refieren a errores lógicos y de acceso a recursos compartidos utilizando cómputo distribuido. Este tipo de errores generan ataques de negación de servicios o mal-funcionamiento. Algunos ejemplos de errores de este tipo son el uso de llamadas System.exit en aplicaciones web así como evitar el uso de hilos en aplicaciones web.

5. **Manejo de errores:** Este tipo de errores son de los más comunes en el desarrollo de sistemas. Generalmente los desarrolladores invierten la mayor cantidad de tiempo en el desarrollo de lo que se denomina "*happy path*" dejando el manejo de errores como tarea secundaria, sin embargo, el manejo de errores es tan importante como la funcionalidad esperada de un sistema. Errores de este tipo pueden generar en su mayoría revelación de información de implementación de sistema como son versión de sistemas operativos, servidores de aplicaciones, bases de datos, plataforma de desarrollo, además de provocar inestabilidad en el sistema y negación de servicios. Estos errores se abordan a lo largo de todo el ciclo de desarrollo de software, desde la captura de requerimientos funcionales, análisis, diseño, implementación e instalación del sistema y sus dependencias externas. La mejor manera de resolver estos errores es mediante una auditoría constante del sistema a lo largo de todo el desarrollo del mismo. Ejemplos de este tipo de errores son el uso de bloques *catch* vacíos.

6. **Calidad del código:** Un sistema no solo es un código en ejecución que respeta reglas de negocio establecidas, sino que está ampliamente relacionado con el tiempo de vida del mismo. Respetar estándares y reglas de codificación no solo ayuda a agilizar el desarrollo de actualizaciones de un producto, sino que se relaciona ampliamente con su estabilidad y evita fugas de información del sistema como pueden ser errores previos del sistema, parches aplicados, nombres de usuarios y contraseñas comentadas, servidores y cuenta de acceso en uso, información que revela el estatus de un sistema. La mayoría de estos errores se detectan en etapas de implementación y pruebas de caja blanca.

7. **Encapsulamiento:** Los errores de encapsulamiento son aquellos defectos insertados al momento de establecer los límites de ejecución necesarios para cada elemento de un sistema. Muchos errores de este tipo están ligados a lenguajes de programación orientados a objetos sin embargo también se aplican para los lenguajes de código móvil o remoto. En este sentido el lenguaje de programación Java cumple con ambas características, por tanto, es susceptible a ambos tipos de errores. En este sentido, la integración del profile *UMLSec* descrita en este trabajo y su relación con el lenguaje de programación Java ayuda a definir y mantener los requerimientos de seguridad al momento de codificar una aplicación. Algunos ejemplos de este tipo de error son el uso de clases internas y el uso incorrecto de los modificadores de acceso de las clases, métodos y atributos.

8. **Ambiente:** Generalmente son los defectos existentes en las dependencias externas del sistema. En este sentido, el análisis de dependencias externas permitirá identificar aquellos sistemas y configuraciones críticas que tienen una relación estrecha con el sistema en desarrollo y cuyas vulnerabilidades afectarían directamente la seguridad del desarrollo. Generalmente se aplican guías de configuración especiales dependiendo del tipo y versión de la dependencia. Se requiere que este análisis de dependencias se realice en etapas iniciales del desarrollo ya que información revelada en esta etapa puede establecer parámetros importantes en la arquitectura del sistema y selección de tecnología. Algunos errores clasificados en este grupo son el uso de protocolos de transporte inseguros, identificadores de sesión menores a 128 bits y la ausencia de páginas de error por default para evitar el revelado de información sensible del contenedor de aplicaciones.

A pesar de que la clasificación taxonómica anterior, es una clasificación orientada a las etapas de implementación y está dirigida principalmente a los desarrolladores de sistemas, muchos se pueden atacar en etapas previas como son la captura de requerimientos y el modelado de sistemas, aunque en general requiere de un proceso continuo ya descrito a lo largo de este trabajo.

Una vez presentados los análisis de debilidades comunes y la taxonomía de errores anterior, es posible conocer con mayor exactitud el problema de seguridad en el desarrollo de aplicaciones, y una vez realizado el estudio de las alternativas existentes, podemos concluir lo siguiente.

6.3 Conclusiones y propuestas.

Está demostrado que la seguridad es uno de los requerimientos indispensables en el manejo de información, sin embargo, no se considera de manera adecuada en el desarrollo de sistemas actual. Esta visión ha cambiado en años recientes y se han propuesto diversos métodos y mecanismos para atacar este requerimiento. El desarrollo de sistemas es complejo por sí mismo y agregar requerimientos de seguridad, aumenta su complejidad, por esto, se han desarrollado alternativas para facilitar el cumplimiento de ambos requerimientos.

Definitivamente las mejores opciones son aquellas que se adaptan a los conocimientos ya aprendidos, de tal manera que la curva de aprendizaje se minimice y permita su fácil y rápida adopción. En este contexto se crea el profile *UMLSec*, el cual se basa en el amplio uso del lenguaje UML para el modelado de sistemas. A pesar de que su creación es una aportación importante para el desarrollo seguro de sistemas, no es suficiente, ya que además de tener las desventajas ya descritas, no ataca todos los errores que originan problemas de seguridad.

UMLSec aborda uno de los problemas más importantes en el desarrollo de sistemas que es el problema de refinamiento aunque lo aborda de manera limitada, ya que no lo asocia a lo largo de un proceso de desarrollo en particular. Esto es comprensible ya que no pretende definir el proceso de desarrollo a utilizarse, sin embargo, una solución más completa para el desarrollo seguro de software debe definir los procesos necesarios que vigilen todos los requerimientos de seguridad asociados a un sistema. Es aquí donde se aplican los estándares de seguridad propuestos a través de modelos de capacidad y madurez, como el SSE-CMM. Al igual que UMLSec, este estándar no está asociado a un proceso de desarrollo, pero si define los procesos necesarios que cualquier institución debería realizar para proporcionar un nivel de riesgo aceptable y no solo se enfocan en el desarrollo de sistemas si no que lo conceptualiza desde un punto de vista organizacional.

Siguiendo con la idea de adaptar los procesos necesarios en modelos ya conocidos, se sugiere el proceso de desarrollo unificado con actividades propias para requerimientos de seguridad. En este sentido, para cada etapa de desarrollo del proceso unificado se sugieren actividades complementarias que permitan adoptar los requerimientos de seguridad en todo el desarrollo de sistemas, desde la captura de requerimientos de seguridad, creación de artefactos de seguridad específicos, análisis y modelado, implementación con estándares de codificación segura, pruebas y despliegue. En la actualidad, de todas estas etapas de desarrollo, las más críticas, de acuerdo con diversos estudios, son las de análisis, diseño e implementación, ya que se considera

que es en estas etapas donde se insertan la mayor cantidad de errores que derivan en problemas de seguridad. No obstante, esta hipótesis se basa en un proceso de desarrollo que no contempla actividades específicas para la verificación de seguridad, por tanto, en un proceso como Secure UP, en realidad todos los problemas de seguridad son creados desde las etapas iniciales del desarrollo, ya que no se contemplan ni requerimientos de seguridad, ni análisis de riesgos ni el análisis de patrones de ataque y casos de mal uso.

De acuerdo con lo anterior, concluimos que no hay una etapa dentro de un proceso de desarrollo, que con actividades de seguridad específicas, garantice la seguridad de los sistemas, es decir, seguir las recomendaciones de seguridad en la etapa de implementación, no garantiza que un sistema este seguro, se requiere contemplar las actividades de seguridad recomendadas para cada etapa de desarrollo.

En este trabajo se han definido y ejemplificado las actividades de seguridad recomendadas por los estándares definidos en la materia, así como software de apoyo que permite garantizar que los requerimientos se cumplan en cada etapa de desarrollo. Se utilizó el lenguaje de programación Java por interés propio y por sus características de seguridad inherentes al mismo, sorprendiéndonos el hecho de descubrir que efectivamente, desarrollar en un lenguaje con características de seguridad importantes no significa desarrollar código seguro, si no que se requiere de un amplio conocimiento en el lenguaje y de su correcta utilización para garantizar sus propiedades de seguridad.

Por todo lo anterior y una vez descritos las recomendaciones y mejores prácticas en el desarrollo seguro de sistemas, se ponen a consideración los siguientes puntos:

1. Está demostrado que la seguridad de la información no es un requerimiento que se deba abordar en las etapas finales del desarrollo de sistemas.
2. La seguridad no debe verse como una actividad reactiva sino preventiva.
3. Los mecanismos de seguridad son un medio, no el fin.
4. Una mala administración de proyectos es un problema importante de seguridad.
5. Se requiere de una capacitación constante en el tema para todo el personal involucrado en el desarrollo de sistemas, desde el analista de requerimientos hasta el administrador de sistema.
6. En las etapas de análisis, diseño e implementación no olvidar los 8 principios básicos al momento de crear la arquitectura del sistema.

7. Para los desarrolladores, considerar las características de seguridad y vulnerabilidades de cada tecnología o lenguaje de programación, desde el momento de su aprendizaje, ayudará a generar código de mejor calidad.
8. Para la etapa de pruebas, además de verificar la correcta funcionalidad de los sistemas, es recomendable el diseño de pruebas unitarias de seguridad específicas para la tecnología en uso.
9. Apoyarse en herramientas automáticas de pruebas de seguridad ayuda a detectar vulnerabilidades en etapas tempranas del desarrollo.
10. Y por último, nunca olvidar la frase: “Una cadena es tan fuerte como su eslabón más débil”.

Bibliografía

Allen, J. H., Barnum, S., Ellison, R. J., McGraw, G., & Mead, N. R. (2008). *Software Security Engineering.* Pearson Education.

Center, M. D. (n.d.). *.NET Framework Developer´s Guide*. Retrieved from http://msdn.microsoft.com/en-us/library/930b76w0(VS.71).aspx

(2006). *Common Criteria for Information Technology Security Evaluation.*

Consortium, P. S. (n.d.). *PHP Security Guide*. Retrieved from http://phpsec.org/projects/guide/1.html

Corporation, T. M. (2009). *Common Vulnerabilities and Exposures*. Retrieved from http://cve.mitre.org/

Gong, L., Ellison, G., & Dageforde, M. (2003). *Inside Java 2 Platform Security, Second Edition.* Addison-Wesley.

Gosling, J., & McGilton, H. (n.d.). *The Java Language Environmente*. Retrieved from http://java.sun.com/docs/white/lanenv

Guide, R. S. (n.d.). *Rational Software Modeler*. Retrieved from http://publib.boulder.ibm.com/infocenter/rsmhelp/v7r0m0/index.jsp?topic=/com.ibm.xtools.modeler.doc/topics/cdepend.html

HogLung, G., & McGraw, G. (2004). *Exploiting Software, How to break code.* Addison-Wesley.

Howard, M., & Lipner, S. (2006). *The Security Development Lifecycle.* USA: Microsoft Press.

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The Unified Software Development Process.* Addison Wesley.

Java Security Overview. (n.d.). Retrieved from Java Security Overview: http://java.sun.com/javase/6/docs/technotes/guides/security/overview/jsoverview.html

JODE. (n.d.). Retrieved from JODE: http://jode.sourceforge.net/

Jürgens, J. (2005). *Secure Systems Development with UML.* Springer.

McGraw, G. *Seven Pernicious Kingdoms: A taxonomy of Software Security Errors.* Building Security In .

Noopur, D. (2006, 07 05). *Secure Software Development Life Cycle Processes*. Retrieved 03 16, 2009, from https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/sdlc/326-BSI.html

Saltzer, J. H., & Schroeder, M. D. (Septiembre 1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE* , 1278-1308.

Seacord, R. c. (2006). *Secure Coding in C and C++*. US: Addison Wesley.

Shostack, A. (2007, Septiembre 11). *STRIDE Chart*. Retrieved Julio 27, 2009, from http://blogs.msdn.com/sdl/archive/2007/09/11/stride-chart.aspx

Software, F. (2009). *Fortify Taxonomy: Software Security Errors*. Retrieved Julio 20, 2009, from http://www.fortify.com/vulncat/en/vulncat/index.html

Steel, C., Nagappan, R., & Lai, R. (2006). *Core Security Patterns.* US: Prentice Hall.

The Legion Of Bouncy Castle. (n.d.). Retrieved from http://www.bouncycastle.org/

Vries, S. d. (2006). *Security Testing through Automated Software Test.* The OWASP Foundation.

Wutka, M. (1996, Enero 11). *Java Tip 20: Write a custom security manager that gathers execution information*. Retrieved Julio 19, 2009, from http://www.javaworld.com/javaworld/javatips/jw-javatip20.html