Universidad de las Ciencias Informáticas.

Facultad 5, Centro Informática Industrial.

Título: Pruebas automatizadas para el módulo Seguridad del Centro de Informática Industrial.

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas.

Autor(es): Lilian Laudinot Quiñones.

Miguel Ángel Ramírez Maturell.

Tutores: Ing. Enrique Reyes Bermúdez.

Ing. Hardys Pérez Bermúdez.

Habana, 2013.

AGRADECIMIENTOS

De Lilian:

A mi mami, Esmerida por ser la persona que más quiero, por haberse sacrificado y darme todo lo que ha podido sin pedir nunca nada a cambio, por apoyarme y respetar cada una de las decisiones que he tomado. Gracias mami por ser esa personita tan linda y maravillosa que eres.

A mis suegros, Estrella y Luis Gabriel por haberme admitido y querido como una hija más, por darme su confianza y su cariño.

A mi novio, William por apoyarme en los momentos buenos y malos, por ayudarme durante estos años que hemos compartido juntos, por su paciencia, su compresión y amor.

A toda mi familia, a mi abuela Esmerida, a mis tíos Fernando, Manolo, Oscar, Gorgue, Máximo, a mis tías Elaisi, Ada, a mis primos y primas por haberme dado su cariño.

A mi otra familia, en especial a mi tío Guevarin, a Fela, Zoe, Rita María, Cary por tratarme como una más de la familia, por ayudarme y darme su amor.

A mis amistades, especialmente a las del 92105 Lilibet, Lislien, Suly, Yamilka, Aliane, Madonna, Amal, Arianna, Yensy y Alina por haberme dado su amistad incondicional, por ser las personas con las cuales comparto el apartamento y con las que más converso.

De Miguel Ángel:

Agradezco a todos los que han colaborado de una forma u otra al desarrollo en este trabajo.

DEDICATORIA

De Lilian:

A mi mami Esmerida, por apoyarme en todas las decisiones que he tomado, por estar presente cuando más te he necesitado, por haberme guiado por el camino correcto y por ser mi guía.

A mis suegros, Estrella y Luis Gabriel, por darme su cariño, su confianza y haberme admitido como una hija más, gracias por ser mis segundo padres.

A mi novio William por ser mi fiel compañero durante estos 4 años, por compartir los bueno y malos momentos, por su apoyo y amor.

De Miguel Ángel:

Dedico esta tesis en especial a mi mama y a mis abuelos maternos por ser la luz de mi vida.

DECLARACIÓN DE AUTORÍA

Declaramos ser autores de la presente tesis y reconocemos a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste se firma a los ____ días del mes de __________ del año ____.

Lilian Laudinot Quiñones	Miguel Ángel Ramírez Maturell
________________________	________________________
FIRMA DE LA AUTORA	**FIRMA DEL AUTOR**
Enrique Reyes Bermúdez	Hardys Pérez Bermúdez
________________________	________________________
FIRMA DEL TUTOR	**FIRMA DEL TUTOR**

DATOS DE CONTACTO

Tutor: Ing. Enrique Reyes Bermúdez.

Edad: 27 años.

Ciudadanía: cubano.

Título: Ingeniero en Ciencias Informáticas.

Categoría docente: Instructor.

Email: ereyes@uci.cu.

Graduado de Ingeniería en Ciencias Informáticas en la Universidad de las Ciencias Informáticas. Actualmente se desempeña como jefe de la línea de Seguridad del Centro de Ingeniería Industrial de la UCI con 8 años de experiencia.

Tutor: Ing. Hardys Pérez Bermúdez

Edad: 30 años.

Ciudadanía: cubano.

Título: Ingeniero en Ciencias Informáticas.

Categoría docente: Instructor.

Email: hperezb@uci.cu.

Graduado de Ingeniería en Ciencias Informáticas en la Universidad de las Ciencias Informáticas. Actualmente se desempeña como Instructor y tiene 6 años de experiencia en los sistemas SCADA.

RESUMEN

La Universidad de las Ciencias Informáticas (UCI) cuenta con diferentes Centros de Desarrollo de Software, uno de estos centros es el de Informática Industrial (CEDIN) de la Facultad 5, el mismo tiene como objetivo desarrollar productos para varios clientes tanto nacionales como internacionales. Estos productos cuentan con un módulo Seguridad destinado a cumplir con todas las tareas referentes a la seguridad del sistema, tales como autenticar usuario, controlar acceso a los recursos, así como administrar usuarios, perfiles y grupos operacionales de privilegios. La seguridad es un tema bien complejo y en constante evolución, tanto para fabricantes de tecnologías, diseñadores de la ingeniería de sus aplicaciones, como para las organizaciones profesionales que se orientan en el planteamiento de su normativa y regulación.

En el CEDIN existe un grupo de estudiantes y profesores encargados de realizar las pruebas de forma manual a estos productos, método que consume gran cantidad de tiempo y recursos, obteniéndose imprecisión en la detección de las no conformidades. Es por ello que se necesita desarrollar una aplicación que permita probar todas las funcionalidades del módulo Seguridad de forma automatizada, detectando la mayor cantidad de errores posibles para ser corregidos y obtener un producto de una alta calidad.

El resultado de la presente investigación es una aplicación que permita realizar pruebas automatizadas al módulo Seguridad del CEDIN.

Palabras clave: automatizadas, pruebas, seguridad.

ÍNDICE

AGRADECIMIENTOS .... **II**

DEDICATORIA .... **III**

DECLARACIÓN DE AUTORÍA .... **IV**

DATOS DE CONTACTO .... **V**

RESUMEN .... **VI**

ÍNDICE .... **VII**

ÍNDICE DE FIGURAS .... **X**

ÍNDICE DE TABLAS. .... **XII**

INTRODUCCIÓN **13**

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA. **15**

1.1 PRUEBAS DE SOFTWARE 15

1.1.1 Objetivos de las Pruebas de software 15

1.1.2 Principios de las Pruebas de software 15

1.1.3 Características de las Pruebas de software 16

1.1.4 Niveles de pruebas 16

1.1.5 Métodos de Pruebas 18

1.1.6 Técnicas de caja blanca 18

1.1.7 Artefactos de Prueba 19

1.1.8 Pruebas manuales 20

1.1.9 Pruebas automatizadas 20

1.2 FRAMEWORKS PARA PRUEBAS AUTOMATIZADAS 20

1.2.1 Selección del framework 22

1.2.2 Importancia de los criterios de selección 22

1.2.3 Alternativas 23

1.2.4 Selección del framework de prueba 23

1.3 METODOLOGÍAS DE DESARROLLO DE SOFTWARE 24

1.3.1 Proceso Unificado de desarrollo (RUP para sus siglas en inglés) 24

1.3.2 OpenUP..........25
1.3.3 Programación Extrema (XP para sus siglas en inglés)..........27
1.4 HERRAMIENTAS DE DESARROLLO..........28
1.4.1 Entorno de Desarrollo Integrado (IDE por sus siglas en inglés)..........28
1.4.2 Lenguaje de programación..........28
1.5 CONCLUSIONES PARCIALES..........29
CAPÍTULO 2: PROCESO DE PRUEBAS AUTOMATIZADAS..........31
2.1 REQUISITOS FUNCIONALES DEL MÓDULO SEGURIDAD..........31
2.2 MODELO DE CASOS DE USOS DEL MÓDULO SEGURIDAD..........32
2.3 PLAN DE PRUEBAS..........32
2.3.1 Roles y responsabilidades..........32
2.3.2 Recurso necesarios para la ejecución de las pruebas..........33
2.3.3 Estrategia de prueba..........33
2.4 DISEÑO DE LOS CASOS DE PRUEBA..........34
2.4.1 DISEÑO DE LOS CASOS DE PRUEBAS DE LA BIBLIOTECA SSLMM..........34
2.4.1.1 Método de base64DecodeString..........34
2.4.1.2 Método de base64EncodeString..........37
2.4.1.3 Método aesEncrypt..........38
2.4.1.4 Método aesDecrypt..........40
2.4.1.5 Método getHash..........41
2.4.1.6 Método makeX509..........44
2.4.1.7 Método saveCertificate..........47
2.4.1.8 Método LoadCertificate..........49
2.4.1.9 Método makePrivateKey..........51
2.4.1.10 Método savePrivateKey..........53
2.4.1.11 Método loadPrivateKey..........54
2.4.2 DISEÑO DE LOS CASOS DE PRUEBAS DE LA BIBLIOTECA ISECURITY..........56
2.4.2.1 Método ClientAuthentication..........56
2.4.2.2 Método GetTicketsForService..........60

2.4.2.3 Método VerifyServiceForClientTicket 62
2.4.3 CASOS DE PRUEBAS DEL SERVIDOR 64
2.4.3.1 Método Server Authentication 64
2.4.3.2 Método AddService...68
2.4.3.3 Método RegisterClient..70
2.5 GENERADOR DE DATOS 73
2.6 CONCLUSIONES PARCIALES 74
CAPÍTULO 3: IMPLEMENTACIÓN Y RESULTADOS OBTENIDOS 75
3.1 DIAGRAMA DE CLASES DEL DISEÑO...75
3.2 DIAGRAMA DE COMPONENTES..76
3.3 IMPLEMENTACIÓN..76
3.4 ANÁLISIS DE LOS RESULTADOS 78
3.5 CONCLUSIONES PARCIALES. 81
CONCLUSIONES 82
RECOMENDACIONES 83
BIBLIOGRAFÍA 84
GLOSARIO 86
ANEXOS..87

ÍNDICE DE FIGURAS

Figura 1 Notación de grafos de flujos para las instrucciones Case, While, If y Secuencia 19

Figura 2 Metodología OpenUP 26

Figura 3 Modelo de casos de uso del módulo Seguridad..32

Figura 4 Grafo del método base64DecodeString...34

Figura 5 Grafo del método base64 35

Figura 6 Grafo del método uniteLines..36

Figura 7 Grafo del método base64EncodeString 37

Figura 8 Grafo del método aesEncrypt 39

Figura 9 Grafo del método aesDecrypt..40

Figura 10 Grafo del método getHash...42

Figura 11 Grafo del método digestToHexString..43

Figura 12 Grafo del método makex509...45

Figura 13 Grafo del método strToEVP_PKEY 46

Figura 14 Grafo del método saveCertificate...47

Figura 15 Grafo del método strToX509..49

Figura 16 Grafo del método loadcertificate 49

Figura 17 Grafo del método x509ToStr..50

Figura 18 Grafo del método makePrivateKey 51

Figura 19 Grafo del método EVP_PKEYToStr...52

Figura 20 Grafo del método savePrivateKey 53

Figura 21 Grafo del método loadPrivateKey 54

Figura 22 Grafo del método ClientAuthentication...56

Figura 23 Grafo del método clientAuthentication 58

Figura 24 Grafo del método sendSecurityData 59

Figura 25 Grafo del método GetTicketsForService..61

Figura 26 Grafo del método VerifyServiceForClientTicket 62

Figura 27 Grafo del método ServerAuthentication 64

Figura 28 Grafo del método AddService...68

Figura 29 Grafo del método addService de iComm 69

Figura 30 Grafo del método RegisterClient....................70

Figura 31 Diagrama de clases....................75

Figura 32 Diagrama de componente....................76

Figura 33 Reporte de pruebas....................78

ÍNDICE DE TABLAS.

Tabla 1 Importancia de los criterios de selección........23

Tabla 2 Cálculo de puntajes basado en criterios e importancia........23

Tabla 3 Roles y responsabilidades........33

Tabla 4 Servidores........33

Tabla 5 PC clientes........33

Tabla 6 Representación del caso de prueba del método base64DecodeString........35

Tabla 7 Representación de los casos de prueba del método base64StringFormat........36

Tabla 8 Representación de los casos de prueba del método uniteLines........37

Tabla 9 Representación de los casos de prueba del método base64EncodeString........38

Tabla 10 Representación de los casos de prueba del método aesEncrypt........39

Tabla 11 Representación de los casos de prueba del método aesDecrypt........41

Tabla 12 Representación de los casos de prueba del método getHash........43

Tabla 13 Representación de los casos de prueba del método digestToHexString........44

Tabla 14 Representación de los casos de prueba del método makeX509........46

Tabla 15 Representación de los casos de prueba del método strToEVP_PKEY........47

Tabla 16 Representación de los casos de prueba del método saveCertificate........48

Tabla 17 Representación de los casos de prueba del método strToX509........49

Tabla 18 Representación de los casos de prueba del método loadcertificate........50

Tabla 19 Representación de los casos de prueba del método x509ToStr........51

Tabla 20 Representación de los casos de prueba del método makePrivateKey........52

Tabla 21 Representación de los casos de prueba del método EVP_PKEYToStr........53

Tabla 22 Representación de los casos de prueba del método savePrivateKey........54

Tabla 23 Representación de los casos de prueba del método loadPrivateKey........55

Tabla 24 Representación de los casos de prueba del método ClientAuthentication........58

Tabla 25 Representación de los casos de prueba del método ClientAuthentication........59

Tabla 26 Representación de los casos de prueba del método ClientAuthentication........60

Tabla 27 Representación de los casos de prueba del método GetTicketsForService........62

Tabla 28 Representación de los casos de prueba del método VerifyServiceForClientTicket........64

Tabla 29 Representación de los casos de prueba del método ServerAuthentication........68

Tabla 30 Representación de los casos de prueba del método ServerAuthentication 68

Tabla 31 Representación de los casos de prueba del método addService de iComm 70

Tabla 32 Representación de los casos de prueba del método..73

Tabla 33 Casos de prueba de la biblioteca SSLmm 78

Tabla 34 Casos de prueba del cliente y servidor de seguridad...80

Tabla 35 No conformidades de la biblioteca SSLmm...81

INTRODUCCIÓN

Cuba en su estrategia de inclusión en el mercado internacional, ha destinado importantes recursos para llevar a cabo el desarrollo e investigación del sector informático. Las compañías dedicadas al desarrollo del software se han tornado más competentes en cuanto a los productos que se comercializan y el país para poder ocupar un lugar de prestigio a nivel mundial en esta esfera, demanda la liberación de productos con una alta calidad.

En la Universidad de las Ciencias Informáticas (UCI) se cuenta con diferentes Centros de Desarrollo de Software, orientados a cumplir las estrategias del avance tecnológico del país con una variedad de proyectos tanto nacionales como internacionales. En ellos, la calidad es una premisa importante en aras de garantizar un producto de competencia en el mercado internacional, cumpliendo con los estándares establecidos dentro de la calidad de software.

El Centro de Informática Industrial (CEDIN) de la Facultad 5 tiene como objetivo el desarrollo de productos para varios clientes. Es indispensable que la calidad requerida de estos productos tome un valor mucho más significativo para evitar que las dificultades en sus diseños e implementación puedan causar daños económicos irreparables. *Un error de software, comúnmente conocido como bug (bicho), es una falla en un programa de computador o sistema de software que desencadena un resultado indeseado* (1). Otro factor de vital importancia es la seguridad que está basado fundamentalmente en el control de acceso y permisos. Tiene su mayor auge desde que personas con intereses mal intencionados han intentado hacer estragos en cualquier tipo de sistemas, siendo los más afectados los basados en aplicaciones web, debido al creciente uso de las redes; no obstante, ningún sistema está exento de ataques ya sea intencionados o no.

Los productos desarrollados por el CEDIN cuentan con un módulo llamado Seguridad destinado a cumplir con todas las tareas referentes a la seguridad del sistema entre ellas autenticar usuarios, controlar el acceso a los recursos, así como administrar usuarios, perfiles y grupos operacionales de privilegios. El diseño de la seguridad es una tarea compleja porque es necesario cumplir con un grupo de estándares establecidos internacionalmente en un mercado donde, muy a menudo, surgen nuevas formas y métodos de burlar los sistemas de seguridad de cualquier software, por lo tanto es necesario e imprescindible un seguimiento investigativo periódico de temas relacionados con la seguridad.

En el proceso de desarrollo de software las pruebas comprenden una fase muy importante pues son las que permiten verificar y revelar la calidad de un producto, siendo utilizadas para identificar posibles fallos de implementación, calidad o usabilidad de un sistema. Estas son realizadas por el grupo de calidad y prueba del Departamento de Integración y Despliegue, el cual está integrado por estudiantes y profesores que se encargan de hacerles pruebas manuales con las cuales son verificados cada uno de los componentes y funcionalidades del sistema. Además, por la importancia que tiene, es necesario hacerles pruebas extensas y rigurosas al módulo de seguridad de modo tal que este cumpla con todo lo que se definió. Dada las características de las pruebas existe la probabilidad de error humano y el tiempo de entrega del producto se hace más extenso, provocando un elevado esfuerzo del grupo que las realiza.

La presente investigación parte del siguiente **problema**: ¿Cómo detectar con mayor precisión y eficiencia los errores en las funcionalidades del módulo Seguridad del Centro Informática Industrial?

Para solucionar esta interrogante se plantea el siguiente **objeto de estudio**: las pruebas de software.

El **objetivo general** de la investigación: Desarrollar una aplicación para realizar pruebas automatizadas al módulo Seguridad que permita disminuir sus vulnerabilidades. Dentro de esta área se define el **campo de acción**: las pruebas automatizadas de software para el módulo Seguridad del Centro Informática Industrial.

Para lograr el total cumplimiento del objetivo se trazaron las siguientes **tareas investigativas**:

- Estudio y análisis de la información referente a las pruebas de software, encaminado dentro del proceso, a las pruebas automatizadas.
- Elaboración del marco teórico de la investigación a partir del estado del arte existente sobre el tema.
- Selección de las herramientas y tecnologías a utilizar.
- Realización del Plan de pruebas.
- Aplicación de la técnica del camino básico para obtener los diseños de casos de prueba.
- Implementación de una aplicación de pruebas automatizadas para la biblioteca *SSLmm* e *ISecurity*, además del servidor de Seguridad.
- Ejecución de las Pruebas automatizadas.
- Documentación de los resultados obtenidos.

Para el desarrollo de esta investigación fue necesario utilizar algunos métodos investigativos que se listan a continuación:

Métodos teóricos:

- Analítico-Sintético: se empleó con el objetivo de analizar información necesaria en la investigación a partir del estudio de diferentes documentos, para luego hacer una extracción de los elementos más importantes que se relaciona con las pruebas de software.
- Análisis Histórico-Lógico: para comenzar con el trabajo se hizo un estudio del arte de las pruebas de software.

El presente trabajo de diploma consta de 3 capítulos de contenido. Los que se encuentran estructurados de la siguiente manera.

Capítulo 1: Fundamentación teórica: se hace un análisis bibliográfico en donde se investigan conceptos, objetivos, principios, características de las pruebas de software, además de un pequeño análisis de las posibles metodologías a utilizar y herramientas.

Capítulo 2: Proceso de Pruebas automatizadas: en este capítulo se realiza un análisis de los requisitos, se hace una breve descripción de los casos de uso, el plan de prueba y los diseños de casos de prueba.

Capítulo 3: Implementación y resultados obtenidos: se hace un breve resumen de los algoritmos utilizados para el desarrollo de la aplicación y una descripción de los resultados obtenidos.

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA.

Introducción

En la Ingeniería de software las pruebas han tomado cada vez más importancia, debido al incremento de la complejidad de los productos de software. Durante la implementación se pueden cometer errores debido a los cambios constantes durante esta fase, el nuevo código generado puede tener errores o simplemente afectar el funcionamiento del que fue previamente implementado y probado. Es por esto que se hace indispensable realizar pruebas constantes durante todo el desarrollo del producto, para que esto se pueda llevar a cabo de manera exitosa las pruebas manuales no son las más ideales porque se vuelven más costosas y toman mucho tiempo, en cambio, las pruebas automatizadas se pueden hacer en un corto tiempo y con resultados más fiables.

1.1 Pruebas de software

En el proceso de desarrollo de software intervienen dinámicamente la realización de las pruebas, para así garantizar que el producto sea confiable y funcional.

Roger Pressman define las pruebas de software como: un elemento crítico para la garantía de calidad del software y representa una revisión final de las especificaciones, del diseño y de la codificación. (9)

Por otra parte la IEEE define una prueba como: una actividad en la cual un sistema o componente es ejecutado bajo condiciones específicas, se observan o almacenan los resultados y se realiza una evaluación de algún aspecto del sistema o componente. (18)

Luego de haber analizado estos conceptos se concluye que las pruebas de software es una actividad en la cual el software es ejecutado bajo condiciones específicas, donde las condiciones están determinadas por todas las posibles combinaciones donde este sea propenso al fallo. Además los resultados son observados y registrados.

1.1.1 Objetivos de las Pruebas de software

A continuación se mencionan algunos objetivos que se deben perseguir en el diseño y ejecuciones de pruebas de software (9).

- Probar si el software hace lo que no debe.
- Una prueba tiene éxito si descubre un error no descubierto hasta entonces.
- Encontrar el mayor número de errores con la menor cantidad de tiempo y esfuerzo posibles.
- Mostrar hasta qué punto las funciones del software operan de acuerdo con las especificaciones y requisitos del cliente.

1.1.2 Principios de las Pruebas de software

Las pruebas se rigen por una serie de principios, una buena comprensión de estos facilitará el posterior uso de los métodos en un efectivo diseño de casos de prueba. A continuación se mencionan los principios (9):

- La prueba puede ser usada para mostrar la presencia de errores, pero nunca su ausencia.

- Evitar casos de pruebas no planificados, no reusables y triviales a menos que el programa sea verdaderamente sencillo.
- Una parte necesaria de un caso de prueba es la definición del resultado esperado.
- Los casos de pruebas tienen que ser escritos no solo para condiciones de entradas válidas y esperadas sino también para condiciones no válidas e inesperadas.
- El número de errores sin descubrir es directamente proporcional al número de errores descubiertos.
- Se deben evitar los casos desechables.
- Las pruebas son una tarea creativa como el desarrollo del software.

1.1.3 Características de las Pruebas de software

A continuación se mencionan algunas características que una prueba debe cumplir (9):

- Alta probabilidad de encontrar un error: el ingeniero de software debe tener un alto nivel de entendimiento de la aplicación a construir para poder diseñar casos de prueba con las que se encuentre el mayor número de defectos.
- Una prueba no debe ser redundante: el tiempo y los recursos destinados a las pruebas son limitados. No hay razón para realizar una prueba que tenga el mismo propósito que otra. Cada prueba debe tener un propósito diferente.
- Una prueba no debería ser ni demasiado sencilla ni demasiado compleja: aunque a veces es posible combinar una serie de pruebas en un caso de prueba, los posibles efectos colaterales asociados con este enfoque podrían enmascarar errores. En general, cada prueba debe ejecutarse de forma separado.

1.1.4 Niveles de pruebas

Los niveles de prueba son diferentes ángulos para verificar y validar un producto de software. Existen diferentes niveles de Prueba de software, los cuales se mencionan a continuación:

Prueba unitaria: se concentra en probar cada componente individualmente para asegurar que funcione de manera apropiada como unidad. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.

Prueba de integración: es una técnica sistemática para construir la estructura del programa, mientras al mismo tiempo, se aplican pruebas para detectar errores asociados con la interacción. El objetivo es tomar los módulos probados en unidad y estructurar un programa que esté de acuerdo con el que dicta el diseño (7).

Las pruebas de integración tienen como base las pruebas unitarias y consisten en una progresión ordenada de testeos para los cuales los distintos módulos van siendo ensamblados y probados hasta haber integrado el sistema completo. Si bien se realizan sobre módulos ya probados en forma individual, no es necesario que se terminen todas las pruebas unitarias para comenzar con las de integración. Dependiendo de la forma en que se organicen, se pueden realizar en paralelo a las unitarias.

Existen principalmente dos enfoques de integración: (2)

- La integración descendiente: el módulo principal es usado como controlador y todos sus módulos subordinados son remplazados por módulos simulados. Los módulos simulados se remplazan uno a la vez con los componentes reales (en profundidad) y se van probando.
- La integración ascendiente: a diferencia del enfoque descendente, éste inicia la construcción y prueba de los módulos en los niveles más bajos de la estructura del programa. No se requiere el uso de módulos simulados.

Pruebas de sistema: se realizan cuando el software está funcionando como un todo. Es la actividad de prueba dirigida a verificar el programa final, después que todos los componentes de software y hardware han sido integrados. En un ciclo iterativo estas pruebas ocurren más temprano, tan pronto como subconjuntos bien formados de comportamiento de caso de uso son implementados (30).

Tipos de prueba de sistema:

- Pruebas de recuperación: se simulan fallas de software o hardware para verificar la eficacia del proceso de recuperación.
- Pruebas de rendimiento: tiene como objeto evaluar el rendimiento del sistema integrado en condiciones de uso habitual.
- Pruebas de resistencia o de estrés: comprueban el comportamiento del sistema ante situaciones donde se demanden cantidades extremas de recursos (número de transacciones simultáneas anormal, excesivo uso de las memorias).
- Pruebas de seguridad: se utilizan para testear el esquema de seguridad intentando vulnerar los métodos utilizados para el control de accesos no autorizados al sistema.

Pruebas de aceptación: al igual que las de sistema, se realizan sobre el producto terminado e integrado, pero a diferencia de aquellas, están concebidas para que sea un usuario final quien detecte los posibles errores. Su objetivo es verificar que el software está listo y que puede ser usado por usuarios finales para ejecutar aquellas funciones y tareas para las cuales el software fue construido.

Se clasifican en dos tipos:

- Pruebas Alfa: se realizan por un cliente en un entorno controlado por el equipo de desarrollo. Para que tengan validez, se debe primero crear un ambiente con las mismas condiciones que se encontrarán en las instalaciones del cliente. Una vez logrado esto, se procede a realizar las pruebas y a documentar los resultados.
- Pruebas Beta: se realizan en las instalaciones propias de los clientes. Para que tengan lugar, en primer término, se deben distribuir copias del sistema para que cada cliente lo instale en sus oficinas, dependencias o sucursales. Si se trata de un número reducido de clientes, el tema de la distribución de las copias no representa grandes dificultades, pero en el caso de productos de venta masiva, la elección de los betas testers debe realizarse con sumo cuidado. En este caso cada usuario realizará

sus propias pruebas y documentará los errores que encuentre, así como las sugerencias que crea conveniente realizar.

1.1.5 Métodos de Pruebas

Existen dos métodos que pueden probar cualquier producto construido.

- Pruebas de caja negra: se refiere a las pruebas que se llevan a cabo sobre la interfaz del software. Este tipo de prueba examina algún aspecto funcional de un sistema que tiene poca relación con la estructura lógica interna del software.
- Pruebas de caja blanca: este método permite probar los caminos lógicas del software, al proporcionar casos de pruebas que ejerciten conjuntos específicos de condiciones, bucles o ambos. Se puede inspeccionar el estado del programa en diferentes puntos para determinar si el estado real coincide con el esperado. El objetivo de este método es diseñar casos de prueba que se ejecuten, al menos una vez todas las sentencias del programa y todas las condiciones. .

1.1.6 Técnicas de caja blanca

Para desarrollar la prueba de caja blanca existen varias técnicas, entre las cuales se encuentran:

- La prueba de condición: es un método de diseño de casos de prueba que ejercita las condiciones lógicas contenidas en el módulo de un programa.
- La prueba de flujo de datos: se selecciona caminos de prueba de un programa de acuerdo con la ubicación de las definiciones y los usos de las variables del programa.
- La prueba de bucles: se centra exclusivamente en la validez de las construcciones de bucles.
- La prueba del camino básico: permite obtener una medida la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución.

El nivel utilizado para realizar las pruebas a los distintos componentes del módulo Seguridad son las pruebas unitarias con el método de caja blanca debido a que este método permite probar los caminos lógicos del software y para poder realizar esto, se escoge como mejor opción la técnica del camino básico donde sus principales características son: fáciles de usar y permite obtener una medida de la complejidad lógica de cada uno de los métodos de los distintos componentes del módulo Seguridad y usar esa medida para definición de un conjunto de caminos de ejecución.

Los pasos que se siguen para aplicar esta técnica son:

1. A partir del código fuente, se dibuja el grafo de flujo asociado.

2. Se calcula la complejidad ciclomática del grafo.

2.1 El número de regiones del grafo de flujo coincide con la complejidad ciclomática.

La complejidad ciclomática (G), se define como (G)=A-N+2 donde A: es el número de aristas del grafo y N es el número de nodos.

2.2 La complejidad ciclomática, V (G), también se define como (G)=P+1 donde P es el número de nodos predicados.

3. Se determina un conjunto básico de caminos independientes.

4. Se preparan los casos de prueba que obliguen la ejecución de cada camino del conjunto básico.

Para aplicar esta técnica se debe introducir una sencilla notación para la representación del flujo de control. A continuación se presenta la notación:

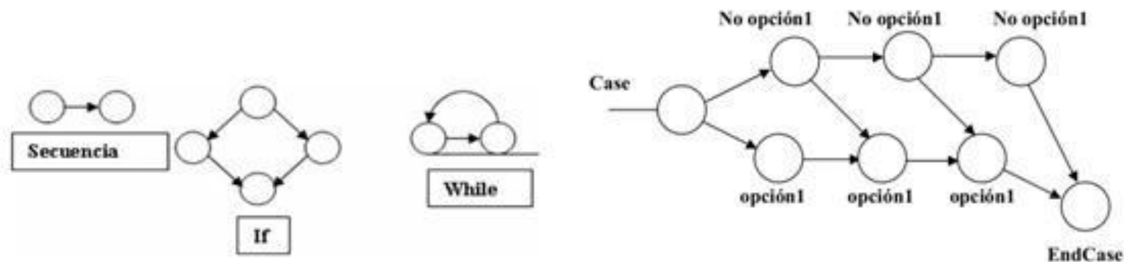

Figura 1 *Notación de grafos de flujos para las instrucciones Case, While, If y Secuencia*

Un grafo de flujo está formado por 3 componentes fundamentales:

- Nodos: cada círculo es denominado un nodo del grafo de flujo, representa una o más sentencias procedimentales.
- Flechas: denominadas aristas, las mismas deben terminar en un nodo, incluso aunque no represente ninguna sentencia procedimental.
- Regiones: las áreas delimitadas por las aristas y nodos. También se incluye el área exterior del grafo, contando como una región más.

1.1.7 Artefactos de Prueba

Durante el proceso de prueba se genera información que es usada y modificada durante el proceso de desarrollo del software, esta información es considerada un artefacto.

- Plan de prueba: es el documento rector del proceso de prueba, un buen plan es el principal factor de éxito para la puesta en práctica de una estrategia de pruebas que permita entregar un software de mejor nivel. Es el artefacto que permite definir el tipo de prueba a aplicar al producto, definiendo el alcance, los recursos, el tiempo y las personas necesarias para realizar el proceso de pruebas.
- Casos de prueba: es un conjunto de entradas de pruebas, condiciones de ejecución y resultados esperados desarrollados para cumplir un objetivo en particular o una función esperada. El diseño de casos de prueba es una de los pasos más importantes porque los mismos constituyen la fuente para evaluar los resultados del software.
- Documento de no conformidades encontradas: es un documento en el cual se queda plasmado todas las no conformidades descubiertas en el sistema al que se le realizaron las pruebas.

Pruebas según el **grado** de **automatización**:

- Pruebas manuales: una prueba manual es un archivo de texto que describe la finalidad de la prueba y contiene una lista de pasos que debe seguir un evaluador.
- Pruebas automatizadas: se usa un determinado software para sistematizar las pruebas y obtener los resultados de las mismas.

1.1.8 Pruebas manuales

Las pruebas manuales es el método de pruebas de software más antiguo y riguroso. Requieren de un probador que ejecute de manera manual operaciones en la aplicación sin ayuda de herramientas de automatización.

Por lo que se recomienda que estas pruebas sean llevadas a cabo por personas observadoras, pacientes, creativas e innovadoras. Estas pruebas detectan cualquier problema relacionado con la funcionalidad del producto, especialmente defectos vinculados con la usabilidad y la interfaz de usuario.

1.1.9 Pruebas automatizadas

Es el uso por medio de una herramienta para el control de la ejecución de las pruebas, la preparación de pre-condiciones, la comparación de los resultados actuales con los que se han predecido y la realización de informes.

El empleo de software para la ejecución de pruebas puede llegar a ser un proceso complejo, sin embargo este tiene varios beneficios porque cuando es instalado ahorra el tiempo de una operación que se tendría que hacer de forma manual.

Ventajas que proporciona la automatización:

- Repetible: se puede probar cómo el software reacciona bajo ejecución repetida de las mismas operaciones.
- Programables: se puede programar las pruebas sofisticadas que ponen en evidencia la información ocultada del uso.
- Factibles: se pueden ejecutar más pruebas en menos tiempo y el número de los recursos se reduce.
- Reutilizables: se pueden reutilizar pruebas en diversas versiones.
- Software de una calidad mejor: porque permite hacer funcionar más pruebas en menos tiempo y con pocos recursos.
- Rápidas: su ejecución es perceptiblemente más rápida.

1.2 Frameworks para pruebas automatizadas

Para la realización de las pruebas de unidad, se investigaron algunos *frameworks* que utilizan C++ como lenguaje de programación.

CppUnit: es un *framework* de pruebas unitarias para el lenguaje de programación C++. La biblioteca se distribuye bajo la Licencia Pública General GNU, esta puede ser compilada para una variedad de plataformas de Interfaces de Sistemas Operativos Portables (POSIX para sus siglas en inglés), permitiendo realizar

pruebas de unidad a código desarrollado tanto en C como C++ con un mínimo de modificación de fuente. El *framework* tiene una interfaz de usuario neutral, ejecutando las pruebas en una suite.

Entre sus ventajas se encuentran la facilidad para agregar nuevas pruebas constantemente de manera sencilla. Además es modificable y portable, ya que cuenta con una versión para Windows y otra para GNU/Linux. Cuenta con manejadores de excepciones y de recuperación porque utiliza el concepto de protectores. CppUnit intenta captar e identificar las excepciones por defecto, aunque se pueden crear excepciones propias para que sean combinadas con las ya existentes. Tiene un sistema de pocas declaraciones ASSERT, incluyendo la de comparación de números de tipo punto-flotante.

Boost.Test: no es exclusivamente un *framework* de pruebas de unidad, sino que tiene otras prestaciones. No está basado en la familia XUNIT. Boost.Test es una biblioteca con una gran potencialidad, tiene gran soporte para el manejo de excepciones y declaraciones avanzadas, además de otras funcionalidades como la ayuda para el chequeo de lazos infinitos.

Mínima cantidad de código a la hora de agregar nuevas pruebas: requiere un mínimo de trabajo para adicionar nuevas pruebas.

Facilidad para modificarse y ser portable: se obtienen marcas combinadas por la misma razón que en el CppUnit. Esta biblioteca permite portabilidad y un mejor trabajo sobre Linux. Hacerles modificaciones a la Boost.Test puede convertirse en algo complejo.

Manejadores de excepciones y de recuperación: no solo se hace el manejo de excepciones de manera correcta sino que imprime información sobre ellas, captura las excepciones de Linux y tiene líneas de comandos que deshabilitan el manejo de estas, permitiendo así capturar los problemas fácilmente.

Buena funcionalidad del ASSERT: tiene declaraciones para cualquier tipo de operación deseada, ya sea de igualdad, menor que, mayor que, entre otras. Tiene ayuda para el chequeo de excepciones que se hayan lanzado. Las declaraciones acertadas correctamente imprimen el contenido de las variables que están siendo chequeadas.

Grupos de accesorios: Al crear el conjunto de objetos delegados (helpers), se requiere una cantidad determinada de declaraciones y modificaciones en el ejecutor de las pruebas.

CxxTest: este *framework* analiza el código de las pruebas implementadas por el usuario y genera a un corredor de prueba en C++ que se ejecuta directamente desde las pruebas del usuario. Por su flexibilidad es posible hacer pruebas rigurosas con código sencillo.

Mínima cantidad de código a la hora de agregar nuevas pruebas: es muy bueno. Permite pruebas sin la necesidad de declarar una clase contenedora.

Facilidad para modificarse y ser portable: CxxTest requiere el sistema más simple de características de lenguaje. No necesita bibliotecas externas. También se distribuye simplemente como sistema de archivos de cabecera, por lo que no hay necesidad de compilar en una biblioteca separada o algo similar.

Grupos de accesorios: crear los accesorios es bastante directo y apenas requiere la herencia de una clase, pudiendo crear todas las funciones que se necesiten.

Manejadores de excepciones y de recuperación: captura las excepciones e imprime la información sobre ellas, de cualquier tipo de error, aunque no captura excepciones del sistema con GNU/Linux. Se pueden volver a efectuar fácilmente las pruebas. También brinda un paquete de macros el cual permite la captura de excepciones por el usuario siempre que lo necesite.

Buena funcionalidad del ASSERT: implementa una suite de prueba completa, con funciones assert fáciles de entender, incluyendo las de manejo de excepciones, la comprobación de predicados y las relaciones arbitrarias. Incluso se pueden imprimir *warnings*, los cuales pueden ser usados para diferenciar dos partes del código llamado en una misma prueba o se pueden imprimir mensajes hechos por el usuario.

Grupo de accesorios: tiene un buen soporte de suite porque todas las pruebas se encuentran en una suite.

1.2.1 Selección del *framework*

Para seleccionar el *framework* de prueba a utilizar se realizará mediante una matriz de decisión donde se comparan los siguientes criterios de selección:

- Mínima cantidad de código a la hora de agregar nuevas pruebas: el *framework* en el cual menos se tenga que implementar será más eficiente a la hora de realizar las pruebas generando más pruebas en menos tiempo.
- Facilidad para modificarse y ser portable: no debe tener ninguna dependencia con las bibliotecas no estándar.
- Manejadores de excepciones y de recuperación: no es bueno que paren las pruebas por cierto código que fue ejecutado, porque hubo un acceso a cierta posición de memoria inválida o por una división por cero. El *framework* de prueba debe reportar las excepciones y tanta información sobre ellas como sea posible. Debe también ser posible ejecutarlo otra vez y tener la ruptura de depuración en el lugar en donde la excepción fue accionada.
- Buena funcionalidad del ASSERT: de fallar una declaración, el ASSERT debe imprimir el contenido de las variables que fueron comparadas. Debe también proporcionar un buen sistema de ASSERT, el cual proporcione casi todas las posibles comparaciones y los diferentes tipos de datos, así como comprobar si las excepciones fueron o no lanzadas y sus tipos.
- Grupos de accesorios: de esto depende el buen funcionamiento de un *framework* de prueba. Las pruebas deben formar parte de una suite o súper-clase de prueba de la cual hereden las que implementan los probadores.

1.2.2 Importancia de los criterios de selección

A cada uno de estos criterios de selección se le asigna una importancia entre los valores de 0-25.

Criterios de selección	Importancia
Mínima cantidad de código a la hora de agregar nuevas pruebas	25
Facilidad para modificarse y ser portable	10

Manejadores de excepciones y de recuperación	20
Buena funcionalidad del ASSERT	25
Grupos de accesorios	20

Tabla 1 *Importancia de los criterios de selección*

1.2.3 Alternativas

Modelo de decisión		**Alternativas**					
		CppUnit		**Boost.Test**		**CxxTest**	
Criterios	Importancia	Calificación	Puntaje	Calificación	Puntaje	Calificación	Puntaje
Mínima cantidad de código a la hora de agregar nuevas pruebas	25	4	100	3	75	5	125
Facilidad para modificarse y ser portable.	10	3	30	2	20	5	50
Manejadores de excepciones y de recuperación.	20	4	80	5	100	5	100
Buena funcionalidad del ASSERT.	25	2	50	5	125	5	125
Grupos de accesorios	20	5	100	4	80	5	100
Total	100	28	360	23	400	34	500

Tabla 2 *Cálculo de puntajes basado en criterios e importancia*

1.2.4 Selección del *framework* de prueba

CxxTest resulta ser el *framework* más indicado para la realización de las pruebas, en primer lugar porque se tuvo en cuenta los aspectos antes mencionados. En segundo lugar, con motivo a los requerimientos del ambiente del trabajo: plataforma GNU/Linux y la plataforma de software Eclipse en la cual se implementaron las clases que se desean probar, además porque está integrado al IDE de desarrollo y no posee dependencias con bibliotecas externas.

1.3 Metodologías de desarrollo de software

Las metodologías se basan en una combinación de los modelos de proceso genéricos (cascada, evolutivo, incremental). Adicionalmente una metodología debería definir con precisión los artefactos, roles y actividades involucrados, junto con prácticas y técnicas recomendadas, guías de adaptación de la metodología al proyecto, guías para uso de herramientas de apoyo. Habitualmente se utiliza el término “método” para referirse a técnicas, notaciones y guías asociadas, que son aplicables a una o algunas actividades del proceso de desarrollo.

Las metodologías de desarrollo de software surgieron a raíz de la necesidad de controlar y documentar proyectos cada vez más complejos, impulsadas principalmente por instituciones económicamente importantes y con requisitos de seguridad y fiabilidad en sus sistemas sumamente estrictos.

Las metodologías son un conjunto de procedimientos, técnicas, herramientas y un soporte documental que ayuda a los desarrolladores a realizar nuevo software (3) .

Para guiar un proceso es necesario emplear alguna metodología de desarrollo, en los sub-epígrafes siguientes se hace un análisis de tres de ellas para seleccionar la que sea más conveniente para la elaboración del trabajo.

1.3.1 Proceso Unificado de desarrollo (RUP para sus siglas en inglés)

El Proceso Unificado de desarrollo se divide en cuatro fases o etapas:

- Inicio: se describe el negocio y se delimita el proyecto describiendo sus alcances con la identificación de los casos de uso del sistema.
- Elaboración: se define la arquitectura del sistema y se obtiene una aplicación ejecutable que responde a los casos de uso que la comprometen. A pesar de que se desarrolla a profundidad una parte del sistema, las decisiones sobre la arquitectura se hacen sobre la base de la comprensión del sistema completo y los requerimientos (funcionales y no funcionales) identificados de acuerdo al alcance definido.
- Construcción: se obtiene un producto listo para su utilización que está documentado y tiene un manual de usuario. Se obtiene 1 o varios release del producto que han pasado las pruebas. Se ponen estos release a consideración de un subconjunto de usuarios
- Transición: el release ya está listo para su instalación en las condiciones reales. Puede implicar reparación de errores.

Cada una de estas etapas es desarrollada mediante el ciclo de iteraciones, la cual consiste en reproducir el ciclo de vida en cascada a menor escala. Los objetivos de una iteración se establecen en función de la evaluación de las iteraciones precedentes.

Vale mencionar que el ciclo de vida que se desarrolla por cada iteración es llevada bajo las disciplinas.

Disciplinas de desarrollo:

- Modelamiento de Negocios: describe los procesos de negocio, identificando quiénes participan y las actividades que requieren automatización.
- Requerimientos: define qué es lo que el sistema debe hacer, para lo cual se identifican las funcionalidades requeridas y las restricciones que se imponen.
- Análisis y Diseño: describe cómo el sistema será realizado a partir de la funcionalidad previstas y

  las restricciones impuestas, por lo que indica con precisión lo que se debe programar.
- Implementación: define cómo se organizan las clases y objetos en componentes, cuáles nodos se utilizarán y la ubicación en ellos de los componentes y la estructura de capas de la aplicación.
- Pruebas: busca los defectos a los largo del ciclo de vida.
- Despliegue: produce *release* del producto y realiza actividades (empaque, instalación, asistencia a usuarios) para entregar el software a los usuarios finales.

Disciplinas de apoyo:

- Administración de proyecto: involucra actividades con las que se busca producir un producto que satisfaga las necesidades de los clientes.
- Administración de configuración y cambio: describe cómo controlar los elementos producidos por todos los integrantes del equipo de proyecto en cuanto a: utilización/actualización concurrente de elementos, control de versiones, etc.
- Ambiente: contiene actividades que describen los procesos y herramientas que brindarán soporte al el equipo de trabajo del proyecto; así como el procedimiento para implementar el proceso en una organización.

Una particularidad de esta metodología es que en cada ciclo de iteración, se hace exigente el uso de artefactos, siendo por este motivo, una de las metodologías más importantes para alcanzar un grado de certificación en el desarrollo del software. Es más adaptable para proyectos de largo plazo.

1.3.2 OpenUP

Es clasificada como una metodología ágil, se construyó sobre una donación realizada por IBM del Basic Proceso Unificado y entregada a Eclipse a fines de 2005, renombrado como OpenUP en 2006.

OpenUP es un marco de trabajo para procesos de desarrollo de software de código abierto. Es un proceso modelo y extensible, dirigido a gestión y desarrollo de proyectos de software basados en desarrollo iterativo, ágil e incremental; y es aplicable a un conjunto amplio de plataformas y aplicaciones de desarrollo.

Este proceso de desarrollo unificado está basado en proceso unificado Rational (RUP por sus siglas en inglés), desarrollado por IBM.

Está caracterizado por cuatro **principios básicos** interrelacionados (11):

- Colaboración para unificar intereses y compartir conocimientos.

- Equilibrio de prioridades competentes a maximizar el valor de los involucrados con el resultado del proyecto.
- Enfoque en la articulación de la arquitectura.
- Desarrollo continuo para obtener realimentación y realizar las mejoras respectivas. OpenUP se centra en articular la arquitectura para facilitar la colaboración técnica, reducir el riesgo y minimizar el sobreesfuerzo de desarrollo.

OpenUP desarrolla un ciclo de vida interactivo que mitiga el riesgo a tiempo y ofrece resultados en curso al cliente del proyecto.

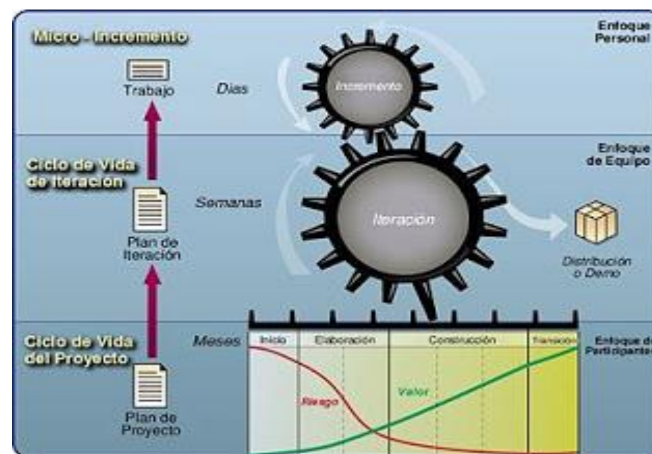

Figura 2 *Metodología OpenUP*

Esta metodología presenta algunos **beneficios** los cuales se mencionan a continuación (12):

- Es apropiado para proyectos pequeños y de bajos recursos, permitiendo disminuir las probabilidades de fracaso e incrementa las probabilidades de éxito.
- Permite detectar errores tempranos a través de un ciclo iterativo.
- Evita la elaboración de documentación, diagramas e iteraciones innecesarios requeridos en la metodología RUP.
- Por ser una metodología ágil tiene un enfoque centrado al cliente y con iteraciones cortas.

Disciplina:

- Arquitectura: crea una arquitectura de los requisitos de gran importancia arquitectónica y se basa en la disciplina de Desarrollo. Su propósito es desarrollar una arquitectura robusta para el sistema.
- Configuración y administración del cambio: controla los cambios en artefactos. El propósito de esta disciplina es mantener un conjunto consistente de productos de trabajo a medida que evolucionan, mantiene compilaciones coherentes del software, proporciona un medio eficiente para adaptarse a los cambios, los problemas y los datos para medir el progreso.
- Diseño: diseña una solución técnica que se ajuste a la arquitectura y es compatible con los requisitos. El propósito de esta disciplina es transformar los requisitos en un diseño del sistema, adaptar el diseño

para que coincida con el entorno de ejecución, construir el sistema de forma incremental y verificar que las unidades técnicas utilizadas para construir el sistema funcionen como se especifica.

- Gestión de Proyectos**:** entrena, facilita y apoya al equipo, ayudando a hacer frente a los riesgos y los obstáculos encontrados en la construcción de software. Su propósito es estimular la colaboración en equipo en la creación de planes a largo y corto plazo del proyecto, se centra en el equipo, ayuda a crear un entorno de trabajo eficaz para maximizar la productividad de los equipos, mantiene las partes interesadas y el equipo que informó sobre los avances del proyecto y proporciona un marco para gestionar los riesgos del proyecto.
- Requerimientos: analiza, especifica, válida y gestiona los requisitos para el sistema a desarrollar. El propósito de esta disciplina es entender el problema a ser resuelto, las necesidades de los interesados directos, define los requisitos para la solución (lo que el sistema debe hacer), los limites (alcance del sistema), identifica interfaces externas para el sistema, las limitaciones técnicas de la solución, proporciona las bases para la planificación de iteraciones y la estimación de costos.
- Prueba: facilita comentarios sobre la maduración del sistema mediante el diseño, implementación, ejecución y evaluación de las pruebas. Su propósito es proporcionar retroalimentación, asegurar que los cambios en el sistema no introducen nuevos defectos.

1.3.3 Programación Extrema (XP para sus siglas en inglés)

Es una de las metodologías de desarrollo de software más exitosas en la actualidad utilizada para proyectos (pequeños, grandes, medianos) con requisitos imprecisos, muy cambiantes y donde existe un alto riesgo técnico. La metodología consiste en una programación rápida o extrema, cuya particularidad es tener como parte del equipo, al usuario final, pues es uno de los requisitos para llegar al éxito del proyecto.

XP es una metodología ágil centrada en potenciar relaciones interpersonales como clave para el éxito en desarrollo de software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y propiciando un buen clima de trabajo. Se basa en realimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios (16).

Las **desventaja** de esta metodología es que es recomendable emplearla solo en proyectos a corto plazo, imposible prever todo antes de programar y demasiado costoso.

La metodología se basa en (14):

- Pruebas Unitarias: son aquellas pruebas realizadas a los principales procesos, de tal manera que adelantándonos en algo hacia el futuro, podamos hacer pruebas de las fallas que pudieran ocurrir. Es como si nos adelantáramos a obtener los posibles errores.
- Refabricación: se basa en la reutilización de código, para lo cual se crean patrones o modelos estándares, siendo más flexible al cambio.

- Programación en pares: consiste en que dos desarrolladores participen en un proyecto en una misma estación de trabajo. Cada miembro lleva a cabo la acción que el otro no está haciendo en ese momento.

OpenUP resulta ser la metodología más indicada para el proceso de desarrollo de software porque al ser una versión simplificada de RUP puede ser adaptada y generar solo los artefactos imprescindibles para la elaboración de la aplicación. Esta metodología intenta incluir dentro de su proceso desarrollo únicamente el contenido imprescindible para garantizar un proceso de desarrollo de calidad. Por este motivo hay ciertas disciplinas de RUP que han sido omitidas, como por ejemplo Modelamiento de negocio, Despliegue y Ambiente que no se incluyen porque fueron consideradas no necesarias para proyectos pequeños.

1.4 Herramientas de desarrollo

Como herramienta de modelado se empleó Visual Paradigm, creado para asistir el proceso de Ingeniería de software, este se encuentra basado en UML y soporta el ciclo de vida completo del proceso de desarrollo del software: análisis y diseño orientados a objetos, construcción, pruebas y despliegue, además cuenta con más funcionalidades que las presente en Rational Rose, lo que permite agilizar considerablemente el trabajo. A continuación se describe sus principales características:

- Disponibilidad en múltiples plataformas (Windows y Linux).
- Disponibilidad de múltiples versiones, para cada necesidad.
- Presenta licencia gratuita y comercial.
- Fácil de instalar y actualizar.
- Entorno gráfico amigable para el usuario.
- Disponible en varios idiomas.
- Soporte de UML versión 2.1.

1.4.1 Entorno de Desarrollo Integrado (IDE por sus siglas en inglés)

Eclipse es un IDE de código abierto y multiplataforma. Emplea módulos para proporcionar toda su funcionalidad, a diferencia de otros entornos monolíticos donde las funcionalidades están todas incluidas, las necesite el usuario o no. Este mecanismo de módulo es una plataforma ligera para componentes de software y le permite soportar otros lenguajes de programación como son C/C++ y Python. La definición que da el proyecto Eclipse acerca de su software es: “una especie de herramienta universal, un IDE abierto y extensible para todo y nada en particular.”

1.4.2 Lenguaje de programación

El lenguaje de programación C++ fue diseñado a mediados de los años 1980 por Bjame Stroustrup. La intención de su creación fue el extender al exitoso lenguaje de programación C con mecanismo que permita la manipulación de objetos. El mismo es un lenguaje de programación que soporta la programación orientada a objetos y la programación genérica. El mismo permite un excelente control de memoria y una buena

administración de los recursos de la computadora. Dentro de las principales ventajas que presenta el lenguaje se encuentran:

- Difusión: al ser uno de los lenguajes más empleado en la actualidad, posee un gran número de usuarios y tiene una excelente bibliografía.
- Versatilidad: C++ es un lenguaje de propósito general, se puede emplear para resolver cualquier tipo de problema.
- Eficiencia: C++ es uno de los lenguajes más rápido en tiempo de ejecución.
- Herramientas: existe gran cantidad de compiladores, depuradores y bibliotecas de clases basada en este lenguaje.

El lenguaje de programación Python fue diseñado por Guido Van Rossum. Su objetivo con este lenguaje era cubrir las necesidades de un lenguaje orientado a objetos de sencillo uso que sirviese para tratar diversas tareas dentro de la programación que habitualmente se hacía en Unix usando C. Es un lenguaje multiparadigma porque soporta orientación a objetos, programación imperativa y en poca medida, programación funcional.

Este lenguaje presenta algunas características las cuales se mencionan a continuación.

- Se puede crear todo tipo de programas. No es un lenguaje creado específicamente para la web, aunque entre sus posibilidades si se encuentra el desarrollo de páginas.
- Es un lenguaje interpretado. Esto quiere decir que no necesita compilar el código fuente para poder ejecutarlo, lo que ofrece ventajas como la rapidez de desarrollo e inconvenientes como una menor velocidad.
- Dispone de un intérprete por línea de comandos en el que se pueden introducir sentencias. Cada sentencia se ejecuta y produce un resultado visible, que puede ayudar a entender mejor al lenguaje y probar los resultados de la ejecución de porciones de código rápidamente.
- Soporta orientación a objetos y ofrece en muchos casos una manera sencilla de crear programas con componentes reutilizables.
- Dispone de muchas funciones incorporadas en el propio lenguaje, para el tratamiento de string, números y archivos. Además existen librerías que se pueden exportar en los programas para tratar temas específicos como la programación de ventanas o sistemas en red o cosas tan importante como crear archivos comprimidos en Zip.

1.5 Conclusiones parciales

Como resultado de la investigación realizada en el presente capítulo se decide:

- Seleccionar OpenUP como la metodología de desarrollo a emplear, la misma genera solo los artefactos imprescindibles y realmente necesarios para la elaboración de la aplicación.
- Los lenguajes de programación más indicados para la implementación de las pruebas automatizadas son C++ y Python, el primer lenguaje es utilizado porque es el lenguaje nativo con que está

implementado el marco de prueba CxxTest y las bibliotecas SSLmm, ISecurityClient e ISecurityServer. El segundo lenguaje para realizar el generador de datos porque C++ no ofrece las ventajas de utilizar una biblioteca estandarizada con algoritmos ya implementados como la Crypto Chiper de Python.

- Se selecciona Eclipse como entorno de desarrollo integrado de código abierto y multiplataforma, además de que es compatible con el *framework* de prueba y soporta los lenguajes antes mencionados.
- Se selecciona CxxTest como *framework* de prueba de unidad, este está integrado al IDE de desarrollo y no posee dependencias con bibliotecas externas.
- Las pruebas realizadas a los distintos componentes del módulo Seguridad serían las pruebas unitarias con el método de caja blanca, este último se utilizará la técnica del camino básico, estos diseños de casos de prueba serán realizados en el capítulo siguiente.

CAPÍTULO 2: PROCESO DE PRUEBAS AUTOMATIZADAS

Introducción

En este capítulo se define el Plan de prueba para una mejor organización de las actividades donde se incluye los roles, la estrategia y los recursos necesarios para la ejecución de las pruebas, además se diseñan los casos de pruebas empleando la técnica del camino básico para los distintos componentes del módulo Seguridad.

2.1 Requisitos funcionales del módulo Seguridad

Los requisitos funcionales son capacidades que el sistema debe cumplir. A continuación se mencionan los requisitos funcionales del módulo Seguridad (31).

RF1: autenticar usuario

RF2: controlar acceso a los recursos

RF3: gestionar perfiles

- **RF3.1**: adicionar perfiles
- **RF3.2**: modificar perfiles
- **RF3.3**: eliminar perfiles

RF4: gestionar grupos

- **RF4.1**: adicionar grupos
- **RF4.2**: modificar grupos
- **RF4.3**: eliminar grupos

RF5: gestionar usuarios

- **RF5.1**: adicionar usuarios
- **RF5.2**: modificar usuarios
- **RF5.3**: eliminar usuarios

Para comprobar que los requisitos antes mencionados sean completamente satisfactorios, es necesario la realización de los casos de prueba, debe haber al menos un caso de prueba para cada requisito a menos que un requisito tenga requisitos secundarios. En caso de que el requisito secundario tenga otros requisitos secundarios, a estos se le realiza un caso de prueba. En el sub-epígrafe 2.4 se realizan los casos de prueba para comprobar cada requisito del módulo Seguridad.

2.2 Modelo de casos de usos del módulo Seguridad

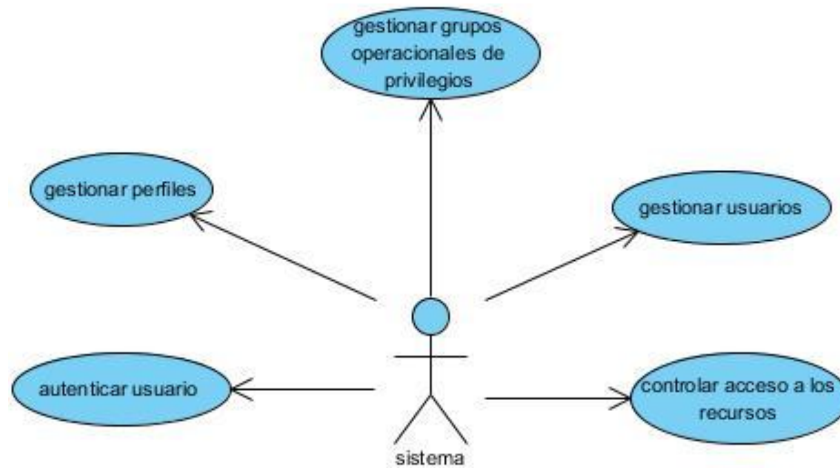

Figura 3 *Modelo de casos de uso del módulo Seguridad*

2.3 Plan de pruebas

El plan de prueba que se utiliza en el CEDIN se compone fundamentalmente por: los roles, responsabilidades, los recursos del sistema y la estrategia de prueba que se pretende aplicar. A continuación se describe el plan que guía las pruebas del módulo Seguridad.

2.3.1 Roles y responsabilidades

La ejecución de las pruebas ya sea manual o automática engloba las actividades para poder ejecutar las pruebas y cada una de esta requiere la asignación de roles con sus respectivas responsabilidades según el programa de mejora llevado a cabo en el CEDIN. Los dos roles definidos en el programa de mejora y que participan directamente con las pruebas son los siguientes:

Actividad	Rol	Cantidad	Responsabilidades
Planificar las pruebas	Diseñador de pruebas	2	Participa en la confección de la estrategia y el plan de pruebas. Identifica los métodos, las técnicas, herramientas y directrices apropiadas para implementar las pruebas necesarias.
Diseñar las pruebas	Diseñador de pruebas	2	Establece en ambiente de comprobación. Encargado de diseñar casos de pruebas para el sistema.

Implementar la prueba	Diseñador de pruebas	2	Dirige la definición del enfoque de prueba y garantiza la implementación satisfactoriamente.
Realizar las pruebas	Probador	2	Encargado de seguir los planes de pruebas. Ejecuta los casos de prueba y genera las no conformidades asociadas al mismo. Registra los resultados de las pruebas. Analiza los resultados de las pruebas realizadas.

Tabla 3 *Roles y responsabilidades*

2.3.2 Recurso necesarios para la ejecución de las pruebas

Recurso	Tipo
Servidor de Seguridad	1 procesador Intel de 1.9GHz, 1GB RAM, y un disco duro de 200 GB

Tabla 4 *Servidores*

Cantidad	1
Descripción	1 procesador Intel de 1.9GHz, 512MB RAM, y un disco duro de 100 GB
Software base	Sistema operativo Ubuntu o Debian.

Tabla 5 *PC clientes*

2.3.3 Estrategia de prueba

Alcance

Las pruebas que se le aplicarán al módulo Seguridad son las pruebas unitarias. El objetivo principal de estas pruebas es encontrar la mayor cantidad de errores en los componentes fundamentales (Cliente, Servidor y biblioteca de encriptación de datos) del módulo Seguridad. Como primer paso se realizó el Plan de prueba donde quedó plasmado como se va a desarrollar el proceso de prueba y los recursos necesarios del mismo. Además se diseñan los diseños de casos de pruebas y las mismas se realizan de forma automatizada.

2.4 Diseño de los casos de prueba

2.4.1 Diseño de los casos de pruebas de la biblioteca SSLmm

2.4.1.1 Método de base64DecodeString

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

std::string base64DecodeString (std::string text)

{

1 text = base64StringFormat (text);

2 EVP_ENCODE_CTX ectx;

2 int size = text.size ();

2 unsigned char* out = (unsigned char*) malloc (size);

2 int outlen = 0;

2 int tlen = 0;

3 EVP_DecodeInit (&ectx);

4 EVP_DecodeUpdate (&ectx, out, &outlen, (const unsigned char*) text.c_str (), text.size ());

5 tlen += outlen;

6 EVP_DecodeFinal (&ectx, out+tlen, &outlen);

7 tlen += outlen;

8 std: string data ((char*) out, tlen);

9 free (out);

10 return data;

11}

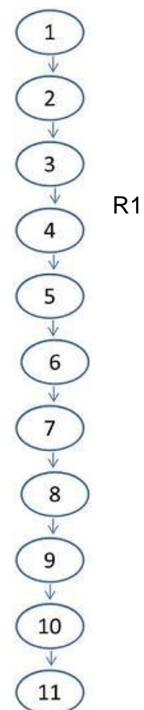

Figura 4 *Grafo del método base64DecodeString*

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1.

Paso2.2: 1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1,2,3,4,5,6,7,8,9,10,11.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	text es una cadena de texto que puede tomar cualquier valor.	El método devuelve una cadena.

Tabla 6 *Representación del caso de prueba del método base64DecodeString*

Como el método de base 64 DecodeString llama internamente al método base64StringFormat implica que también hay que aplicarle la técnica.

A los métodos EVP_ENCODE_CTX, EVP_DecodeInit, EVP_DecodeUpdate, EVP_DecodeFinal también son llamados internamente en el método base 64 DecodeString, a estos métodos no se le aplica la técnica debido que los mismos pertenecen a OpenSSL, producto liberado, validado y certificado internacionalmente por lo que se infiere un resultado satisfactorio del mismo.

Método de base64StringFormat.

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

```
std::string base64StringFormat (std::string text)
    {
    1 text = uniteLines (text);
    1 std::string str = "";
    1 int pos = 0;

    2 while (text. zize () > 0)
     {
    3 pos = text.size ();
    3 pos = pos > 64? 64: pos;
    3 str += text.substr (0, pos) + "\n";
    3 text = text.substr( pos, std::string::npos );
    }
4 return str;
5}
```

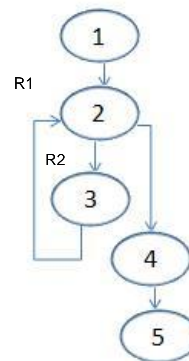

Figura 5 *Grafo del método base64*

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1: 5-5+2=2 por tanto el #regiones: 2 coincide con la complejidad ciclomática: 2.

Paso2.2: 1+1=2 por tanto el #regiones: 2 coincide con la complejidad ciclomática: 2.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1: 1,2.4, 5.

Camino2:1, 2, 3, 4,5.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	text es una cadena de texto vacía.	El método devuelve un string vacío.
Camino2	text es una cadena de texto, de longitud mayor que 0	El método devuelve una cadena

Tabla 7 *Representación de los casos de prueba del método base64StringFormat*

Como el método base64StringFormat llama internamente al método uniteLines esto implica que también hay que aplicarle la técnica del camino básico.

Método de uniteLines.

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

```
std:: string uniteLines( std::string text )
{
1 std::string str;
1 char ch;
1 unsigned int size = text.size ();

2 for (unsigned int i = 0; i < size; ++i)
{
3 ch = text[i];
4 if (ch! = '\n' && ch! = '\r’)
5 str += ch;
6}
7 return str;
8}
```

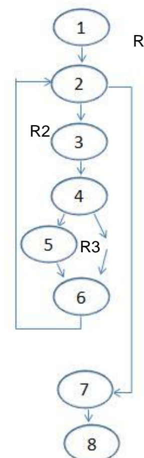

Figura 6 *Grafo del método uniteLines*

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:9-8+2=3 por tanto el #regiones: 3 coincide con la complejidad ciclomática: 3.

Paso2.2:2+1=3 por tanto el #regiones: 3 coincide con la complejidad ciclomática: 3.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1: 1, 2, 7,8

Camino2: 1, 2, 3, 4,5, 6, 2, 7,8

Camino3: 1, 2, 3, 4, 6, 2, 7,8.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	text es una cadena de texto vacía.	El método devuelve un string vacío.
Camino2	text es una cadena de texto que cumple que su longitud sea mayor que 0 y ch distinta de fin de línea y distinta de retorno de carro.	El método devuelve un string.
Camino3	text es una cadena de texto que cumple que su longitud sea mayor que 0 y el carácter ch sea fin de línea o retorno de carro.	El método devuelve un string.

Tabla 8 *Representación de los casos de prueba del método uniteLines*

2.4.1.2 Método de base64EncodeString

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado

std::string base64EncodeString (std::string text)

{

1 EVP_ENCODE_CTX ectx;

1 int size = text.size () * 2;

1 unsigned char* out = (unsigned char*) malloc (size);

1 int outlen = 0;

1 int tlen = 0;

2 EVP_EncodeInit (&ectx);

3 EVP_EncodeUpdate (&ectx, out, &outlen, (const unsigned char*)

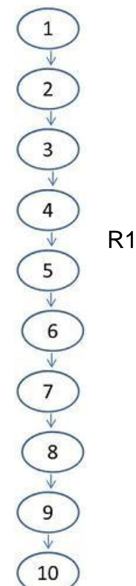

Figura 7 *Grafo del método base64EncodeString*

text.c_str (), text.size ());

4 tlen += outlen;

5 EVP_EncodeFinal (&ectx, out+tlen, &outlen);

6 tlen += outlen;

7 std::string str ((char*) out, tlen);

8 free (out);

9 return str;

10}

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:9-10+2=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1.

Paso2.2:0+1=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4, 5, 6, 7, 8, 9,10.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	text es una cadena de texto que puede tomar cualquier valor	El método devuelve un string.

Tabla 9 *Representación de los casos de prueba del método base64EncodeString*

El método de base 64Encode llama internamente a los métodos EVP_EncodeInit, EVP_EncodeUpdate, EVP_EncodeFinal, a estos métodos no se le aplica la técnica debido que los mismo pertenecen a OpenSSL, producto liberado, validado y certificado internacionalmente por lo que se infiere un resultado satisfactorio del mismo.

3.1.1.1 Método aesEncrypt

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

std:: string aesEncrypt(std::string inText, std::string password)

{

1 EVP_CIPHER_CTX ectx;

1 std:: string key = getHash(password, sha256);

1 std:: string iv = getHash(password, sha256);

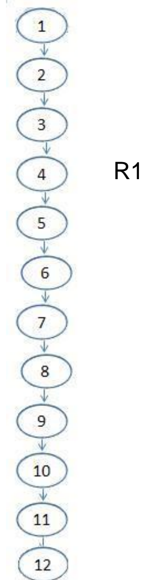

Figura 8 *Grafo del método aesEncrypt*

1 const EVP_CIPHER *cipher = EVP_aes_256_cbc ();

1 int size = inText.size () + EVP_CIPHER_block_size (cipher) - 1;

1 unsigned char* out = (unsigned char*) malloc (sizeof (char) * size);

1 int outlen = 0;

1 int tlen = 0;

2 EVP_CIPHER_CTX_init (&ectx);

3 EVP_EncryptInit (&ectx, EVP_aes_256_cbc (), (const unsigned char*) key.c_str (), (const unsigned char*) iv.c_str ());

4 EVP_EncryptUpdate (&ectx, out, &outlen, (const unsigned char*) inText.c_str (), inText.size ());

5 tlen += outlen;

6 EVP_EncryptFinal (&ectx, out+tlen, &outlen);

7 tlen += outlen;

8 EVP_CIPHER_CTX_cleanup (&ectx);

9 std:: string outText((const char*)out, tlen);

10 free (out);

11 return base64EncodeString (outText);

12}

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:11-12+2=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1.

Paso2.2:0+1=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,12.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	inText y password son cadenas de texto que pueden tomar cualquier valor	El método devuelve una cadena encriptado.

Tabla 10 *Representación de los casos de prueba del método aesEncrypt*

El método de base aesEncrypt llama internamente a los métodos EVP_EncryptInit, EVP_EncodeUpdate, EVP_EncryptFinal, a estos métodos no se le aplica la técnica debido que los mismo pertenecen a OpenSSL, producto liberado, validado y certificado internacionalmente por lo que se infiere un resultado satisfactorio del mismo.

3.1.1.2 Método aesDecrypt

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

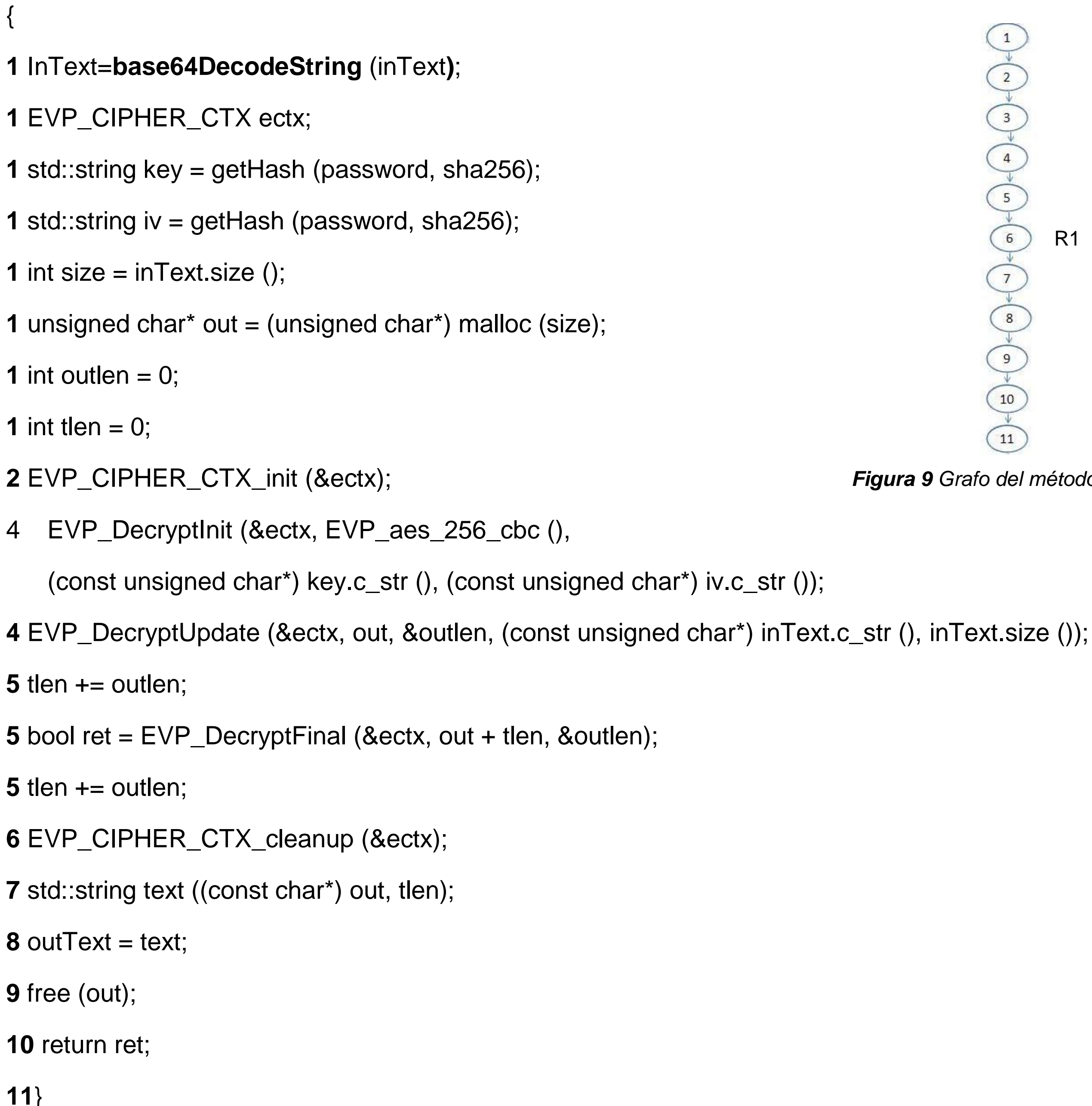

Figura 9 *Grafo del método aesDecrypt*

```
bool aesDecrypt(std::string inText, std::string password, std::string &outText )
{
1 InText=base64DecodeString (inText);
1 EVP_CIPHER_CTX ectx;
1 std::string key = getHash (password, sha256);
1 std::string iv = getHash (password, sha256);
1 int size = inText.size ();
1 unsigned char* out = (unsigned char*) malloc (size);
1 int outlen = 0;
1 int tlen = 0;
2 EVP_CIPHER_CTX_init (&ectx);
4   EVP_DecryptInit (&ectx, EVP_aes_256_cbc (),
    (const unsigned char*) key.c_str (), (const unsigned char*) iv.c_str ());
4 EVP_DecryptUpdate (&ectx, out, &outlen, (const unsigned char*) inText.c_str (), inText.size ());
5 tlen += outlen;
5 bool ret = EVP_DecryptFinal (&ectx, out + tlen, &outlen);
5 tlen += outlen;
6 EVP_CIPHER_CTX_cleanup (&ectx);
7 std::string text ((const char*) out, tlen);
8 outText = text;
9 free (out);
10 return ret;
11}
```

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:10-11+2=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1

Paso2.2:0+1=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4, 5, 6, 7, 8, 9, 10,11.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	inText y password son cadenas de texto que pueden tomar cualquier valor.	El método devuelve falso

***Tabla 11** Representación de los casos de prueba del método aesDecrypt*

El método aesDecrypt llama internamente al método base64DecodeString implica que a este método hay que aplicarle la técnica del camino básico. Ir a 2.4.1.2.

Los siguientes métodos EVP_DecryptInit, EVP_DecryptUpdate, EVP_DecryptFinal son llamados internamente en el método base64DecodeString, pero los mismos no se le aplica la técnica debido que pertenecen a OpenSSL, producto liberado, validado y certificado internacionalmente por lo que se infiere un resultado satisfactorio del mismo.

4.1.1.1 Método getHash

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

```
std:: string getHash( std::string text, Hash hash )
{
        1 const EVP_MD *type;

        2 switch (hash)
        {
         3 case md5:
            {
                type = EVP_md5();
                break;
            }
         4 case sha1:
            {
```

```
                type = EVP_sha1();
                break;
             }
          5 case sha256:
            {
                type = EVP_sha256();
                break;
             }
           6 case sha512:
            {
                type = EVP_sha512();
                break;
            }
        }
```

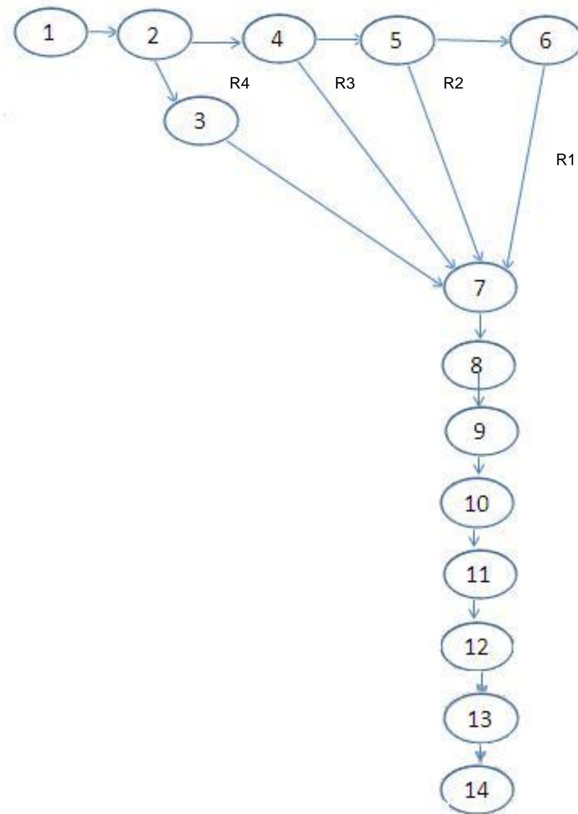

Figura 10 *Grafo del método getHash*

7 EVP_MD_CTX mdctx;

8 unsigned char md_value [EVP_MAX_MD_SIZE];

8 unsigned int md_len;

9 EVP_DigestInit (&mdctx, type);

10 EVP_DigestUpdate (&mdctx, text.c_str (), text.size ());

11 EVP_DigestFinal_ex (&mdctx, md_value, &md_len);

12 EVP_MD_CTX_cleanup (&mdctx);

13 return **digestToHexString** (md_value, md_len);

14}

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:16-14+2=4 por tanto el #regiones: 4 coincide con la complejidad ciclomática: 4

Paso2.2:3+1=4 por tanto el #regiones: 4 coincide con la complejidad ciclomática: 4.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1,2,3,7,8,9,10,11,12,13,14.

Camino2:1, 2, 4, 7,8, 9, 10, 11, 12, 13,14.

Camino3:1, 2, 4, 5, 7,8, 9, 10, 11, 12, 13,14.

Camino4:1, 2, 4, 5, 6, 7,8, 9, 10, 11, 12, 13,14.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	hash es una variable Hash que toma valor md5.	Devuelve el cálculo del hash en modo md5 de la cadena.
Camino2	hash es una variable Hash que toma valor sha1.	Devuelve el cálculo del hash en modo sha1 de la cadena.
Camino3	hash es una variable Hash que toma valor sha256.	Devuelve el cálculo del hash en modo sha256 de la cadena.
Camino4	hash es una variable Hash que toma valor sha512.	Devuelve el cálculo del hash en modo sha512 de la cadena.

***Tabla 12** Representación de los casos de prueba del método getHash*

Como el método getHash llama internamente al método digestToHexString, implica que se le tiene que aplicar la técnica del camino básico.

Método digestToHexString.

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado

std:: string digestToHexString(const unsigned char *digest, unsigned int len)

{

1 char* hexstring = (char*) malloc (len * 2 + 1);

2 len = len > 510? 510: len;   **3**.len=510. **4**. len=len.

5 for (unsigned int i = 0; i < len; ++i)

6 sprintf (&hexstring [2 * i], "%02x", digest[i]);

7 std:: string hex (hexstring);

7 free (hexstring);

8 return hex;

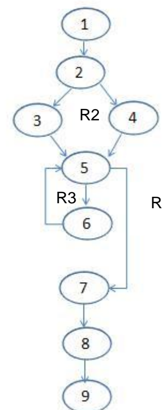

***Figura 11** Grafo del método digestToHexString*

9}

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:10-9+2=3 por tanto el #regiones: 3 coincide con la complejidad ciclomática: 3

Paso2.2:2+1=3 por tanto el #regiones: 3 coincide con la complejidad ciclomática: 3.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 5, 6, 5, 7, 8,9.

Camino2:1, 2, 4, 5, 6, 5, 7, 8,9

Camino3:1, 2, 4, 5, 7, 8,9.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	len tiene que tomar valor entero que sea mayor que 510	Convierte un tipo de datos unsigned char* a string y retorna el mismo convertido.
Camino2	len tiene que tomar valor entero <=510 y ser mayor que 0.	Convierte un tipo de datos unsigned char* a string y retorna el mismo convertido.
Camino3	len toma valor 0.	Devuelve una cadena vacía.

Tabla 13 *Representación de los casos de prueba del método digestToHexString*

4.1.1.2 Método makeX509

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado

```
std::string makeX509 (std::string caPrivateKey, X509mm *x509mm)
{
1 EVP_PKEY* privateKey = strToEVP_PKEY (caPrivateKey);
2 if (! privateKey)
{
3 showMessage->exec (“Error creando certificado CA", “Llave privada no válida",
__FILE__, __LINE__);
4 return "";
```

}

5 X509 *certificate = X509_new ();

6 X509_set_version (certificate, 3);

7 ASN1_INTEGER_set (X509_get_serialNumber (certificate), x509mm->getSerial ());

X509_set_pubkey (certificate, privateKey);

8 X509_gmtime_adj (X509_get_notBefore (certificate), (long) 60 *

60 * 24 * x509mm->getNotBefore ());

9 X509_gmtime_adj (X509_get_notAfter (certificate),

(long) 60 * 60 * 24 * x509mm->getNotAfter ());

10 X509_NAME *name;

10 name = X509_NAME_new ();

11 X509_NAME_add_entry_by_txt (name, "CN", MBSTRING_ASC,

reinterpret_cast<const unsigned char*>(x509mm->getSubject ()

-> GetCommonName ().c_str ()),-1, -1, 0);

12 X509_NAME_add_entry_by_txt (name, "O", MBSTRING_ASC,

reinterpret_cast<const unsigned char *>(x509mm->getSubject ()

–>getOrganization ().c_str ()),-1, -1, 0);

13 X509_NAME_add_entry_by_txt (name, "OU", MBSTRING_ASC,

reinterpret_cast<const unsigned char *>(x509mm->getSubject ()

->getOrganizationalUnit ().c_str ()),-1, -1, 0);

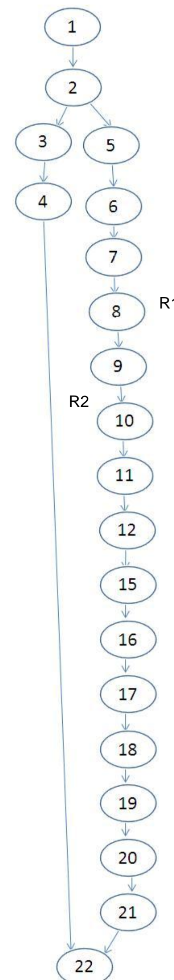

Figura 12 *Grafo del método makex509*

14 X509_NAME_add_entry_by_txt (name, "ST", MBSTRING_ASC, reinterpret_cast<const unsigned char *>(x509mm->getSubject () ->getState ().c_str ()), -1, 0);

15 X509_set_issuer_name (certificate, name);

16 X509_set_subject_name (certificate, name);

17 X509_sign (certificate, privateKey, EVP_md5 ());

18 std::string x509String = x509ToStr (certificate);

19 EVP_PKEY_free (privateKey);

20 X509_free (certificate);

21 return x509String;

} **22**

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:22-22+2=2 por tanto el #regiones: 2 coincide con la complejidad ciclomática: 2

Paso2.2:1+1=3 por tanto el #regiones: 2 coincide con la complejidad ciclomática: 2.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4,22.

Camino2:1,2,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	privateKey es una llave privada no válida.	El método retorna un string vacío
Camino2	privateKey es una llave privada válida.	Se crea el certificado y se devuelve el mismo.

Tabla 14 *Representación de los casos de prueba del método makeX509*

Como el método makeX509 llama internamente el método strToEVP_PKEY implica que hay que aplicarle la técnica.

Método strToEVP_PKEY.

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado

EVP_PKEY * strToEVP_PKEY (std::string text)

{

1 EVP_PKEY* privateKey = 0;

2 BIO* bio = BIO_new (BIO_s_mem ());

3 BIO_write (bio, text.c_str (), text.size ());

4 PEM_read_bio_PrivateKey (bio, &privateKey, 0, 0);

5 BIO_free (bio);

6 return privateKey;

} **7**

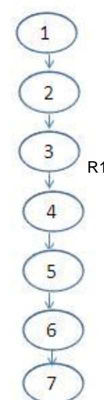

Figura 13 *Grafo del método strToEVP_PKEY*

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:6-7+2=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1

Paso2.2:0+1=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4, 5, 6,7.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	text es una cadena con elementos.	El método se encarga de convertir text a una llave (EVP_PKEY) y retorna esa llave.

Tabla 15 *Representación de los casos de prueba del método strToEVP_PKEY*

4.1.1.3 Método saveCertificate

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado

```
bool saveCertificate( std::string certificate, std::string path )
{
1 bool ret = false;
1 X509* x509 = strToX509 (certificate);
2 if (! x509)
{
3 showMessage->exec ("Error salvando certificado", "Certificado no válido", __FILE__, __LINE__);
4 goto end1;
}
5 FILE *file;
6 file = fopen (path.c_str (), "w");
7 if (! file)
{
8 showMessage->exec ("Error salvando certificado", "Creando fichero",
__FILE__, __LINE__);
9 goto end2;
}
10 ret = PEM_write_X509 (file, x509);
```

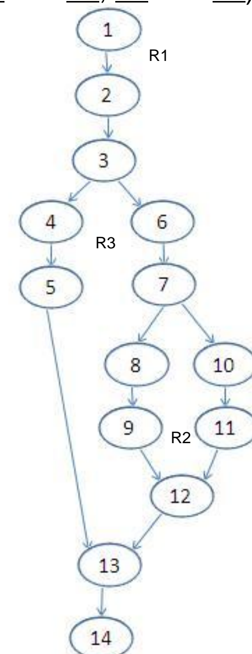

Figura 14 *Grafo del método saveCertificate*

11 fclose (file);

end2:

12 X509_free(x509);

end1:

13 return ret;

} **14**

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:15-14+2=3 por tanto el #regiones: 3 coincide con la complejidad ciclomática: 3

Paso2.2:2+1=3 por tanto el #regiones: 3 no coincide con la complejidad ciclomática: 3.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4, 13,14.

Camino2:1, 2, 3, 6, 7, 8, 9, 12, 13,14.

Camino3:1, 2, 3, 6, 7, 10, 11, 12, 13,14.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	x509 es un certificado no válido.	El método retorna falso
Camino2	x509 es un certificado válido. file es una referencia no válida a un archivo	El método retorna falso.
Camino3	x509 es un certificado válido. file es una referencia válida a un archivo	El método retorna verdadero

Tabla 16 *Representación de los casos de prueba del método saveCertificate*

Como este método llama internamente a strToX509 implica que hay que aplicarle la técnica.

Método strToX509

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado

X509 * strToX509 (std::string text)

{

1 X509* x509 = 0;

1 BIO *bio = BIO_new (BIO_s_mem ());

2 BIO_write (bio, text.c_str (), text.size ());

3 PEM_read_bio_X509 (bio, &x509, 0, 0);

4 BIO_free (bio);

5 return x509;

} **6**

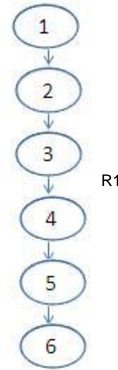

Figura 15 *Grafo del método strToX509*

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:5-6+2=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1

Paso2.2:0+1=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1: 1, 2, 3, 4, 5,6.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	text es una cadena con elementos	El método se encarga de convertir text en un x509 y devuelve el mismo.

Tabla 17 *Representación de los casos de prueba del método strToX509*

4.1.1.4 Método LoadCertificate

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

std:: string loadcertificate(std::string path)

{

1 FILE *file = fopen (path.c_str (), "r");

2 if (! file)

{

2 ShowMessage->exec (“Error cargando certificado”, “Abriendo fichero",
__FILE__, __LINE__);

4 return "";

}

5 X509 *certificate = X509_new ();

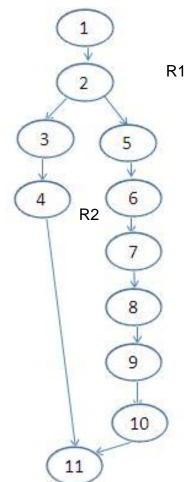

Figura 16 *Grafo del método loadcertificate*

6 PEM_read_X509 (file, &certificate, 0, 0);

7 std::string x509String = x509ToStr (certificate);

8 fclose (file);

9 X509_free (certificate);

10 return x509String;

} **11**

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:11-11+2=2 por tanto el #regiones: 2 coincide con la complejidad ciclomática: 2

Paso2.2:1+1=2 por tanto el #regiones: 2 coincide con la complejidad ciclomática: 2.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4,11.

Camino2:1, 2, 5, 6, 7, 8, 9, 10,11.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	file es una referencia no válida a un archivo	El método retorna un string vacío.
Camino2	file es una referencia válida a un archivo	El método devuelve el certificado x509 convertido a string.

Tabla 18 *Representación de los casos de prueba del método loadcertificate*

Como este método llama internamente al método x509ToStr, implica que a este método se le aplica la técnica.

Método x509ToStr:

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado

std::string x509ToStr(X509 *x509)

{

1 BIO * bio = BIO_new (BIO_s_mem ());

2 PEM_write_bio_X509 (bio, x509);

3 std::string x509String = bioToStr (bio);

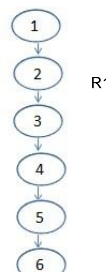

Figura 17 *Grafo del método x509ToStr*

4 BIO_free (bio);

5 return x509String;

} **6**

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:5-6+2=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1

Paso2.2:0+1=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4, 5,6.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	x509 es un certificado	El método se encarga de convertir x509 a un string y retorna el mismo.

Tabla 19 *Representación de los casos de prueba del método x509ToStr*

4.1.1.5 Método makePrivateKey

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

std:: string makePrivateKey(int bits)

{

1 EVP_PKEY *privateKey = EVP_PKEY_new ();

2 RSA *rsa = RSA_generate_key (bits, RSA_F4, callBack, NULL);

3 EVP_PKEY_set1_RSA (privateKey, rsa);

4 std::string pkString = **EVP_PKEYToStr** (privateKey);

5 EVP_PKEY_free (privateKey);

6 RSA_free (rsa);

7 return pkString;

} **8**

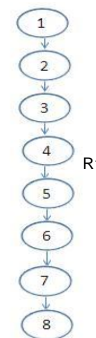

Figura 18 *Grafo del método makePrivateKey*

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:7-8+2=1por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1

Paso2.2:0+1=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4, 5, 6, 7,8.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	bits es un número mayor que cero.	El método se encarga de crear una llave con la longitud bits, convertirla a un string y retornarla la misma.

Tabla 20 *Representación de los casos de prueba del método makePrivateKey*

Como el makePrivateKey llama internamente al método EVP_PKEYToStr, implica que hay que aplicarle la técnica.

Método EVP_PKEYToStr:

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

Std::string EVP_PKEYToStr (EVP_PKEY *privateKey)

{

1 BIO* bio = BIO_new (BIO_s_mem ());

2 PEM_write_bio_PrivateKey (bio, privateKey, 0, 0, 0, 0, 0);

3 std::string pkString = bioToStr (bio);

4 BIO_free (bio);

5 return pkString;

}**6**

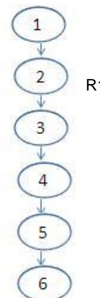

Figura 19 *Grafo del método EVP_PKEYToStr*

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:5-6+2=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1

Paso2.2:0+1=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4, 5,6

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado

Camino1	privateKey es una llave válida.	Este método se encarga de convertir privateKey a un string y retornar la llave convertida.

***Tabla 21** Representación de los casos de prueba del método EVP_PKEYToStr*

4.1.1.6 Método savePrivateKey

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

bool savePrivateKey(std::string privateKey, std::string path)

{

1 EVP_PKEY* pKey = **strToEVP_PKEY** (privateKey);

2 if (! pKey)

{

3 showMessage->exec (“Error salvando llave privada", "Llave privada no válida", __FILE__, __LINE__);

4 return false;

}

5 FILE *file = fopen (path.c_str (), "w”);

6 if (! file)

{

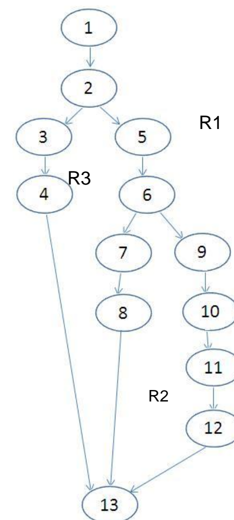

***Figura 20** Grafo del método savePrivateKey*

7 showMessage->exec (“Error salvando llave privada", "Creando fichero", __FILE__, __LINE__);

8 return false;

}

9 bool ret = PEM_write_PrivateKey (file, pKey, 0, 0, 0, 0, 0);

10 fclose (file);

11 EVP_PKEY_free (pKey);

12 return ret;

} **13**

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:14-13+2=3 por tanto el #regiones: 3 coincide con la complejidad ciclomática: 3.

Paso2.2:2+1=3 por tanto el #regiones: 3 coincide con la complejidad ciclomática: 3.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4,13.

Camino2:1, 2, 5, 6, 7, 8,13.

Camino3:1, 2, 5, 6, 9, 10, 11, 12,13.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	pKey es una llave privada no válida	El método retorna falso
Camino2	pKey es una llave privada válida. file es una referencia no válida.	El método retorna falso.
Camino3	pKey es una llave privada válida. file es una referencia válida. Se guarda pKey en file.	El método retorna verdadero

Tabla 22 *Representación de los casos de prueba del método savePrivateKey*

Como el método savePrivateKey llama internamente al método strToEVP_PKEY, implica que a este método se le aplica la técnica. Ir a strToEVP_PKEY

4.1.1.7 Método loadPrivateKey

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado

std:: string loadPrivateKey (std::string path)

{

1 FILE *file = fopen (path.c_str (), "r");

2 if (! file)

{

3 showMessage->exec (“Error cargando llave privada", “Abriendo fichero", __FILE__, __LINE__);

4 return "";

}

5 EVP_PKEY* pKey = EVP_PKEY_new ();

6 PEM_read_PrivateKey (file, &pKey, 0, 0);

7 fclose (file);

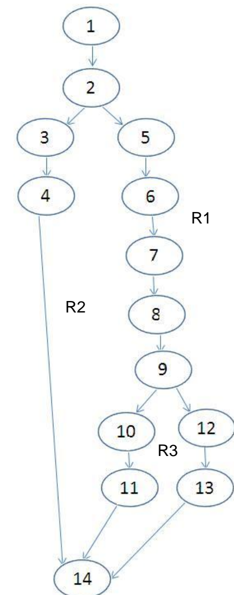

Figura 21 *Grafo del método loadPrivateKey*

8 std::string pkString = **EVP_PKEYToStr** (pKey);

9 if (pkString.empty ())

{

10 showMessage->exec (“Error cargando llave privada", "Llave privada no válida",

 __FILE__, __LINE__);

11 return "";

}

12 EVP_PKEY_free (pKey);

13 return pkString;

} **14**

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:15-14+2=3 por tanto el #regiones: 3 coincide con la complejidad ciclomática: 3.

Paso2.2:2+1=3 por tanto el #regiones: 3 coincide con la complejidad ciclomática: 3.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4,14.

Camino2:1, 2, 5, 6, 7, 8, 9, 10, 11,14.

Camino3: 1, 2, 5, 6, 7, 8, 9, 12, 13,14.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	file es una referencia no válida a un archivo.	El método retorna un string vacío.
Camino2	file es una referencia válida. pkString es una llave privada que se encuentra en file.	El método retorna un string vacío.
Camino3	file es una referencia válida. pkString es una llave privada válida que se encuentra en file	El método retorna pkString.

Tabla 23 *Representación de los casos de prueba del método loadPrivateKey*

El método loadPrivateKey llama internamente al método EVP_PKEYToStr, implica que hay que aplicar la técnica. Ir a EVP_PKEYToStr

4.1.2 Diseño de los casos de pruebas de la biblioteca ISecurity

4.1.2.1 Método ClientAuthentication

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

bool ClientAuthentication**::**exec(std::string nick, std::string password) const

{

1 showMessage->exec (“Autenticando usuario", nick, __FILE__, __LINE__);

2 DataTypes::SecurityTicketsMap *securityTicketsMap;

3 if (! iComm->**clientAuthentication** (nick, securityTicketsMap))

{

4 showMessage->exec (“Error autenticando usuario", "Usuario incorrecto.”

__FILE__, __LINE__);

5 return false;

}

6 modelManager->setSecurityTicketsMap (securityTicketsMap);

7 password = SSLmm::getHash (nick + password, SSLmm::md5);

8 bool serviceCount = false;

9 while (securityTicketsMap->next ())

{

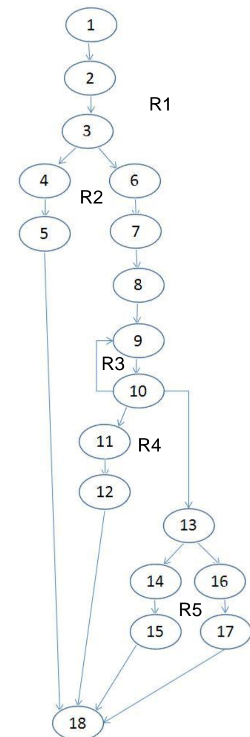

Figura 22 *Grafo del método ClientAuthentication*

10 if (! securityTicketsMap->current ().second->securityForClientTicket->decrypt (password))

{

11 showMessage->exec (“Error autenticando usuario”, “Contraseña incorrecta.” __FILE__, __LINE__);

12 return false;

}

}

13 if (securityTicketsMap->empty ())

{

14 showMessage->exec (“Error autenticando usuario", “No existen servicios registrados en el servidor de Seguridad.", __FILE__, __LINE__);

15 return false;

}

16 showMessage->exec (“Usuario autenticado correctamente", nick, __FILE__, __LINE__);

17 return true;

} **18**

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:21-18+2=5 por tanto el #regiones: 5 coincide con la complejidad ciclomática: 5.

Paso2.2:4+1=5 por tanto el #regiones: 5 coincide con la complejidad ciclomática: 5.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4, 5,18

Camino2:1,2,3,6,7,8,9,13,14,15,18.

Camino3:1,2,3,6,7,8,9,10,11,12,18.

Camino4:1,2,6,7,8,9,10,9,10,9,13,14,15,18.

Camino5:1,2,3,6,7,8,9,10,9,13,16,17,18.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	iComm no es válido.	El método devuelve falso.
Camino2	iComm es válido. securityTicketsMap es vacío.	El método devuelve falso.
Camino3	iComm es válido. Entre securityTicketsMap y password no existe correspondencia.	El método devuelve falso.
Camino4	iComm es válido. Entre securityTicketsMap y password existe correspondencia. securityTicketsMap es vacío.	El método devuelve falso.
Camino5	iComm es válido. Entre securityTicketsMap y password existe	El método devuelve verdadero.

	correspondencia. securityTicketsMap no es vacío.	

Tabla 24 *Representación de los casos de prueba del método ClientAuthentication*

El método ClientAuthentication llama internamente al método ClientAuthentication de la clase iComm, implica que hay que hacerle la técnica.

Método ClientAuthentication de iComm:

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado

bool IComm::clientAuthentication(std::string nick, DataTypes::SecurityTicketsMap *&securityTicketsMap)

{

1 std::string result;

2 if (! sendSecurityData (2, nick, result))

3 return false;

4 DataTypes::ClientAuthentication clientAuthentication;

5 try

{

6 clientAuthentication.in (result);

}

7 catch (...)

{

8 showMessage->exec (“Error autenticando usuario”, “Formato de serialización dañado.” __FILE__, __LINE__);

9 return false;

}

10 securityTicketsMap = clientAuthentication.attr2;

11 return clientAuthentication.attr1;

} **12**

} **18**

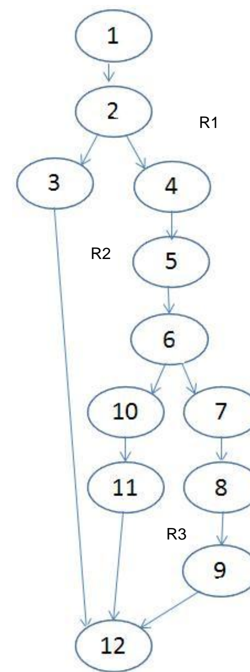

Figura 23 *Grafo del método clientAuthentication*

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:13-12+2=3 por tanto el #regiones: 3 coincide con la complejidad ciclomática: 3.

Paso2.2:2+1=3 por tanto el #regiones: 3 coincide con la complejidad ciclomática: 3.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3,12.

Camino2:1, 2, 4, 5, 6, 7, 8, 9,12.

Camino3:1, 2, 4, 5, 6, 10, 11,12.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	nick es válido.	El método retorna falso.
Camino2	result es un formato dañado.	El método retorna falso.
Camino3	securityTicketsMap toma el mapa de tikets y clientAuthentication.attr1 es true.	El método retorna verdadero.

Tabla 25 *Representación de los casos de prueba del método ClientAuthentication*

El método clientAuthentication llama internamente al método sendSecurityData, implica que hay que aplicarle la técnica.

Método sendSecurityData:

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado

int IComm::sendSecurityData(int method, std::string data, std::string& result)

{

1 int ret;

2 try

{

3 ret = client->sendSecurityData (method, data, result);

}

4 catch (...)

{

5 showMessage->exec (“Error de conexión con el servidor.”);

6 return false;

}

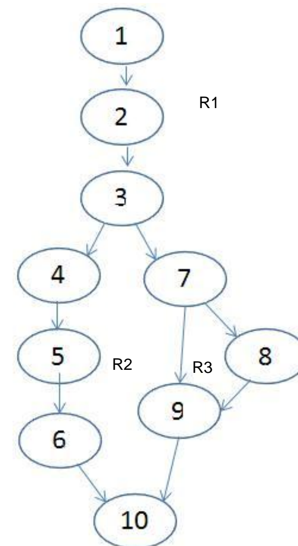

Figura 24 *Grafo del método sendSecurityData*

7 if (ret)

8 showMessage->exec (“Error de conexión con el servidor.”);

9 return !ret;

} **10**

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:11-10+2=3 por tanto el #regiones: 3 coincide con la complejidad ciclomática: 3.

Paso2.2:2+1=3 por tanto el #regiones: 3 coincide con la complejidad ciclomática: 3.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 7, 8, 9,10.

Camino2:1, 2, 3, 7, 9,10.

Camino3:1, 2, 3, 4, 5, 6,10.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	ret es igual a 1.	El método retorna 0.
Camino2	ret es igual a 0.	El método retorna 1.
Camino3	ret es vacío.	El método retorna falso.

Tabla 26 *Representación de los casos de prueba del método ClientAuthentication*

4.1.2.2 Método GetTicketsForService

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

std::string GetTicketsForService::exec (DataTypes::ServiceId serviceId) const

{

1 showMessage->exec (“Solicitando tickets para el servicio",

boost::lexical_cast<std::string> (serviceId), __FILE__, __LINE__);

2 DataTypes::SecurityTicketsMap &securityTicketsMap = *modelManager->

getSecurityTicketsMap ();

3 const DataTypes::SecurityTickets *securityTickets = securityTicketsMap.

getSecurityTickets (serviceId);

4 if (! securityTickets)

{

5 showMessage->exec (“Error obteniendo tickets para el servicio",

"El servicio no se encuentra registrado en el servidor de Seguridad.", __FILE__, __LINE__);

6 return "";

}

7 DataTypes::SecurityForServiceTicket *securityForServiceTicket

= securityTickets-securityForServiceTicket;

8 DataTypes::ClientForServiceTicket clientForServiceTicket;

9 clientForServiceTicket.userId = securityTickets->securityForClientTicket->userId;

10 clientForServiceTicket.timeToIncrease = boost::lexical_cast<std::string> (rand ());

11 DataTypes::SessionKey sessionKey = securityTickets->

securityForClientTicket->sessionKey;

12 clientForServiceTicket.encrypt (SessionKey);

13 DataTypes::TicketsForService ticketsForService (securityForServiceTicket,

clientForServiceTicket);

14 return ticketsForService.out ();

} **15**

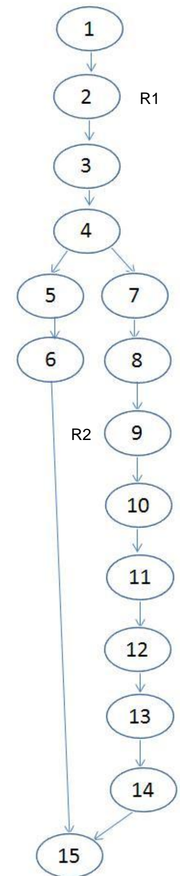

Figura 25 *Grafo del método GetTicketsForService*

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:15-15+2=2 por tanto el #regiones: 2 coincide con la complejidad ciclomática: 2.

Paso2.2:1+1=2 por tanto el #regiones: 2 coincide con la complejidad ciclomática: 2.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4, 5, 6,15.

Camino2:1,2,3,4,7,8,9,10,11,12,13,14,15.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los camino independientes	Caso de prueba	Resultado esperado
Camino1	securityTickets no se encuentra el servicio registrado en el servidor de Seguridad.	El método retorna un string vacío.

Camino2	securityTickets se encuentra registrado en el servidor de Seguridad.	El método retorna en un string el Tickets.

Tabla 27 *Representación de los casos de prueba del método GetTicketsForService*

4.1.2.3 Método VerifyServiceForClientTicket

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

bool VerifyServiceForClientTicket::exec(DataTypes::ServiceId serviceId, std::string serviceForClientTicket) const

{

1 showMessage->exec (“Verificando tickets emitido por el servicio", boost::lexical_cast<std::string> (serviceId), __FILE__, __LINE__);

2 DataTypes::RegisterClient registerClient;

3 try

{

4 registerClient.in (serviceForClientTicket);

}

5 catch (...)

{

6 Utils::ShowMessage *showMessage = Utils::ShowMessage::getInstance ();

showMessage->exec (“Error verificando ticket emitido por el servicio",

"Formato de serialización dañado.", __FILE__, __LINE__

7 return false;

}

8 if (! registerClient.attr1)

{

9 showMessage->exec (“Error verificando ticket emitido por el servicio",

"Ticket no enviado.” __FILE__, __LINE__);

10 delete registerClient.attr2;

11 return false;

}

12 DataTypes::ServiceForClientTicket *serviceForClientTicket = registerClient.attr2;

13 DataTypes::SecurityTicketsMap &securityTicketsMap = *modelManager

->getSecurityTicketsMap ();

14 DataTypes::SecurityTickets *securityTickets

= securityTicketsMap [serviceId];

15 DataTypes::SessionKey sessionKey = securityTickets

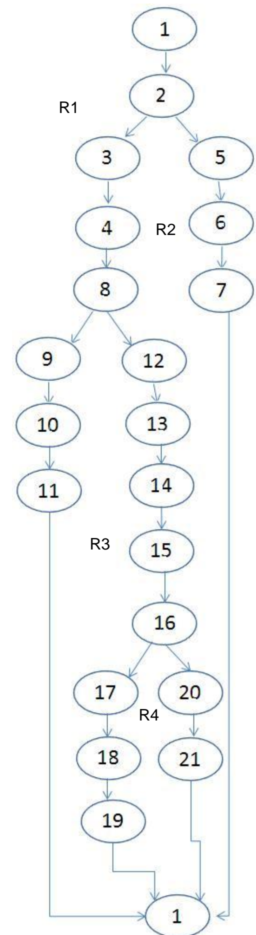

Figura 26 *Grafo del método VerifyServiceForClientTicket*

->securityForClientTicket->sessionKey;
16 if (! serviceForClientTicket->decrypt (sessionKey))
{
17 showMessage->exec (“Error desencriptando tickets emitido por el servicio", "Clave de sesión incorrecta.” __FILE__, __LINE__);

18 delete registerClient.attr2;
19 return false;
}
20 delete registerClient.attr2;
21 return true;
}

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:23-21+2=4 por tanto el #regiones: 4 coincide con la complejidad ciclomática: 4.

Paso2.3:3+1=4 por tanto el #regiones: 4 coincide con la complejidad ciclomática: 4.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 5, 6, 7,22.

Camino2:1, 2, 3, 4, 8, 9, 10, 11,22.

Camino3:1,2,3,4,5,8,12,13,14,15,16,17,18,19,22.

Camino4:1,2,3,4,8,12,13,14,15,16,20,21,22.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	registerClient es un formato de serialización dañado.	El método retorna falso.
Camino2	registerClient es un formato de serialización en buen estado y registerClient es un tickets no emitido por el servicio.	El método retorna falso.
Camino3	registerClient es un formato de serialización en buen estado, registerClient es un tickets emitido por el servicio y sessionKey es una clave de sesión incorrecta	El método retorna falso.
Camino4	registerClient es un formato de serialización en buen	El método retorna

	estado, registerClient es un tickets emitido por el servicio y sessionKey es una llave de sesión correcta	verdadero.

Tabla 28 *Representación de los casos de prueba del método VerifyServiceForClientTicket*

4.1.3 Casos de pruebas del Servidor

4.1.3.1 Método Server Authentication

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

bool ServerAuthentication::exec(Utils::ModuleType moduleType, std::string sharePath, std::string &sessionKey) const

{

1 Utils::ModuleName moduleName = Utils::getModuleName (moduleType);

1 std::string privateKeyPath = sharePath + "security.d/X509/" + moduleName + "Key.pem";

1 std::string certPath = sharePath + "security.d/X509/" + moduleName + "Cert.pem";

1 std::string caCertPath = sharePath + "security.d/X509/" + "caCert.pem";

1 std::string privateKey = SSLmm::loadPrivateKey (privateKeyPath);

2 if (privateKey.empty ())

{

3 Security::Utils::ShowMessage::getInstance()->exec("Error

autenticando servidores", "Cargando llave privada", __FILE__, __LINE__);

4 return false;

}

5 std::string ownCert = SSLmm::loadCertificate (certPath);

5 if (ownCert.empty ())

{

6 Security::Utils::ShowMessage::getInstance()->exec("Error

autenticando servidores", “Cargando certificado digital", __FILE__, __LINE__);

7 return false;

}

8 std::string caCert = SSLmm::loadCertificate (caCertPath);

9 if (caCert.empty ())

{

10 Security::Utils::ShowMessage::getInstance()->exec("Error

autenticando servidores", "Cargando certificado digital de la entidad

certificadora", __FILE__, __LINE__);

11 return false;

}

12 bool verify = SSLmm::verifyCertvsCA (ownCert, caCert);

13 if (! verify)

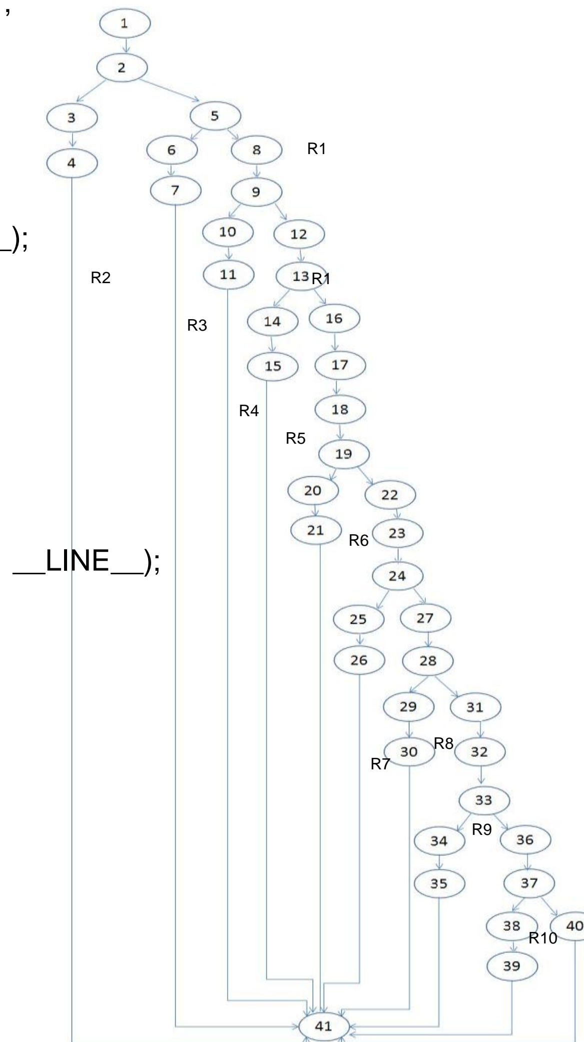

Figura 27 *Grafo del método ServerAuthentication*

{
14 Security::Utils::ShowMessage::getInstance()->exec("Error autenticando servidores" "Verificando certificado digital del servidor", __FILE__, __LINE__);
15 return false;
}
16 std::string ownText = DataTypes::getSessionKey ();
17 std::string ownSign;
18 bool sign = SSLmm::sign (ownText, privateKey, ownCert, ownSign);
19 if (! sign)
{
20 Security::Utils::ShowMessage::getInstance()->exec("Error autenticando servidores", "Firmando texto", __FILE__, __LINE__);
21 return false;
}
22 std::string otherCert;
22 std::string otherSign;
23 bool authenticated = iComm->serverAuthentication (ownSign, ownCert, otherSign, otherCert, sessionKey);
24 if (! authenticated)
{
25 Security::Utils::ShowMessage::getInstance()->exec("Error autenticando servidores", "Información de autenticación no enviada desde el " "servidor al que se desea conectar", __FILE__, __LINE__);
26 return false;
}
27 verify = SSLmm::verifyCertvsCA (otherCert, caCert);
28 if (! verify)
{
29 Security::Utils::ShowMessage::getInstance()->exec("Error autenticando servidores", "Verificando certificado digital del " "servidor al que se desea conectar", __FILE__, __LINE__);
30 return false;
}
31 std::string otherText;
32 verify = SSLmm::verify (otherSign, otherCert, otherText);
33 if (! verify || otherText! = ownText)
{
34 Security::Utils::ShowMessage::getInstance()->exec("Error autenticando servidores", "Verificando firma digital del " "servidor al que se desea conectar", __FILE__, __LINE__);
35 return false;
}
36 verify = SSLmm::decrypt (sessionKey, privateKey, ownCert, sessionKey);
37 if (! verify)

{
38 Security::Utils::ShowMessage::getInstance()->exec("Error autenticando servidores", "Desencriptando llave de sesión enviada por " "el servidor al que se desea conectar", __FILE__ ,__LINE__);
39 return false;
}
40 return true;
} **41**

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:49-41+2=10 por tanto el #regiones: 10 coincide con la complejidad ciclomática: 10.

Paso2.2:9+1=10 por tanto el #regiones: 10 coincide con la complejidad ciclomática: 10.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4,41.

Camino2:1, 2, 3, 4, 5, 6, 7,41.

Camino3:1, 2, 5, 8, 9, 10, 11,41.

Camino4:1, 2, 5, 8, 9, 12, 13, 14, 15,41.

Camino5:1,2,5,8,9,13,16,17,18,19,20,21,41.

Camino6: 1,2,5,8,9,12,13,16,17,18,19,22,23,24,25,26,41.

Camino7: 1,2,5,8,9,12,13,16,17,18,19,22,23,24,27,28,29,30,41.

Camino8: 1,2,5,8,9,12,13,16,17,18,19,22,23,24,27,28,31,32,33,34,35,41.

Camino9: 1,2,5,8,9,12,13,16,17,18,19,22,23,24,27,28,31,32,33,36,37,38,39,41.

Camino10: 1,2,5,8,9,12,13,16,17,18,19,22,23,24,27,28,31,32,33,36,37,,40,41.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico

Número de los caminos independientes	Caso de prueba	Resultado esperado
Camino1	Al cargar privateKey es vacía.	El método retorna falso.
Camino2	Al cargar privateKey es válida, al cargar el certificado digital ownCert es vacío.	El método retorna falso.
Camino3	Al cargar privateKey y el certificado digital ownCert son válidos, pero al cargar el certificado digital caCert de la entidad certificadora es vacío.	El método retorna falso.

Camino4	Al cargar privateKey, el certificado digital ownCert, el certificado digital caCert de la entidad certificadora son válidos, al verificar el certificado digital del servidor verify es falso.	El método retorna falso.
Camino5	Al cargar privateKey, el certificado digital ownCert, el certificado digital caCert son válidos, al verificar el certificado digital del servidor verify es verdadero, pero a la hora de firmar el texto sign es falso.	El método retorna falso.
Camino6	Al cargar privateKey, el certificado digital ownCert, el certificado digital caCert son válidos, al verificar el certificado digital del servidor verify es verdadero, a la hora de firmar el texto sign es verdadero, pero a la de Authenticated es falso.	El método retorna falso.
Camino7	Al cargar privateKey, el certificado digital ownCert, el certificado digital caCert son válidos, al verificar el certificado digital del servidor verify es verdadero, a la hora de firmar el texto sign es verdadero, Authenticated es verdadero, pero verify toma valor falso.	El método retorna falso.
Camino8	Al cargar privateKey, el certificado digital ownCert, el certificado digital caCert son válidos, al verificar el certificado digital del servidor verify es verdadero, a la hora de firmar el texto sign es verdadero, Authenticated es verdadero, verify toma valor verdadero, pero verify toma valor falso y otherText y ownText son iguales.	El método retorna falso.
Camino9	Al cargar privateKey, el certificado digital ownCert, el certificado digital caCert son válidos, al verificar el certificado digital del servidor verify es verdadero, a la hora de firmar el texto sign es verdadero, Authenticated es verdadero, verify toma valor verdadero, verify es verdadero o otherText y ownText son diferentes, pero verify es falso a la hora de desencriptar la llave se sesión.	El método retorna falso.

Camino10	Al cargar privateKey, el certificado digital ownCert, el certificado digital caCert son válidos, al verificar el certificado digital del servidor verify es verdadero, a la hora de firmar el texto sign es verdadero, Authenticated es verdadero, verify toma valor verdadero, verify es verdadero o otherText y ownText son diferentes, verify es verdadero a la hora de desencriptar la llave se sesión.	El método retorna verdadero.

Tabla 29 *Representación de los casos de prueba del método ServerAuthentication*

4.1.3.2 Método AddService

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

void AddService::exec(int serviceId) const

{

1 showMessage->exec (“Agregando servicio",

2 boost::lexical_cast<std::string> (serviceId), __FILE__, __LINE__);

3 DataTypes::SessionKey sessionKey = modelManager->getSessionKey ();

4 iComm->addService (serviceId, sessionKey);

}

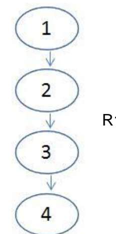

Figura 28 *Grafo del método AddService*

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:2-3+2=1 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1.

Paso2.2:0+1=11 por tanto el #regiones: 1 coincide con la complejidad ciclomática: 1.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2,3.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico

Número de los camino independientes	Caso de prueba	Resultado esperado
Camino1	serviceId es un número entero.	Se agrega el servicio.

Tabla 30 *Representación de los casos de prueba del método ServerAuthentication*

El método AddService llama internamente al método AddService de la clase iComm, implica que hay que hacerle la técnica.

Método addService de la clase iComm:

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

void IComm::addService(DataTypes::ServiceId serviceId, DataTypes::SessionKey sessionKey)

{

1 DataTypes::AddService addService (serviceId, sessionKey);

2 std::string data = addService.out ();

3 std::string result;

4 if (! sendSecurityData (1, data, result))

5 return;

6 if (result == "1")

{

7 showMessage->exec ("Servicio registrado correctamente en el servidor de Seguridad.");

8 return;

}

9 showMessage->exec ("Error registrando servicio en el servidor de Seguridad", "Formato de serialización dañado.", __FILE__, __LINE__);

} **10**

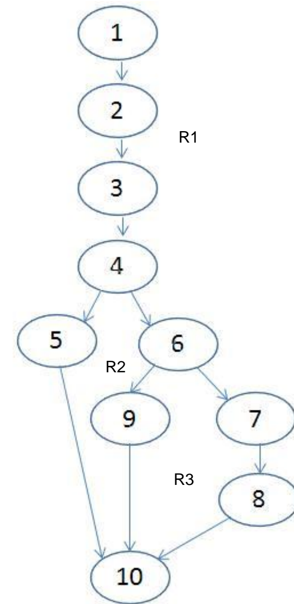

Figura 29 *Grafo del método addService de iComm*

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1: 11-10+2=3 por tanto el #regiones: 3 coincide con la complejidad ciclomática: 3.

Paso2.2: 2+1=3 por tanto el #regiones: 3 coincide con la complejidad ciclomática: 3.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 3, 4, 5,10.

Camino2:1, 2, 3, 4, 6, 7, 8,10.

Camino3: 1, 2, 3, 4, 6, 9,10.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico

Número de los caminos independientes	Caso de prueba	Resultado esperado

Camino1	la llamada al método es falso.	El método no hace nada.
Camino2	result es igual a 1	El servicio se registra correctamente en el servidor.
Camino3	result es igual a 0.	Error registrando servicio en el servidor.

***Tabla 31** Representación de los casos de prueba del método addService de iComm*

El método AddService de la clase iComm llama internamente al método sendSecurityData, implica que también hay que aplicarle la técnica. Ir a sendSecurityData.

4.1.3.3 Método RegisterClient

Paso 1: A partir del diseño o el código fuente, se dibuja el grafo de flujo asociado.

```
std::string RegisterClient::exec (std::string ticketsForService) const
{
1 showMessage->exec ( "Solicitando acceso", "", __FILE__, __LINE__);
2 DataTypes::TicketsForService ticketsForServiceStruct;
2 DataTypes::ServiceForClientTicket serviceForClientTicket;
2 DataTypes::RegisterClient registerClient (false, &serviceForClientTicket);
3 try
{
4 ticketsForServiceStruct.in (ticketsForService);
}
5 catch (...)
{
5    showMessage->exec (“Error solicitando acceso", "Formato de serialización
     dañado.", __FILE__, __LINE__);
7 return registerClient.out ();
}
8 DataTypes::SecurityForServiceTicket *securityForServiceTicket
= ticketsForServiceStruct.attr1;
9 DataTypes::SessionKey servicePassword = modelManager->getSessionKey ();
10 if (! securityForServiceTicket->decrypt (servicePassword))
{
```

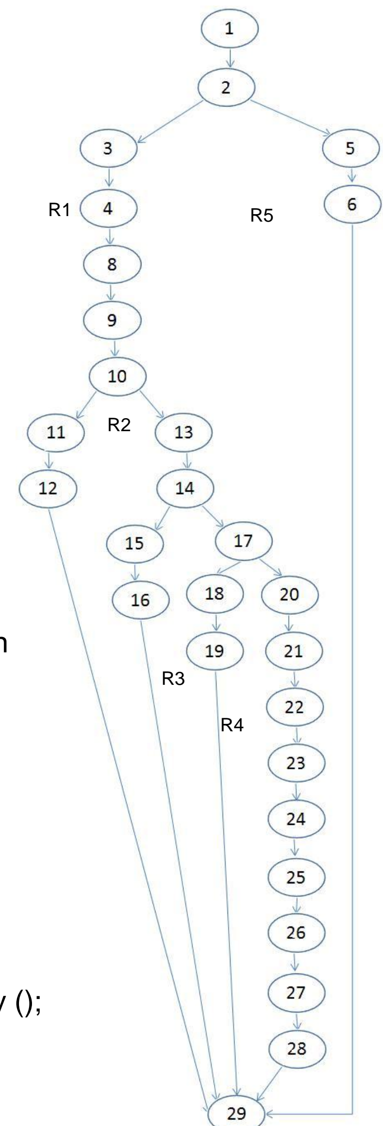

***Figura 30** Grafo del método RegisterClient*

11 showMessage->exec (“Error desencriptando tickets emitido por Seguridad", "Clave de sesión incorrecta.", __FILE__, __LINE__);

12 return registerClient.out ();

}

13 DataTypes::ClientForServiceTicket *clientForServiceTicket = ticketsForServiceStruct.attr2;

13 DataTypes::SessionKey sessionKey = securityForServiceTicket->sessionKey;

14 if (! clientForServiceTicket->decrypt (sessionKey))

{

15 showMessage->exec (“Error desencriptando tickets emitido por el cliente", “Clave de sesión incorrecta.", __FILE__, __LINE__);

16 return registerClient.out ();

}

17 if (clientForServiceTicket->userId! = securityForServiceTicket->userId)

{

18 showMessage->exec ("El usuario enviado por el cliente no coincide con el usuario enviado por el servidor.", "", __FILE__, __LINE__);

19 return registerClient.out ();

}

20 boost::posix_time::ptime time = boost::posix_time::from_iso_string(securityForServiceTicket->time);

20 std::string timeToIncrease = clientForServiceTicket->timeToIncrease;

21 boost::posix_time::time_duration microsecIncreased=boost::poxix_time::microsec(boost::lexical_cast

< unsigned int> (timeToIncrease));

21 time += microsecIncreased;

23 serviceForClientTicket.increasedTime = boost::posix_time::to_iso_string (time);

23 serviceForClientTicket.encrypt (sessionKey);

24 DataTypes::SessionId sessionId;

24 sessionId = boost::lexical_cast<DataTypes::SessionId> (securityForServiceTicket->sessionId);

25 DataTypes::EntityId userId = boost::lexical_cast<DataTypes::EntityId> (securityForServiceTicket->userId);

25 DataTypes::Session *session = new DataTypes::Session (sessionId, sessionKey, userId);

26 modelManager->getSessionManager () ->add (session);

27 registerClient.attr1 = true;

28 return registerClient.out ();

} **29**

Paso 2: Se calcula la complejidad ciclomática.

Paso2.1:31-29+2=5 por tanto el #regiones: 5 coincide con la complejidad ciclomática: 5.

Paso2.2:4+1=5 por tanto el #regiones: 5 coincide con la complejidad ciclomática: 5.

Paso 3: Se determina un conjunto básico de caminos independientes.

Camino1:1, 2, 5, 6, 7,29.

Camino2:1, 2, 3, 4, 8, 9, 10, 11, 12,29.

Camino3:1,2,3,4,8,9,10,11,13,14,15,16,29.

Camino4:1,2,3,4,8,9,10,13,14,17,18,19,29.

Camino5:1,2,3,4,8,9,10,13,14,17,20,21,22,23,24,25,26,27,28,29.

Paso 4: Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Número de los camino independientes	Caso de prueba	Resultado esperado
Camino1	ticketsForService tiene el formato de serialización dañado.	El método devuelve la salida del cliente registrado.
Camino2	ticketsForService es un formato de serialización válido y servicePassword es incorrecto.	El método devuelve la salida del cliente registrado.
Camino3	ticketsForService es un formato de serialización válido, servicePassword es correcta y sessionKey es una llave incorrecta.	El método devuelve la salida del cliente registrado.
Camino4	ticketsForService es un formato de serialización válido, servicePassword es correcta, sessionKey es una llave correcta y el Id de clientForServiceTicket no coincide con el Id de securityForServiceTicket	El método devuelve la salida del cliente registrado.

Camino5	ticketsForService es un formato de serialización válido, servicePassword es correcta, sessionKey es una llave correcta y el Id de clientForServiceTicket coincide con el Id de securityForServiceTicket.	Se pone el att1 del objeto del registro del cliente en true y se retorna la salida del registro del cliente.

Tabla 32 *Representación de los casos de prueba del método*

5.1 Generador de datos

Con el objetivo de hacerle las pruebas automatizadas a la biblioteca de encriptación SSLmm lo más rigurosas posible se desarrolló un generador de datos aleatorios, gracias a este se pueden incrementar la cantidad de datos a procesar para obtener un resultado más fiable en el desarrollo de las mismas. El lenguaje Python entre sus bibliotecas estándar contiene una para la encriptación de ficheros y cadenas de caracteres (Crypto.Cipher), la cual contiene los mismos métodos de cifrado desarrollados en la SSLmm, pero a diferencia de la última esta ya está probada y certificada por lo tanto se puede asumir que los resultados arrojados son correctos.

Para la implementación del generador se desarrolló una clase la cual contiene un conjunto de métodos los cuales generan, codifican y salvan en un fichero las cadenas aleatorias. Luego, estas serán comparadas con el resultado de codificar con la SSLmm arrojando posibles errores a la hora de codificar.

A continuación se mencionan los métodos contenidos en el generador de datos:

GenString: recibe como parámetro una longitud n y genera una cadena aleatoria de caracteres y números.

GenCadenasEntrada: se encarga de generar un número aleatorio que representa la cantidad de palabras que van a ser generadas y codificadas, son creados dos ficheros el primero para guardar las cadenas a encriptar (Cadenas_Entrada.txt) y el segundo para las contraseñas necesarias para codificar con AES (Pass_Entrada.txt). Luego, se generan utilizando el método GenString cada una de las cadenas aleatorias con longitudes de 48 para la cadena de entrada y 24 para las contraseñas.

B64: con método se carga el fichero que contiene las cadenas aleatorias y una a una se codifican usando el método de la Crypto.Cipher base64.encodestring y se guarda el resultado en un fichero (outB64.txt).

AesEncrypt: con este método se carga el fichero que contiene las cadenas aleatorias y otro con las contraseñas, se carga el modo (BCB) en que se usará el AES. Después de eso se encriptan las palabras crean un objeto de tipo AES con las llaves y el modo establecido y para que estas puedan ser procesadas por el CxxTest se les aplica un cambio de base con base64EncodeString y se guarda el resultado en un fichero (outAES.txt).

HashCode: carga el fichero con las cadenas de entrada y crea 4 ficheros para guardar la salida en los distintos modos (md5, sha1, sha256, sha512).

5.2 Conclusiones parciales

- Usando como base los diseños de casos de prueba realizados en este capítulo se llevará a cabo la implementación de las pruebas automatizadas en el próximo capítulo.
- El generador de datos implementado servirá como una vía de incrementar la cantidad de datos a procesar y es usado en el próximo capítulo para la ejecución de las pruebas para así poder obtener resultados más fiables.

CAPÍTULO 3: IMPLEMENTACIÓN Y RESULTADOS OBTENIDOS

Introducción

En este capítulo se lleva a cabo la implementación y la ejecución de las pruebas al módulo Seguridad a partir del plan de prueba y los diseños de casos de prueba definidos en el capítulo anterior. Se realiza además un análisis de los resultados obtenidos.

3.1 Diagrama de clases del diseño

Un diagrama de clases describe la estructura de un sistema mostrando sus clases, atributos y relaciones entre ellos. (Ver Figura 31).

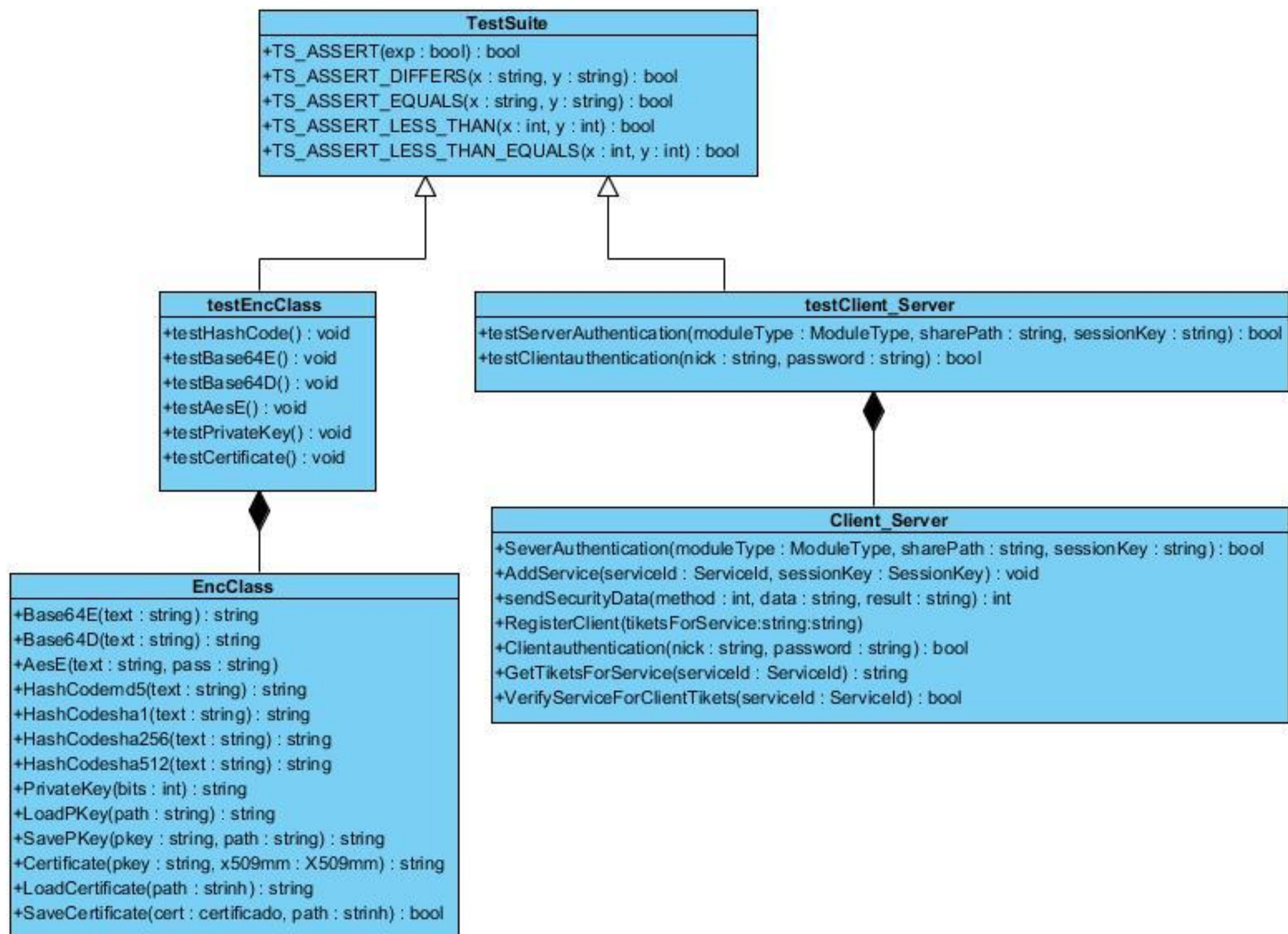

Figura 31 *Diagrama de clases*

TestSuite: clase abstracta de prueba, de la cual deben heredar todas las clases pruebas construidas en el sistema.

testEncClass: es la clase donde se lleva a cabo la ejecución de las pruebas de la biblioteca SSLmm.

Encclass: es la clase que codifica utilizando la librería de seguridad.

testClient_Server: es la clase encargada de la ejecución de las pruebas de la biblioteca ISecurity y el servidor de seguridad.

Client_Server: clase encargada de manejar los eventos del cliente entre estos se encuentran autenticarse en el servidor y pedirle al mismos los tickets para acceder los distintos servicios disponibles. Además de manejar los eventos del servidor de seguridad como registrar nuevos usuarios, la autenticación de los mismos en este, así como adicionar los servicios y controlar el acceso de los usuarios mediante los permisos.

3.2 Diagrama de componentes.

Un componente representa una parte física del sistema, el mismo contiene clases y puede ser implementado por uno o más artefactos (ficheros ejecutables, binarios). Son las piezas reutilizables de alto nivel a partir de las cuales se pueden construir sistemas. Cada componente define una interfaz que describe su funcionalidad y forma de empleo. El diagrama de componente, permite conocer a los desarrolladores y clientes la estructura física que tiene el sistema y como se relacionan sus partes. A continuación se presenta el diagrama de componentes de la aplicación (Ver Figura 32).

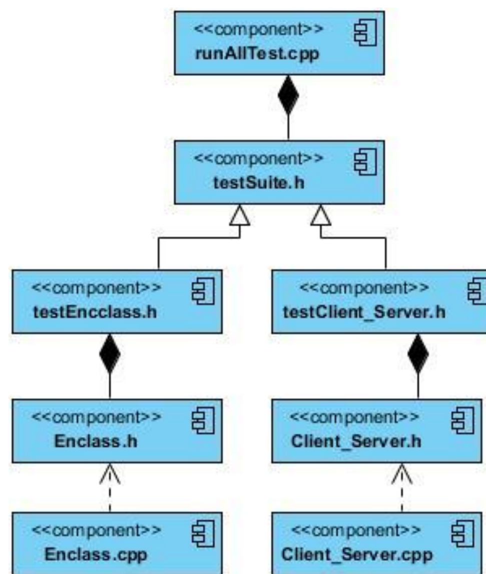

Figura 32 *Diagrama de componente*

3.3 Implementación

Para un mejor entendimiento de la implementación de la aplicación se realizará la explicación por pasos hasta llegar a los resultados obtenidos en este trabajo.

Paso 1: Descripción de los algoritmos desarrollados para probar la biblioteca SSLmm.

El método testHashCode es el encargado de probar el cálculo de Hash Code de una cadena. El mismo crea una clase y carga 2 ficheros, el primero contiene las cadenas a encriptar o de entradas y el segundo las cadenas encriptadas previamente usando la biblioteca Crypto.Cipher de Python en los 4 modos (md5, sha1, sha512 y sha256). Luego, es invocado el método HashCode de las SSLmm y se compara el resultado con el previamente hecho mediante la sentencia TS_ASSERT_EQUALS(x, y). En este método cada cadena es encriptadas usando cada uno de los modos de calcular Hash con la SSLmm y comparado con las previamente encriptadas usando la misma forma. (Ver anexo 1).

El método testBase64E prueba el encriptado del cambio de base de una cadena. El mismo crea una clase y carga 2 ficheros, el primero contiene las cadenas a encriptar o de entradas y el segundo las cadenas encriptadas previamente usando la biblioteca Crypto.Cipher de Python. Luego, es invocado el método base64EncodeString de la SSLmm y se compara el resultado con el previamente hecho mediante la sentencia TS_ASSERT_EQUALS(x, y) (Ver anexo 2).

El método testAesE se encarga de probar el algoritmo AES encriptando. El mismo crea una clase y 3 ficheros, el primero contiene las cadenas a encriptar o de entradas, el segundo las cadenas encriptadas previamente con la biblioteca Crypto.Cipher de Python y el último contiene las claves con las que se va a encriptar. Después se invoca el método aesEncrypt de la SSLmm y se compara el resultado con el previamente hecho mediante la sentencia TS_ASSERT_EQUALS(x, y) para comprobar que el resultado es el mismo (Ver anexo 3).

El método testAesD es el encargado de probar el algoritmo AES desencriptando. El mismo crea una clase y carga 3 ficheros, el primero contiene las cadenas a encriptar o de entradas, el segundo las cadenas encriptadas previamente usando la Crypto.Cipher de Python y el último contiene las claves con las que se va a encriptar. Luego, se invoca el método aesDecrypt de la SSLmm y se compara el resultado con el previamente hecho mediante la sentencia TS_ASSERT_EQUALS(x, y)(Ver anexo 4).

El método TestPrivateKey es el encargado de probar las llaves de las siguientes maneras (Ver anexo 5):

- Se crea una llave privada con una longitud determinada.
- Se prueba si la llave se salva en un fichero .pem de manera exitosa.
- Se carga en otra variable el mismo fichero donde fue guardada la llave y se comprueba que la llave obtenida sea la misma.

El método TestCertificate se encarga de probar los certificados digitales de la siguiente manera (Ver anexo 6):

- Se crea una llave privada que es necesaria en la creación de un certificado.
- Se crea un objeto X509Name y un X509mm para la creación del certificado.
- El certificado se crea pasándole la llave y el X509mm como parámetros.
- Se salva el certificado en un fichero .pem y luego se carga en otra variable y se comparan para determinar si son iguales.

Paso 2: Descripción de los algoritmos desarrollados para probar la biblioteca ISecurityClient y el servidor seguridad.

El método TestServerAuthentication es el encargado de crear un servidor de seguridad y adicionar un servicio, luego autentica un servidor de Base de Datos Histórico usando un certificado digital. Por último comprueba que el proceso se realizó con éxito (Ver anexo 11).

El método TestClientAuthentication es el encargado de crear un servidor de seguridad, un cliente y autenticar el cliente con su usuario y contraseña en el mismo. Luego, comprueba que se autentico con éxito,

solicita un tickets para un servicio y comprueba que el tickets solicitado sea el indicado para el servicio (Ver anexo 12).

Paso 4: Después de visto la descripción de cada uno de los algoritmos desarrollados para probar la biblioteca SSLmm, ISecurity y el servidor, se muestra a continuación el reporte de las pruebas.

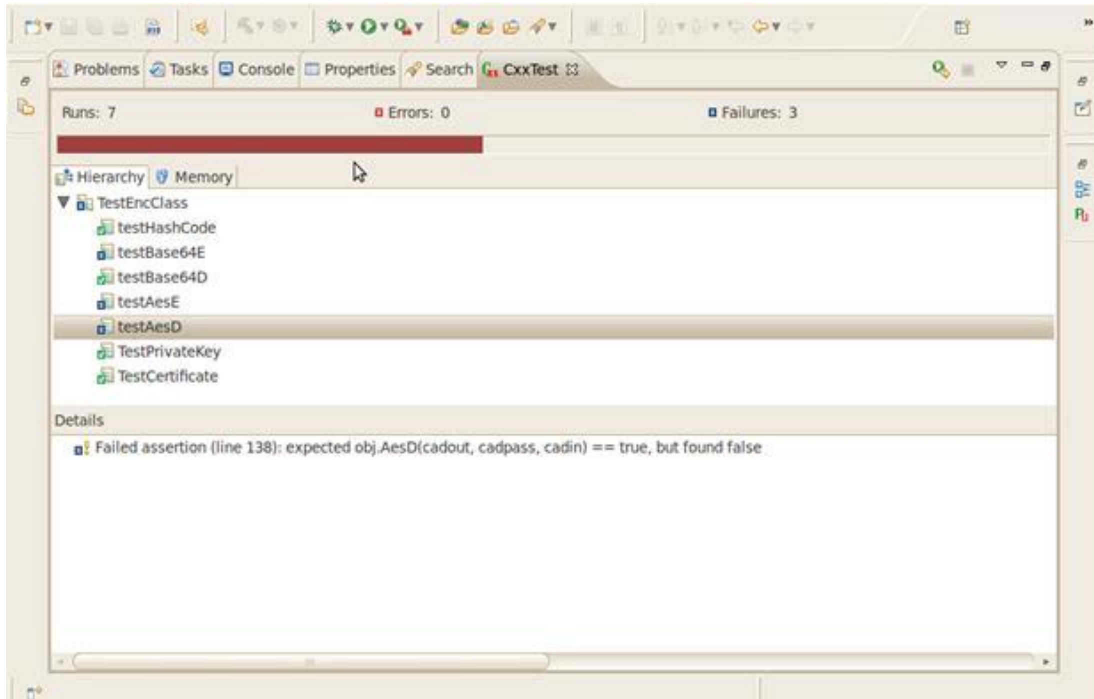

Figura 33 *Reporte de pruebas*

3.4 Análisis de los resultados

Para la Biblioteca SSLmm se identificaron 22 casos de pruebas distribuidos como se muestra en la tabla.

algoritmos	Cantidad de casos de prueba
testHashCode	4
testBase64E	1
testBase64D	1
testAesE	1
testAesD	1
TestPrivateKey	7
TestCertificate	7

Tabla 33 *Casos de prueba de la biblioteca SSLmm*

Los resultados de las pruebas ejecutadas son los siguientes:

testHashCode

Ejecutados, satisfactoriamente- 0 casos de pruebas.

Ejecutados, no satisfactoriamente- 4 casos de prueba.

No ejecutado, no satisfactoriamente – 0 casos de prueba.

testBase64E

Ejecutados, satisfactoriamente- 1 casos de pruebas.

Ejecutados, no satisfactoriamente- 0 casos de prueba.

No ejecutado, no satisfactoriamente – 0 casos de prueba.

testBase64D

Ejecutados, satisfactoriamente- 0 casos de pruebas.

Ejecutados, no satisfactoriamente- 1 casos de prueba.

No ejecutado, no satisfactoriamente – 0 casos de prueba.

testAesE

Ejecutados, satisfactoriamente- 1 casos de pruebas.

Ejecutados, no satisfactoriamente- 0 casos de prueba.

No ejecutado, no satisfactoriamente – 0 casos de prueba

testAesD

Ejecutados, satisfactoriamente- 1 casos de pruebas.

Ejecutados, no satisfactoriamente- 0 casos de prueba.

No ejecutado, no satisfactoriamente – 0 casos de prueba

TestPrivateKey

Ejecutados, satisfactoriamente- 0 casos de pruebas.

Ejecutados, no satisfactoriamente- 7 casos de prueba.

No ejecutado, no satisfactoriamente – 0 casos de prueba

TestCertificate

Ejecutados, satisfactoriamente- 0 casos de pruebas.

Ejecutados, no satisfactoriamente- 7 casos de prueba.

No ejecutado, no satisfactoriamente – 0 casos de prueba

Para la biblioteca ISecutityClient y el servidor de seguridad se identificaron 35 casos de prueba, distribuidos como se muestra en la tabla.

algoritmos	Cantidad de casos de prueba
TestServerAuthentication	21
TestClientAuthentication	14

Tabla 34 *Casos de prueba del cliente y servidor de seguridad*

Los resultados de las pruebas ejecutadas son los siguientes:

TestServerAuthentication

Ejecutados, satisfactoriamente- 0 casos de pruebas.

Ejecutados, no satisfactoriamente- 21 casos de prueba.

No ejecutado, no satisfactoriamente – 0 casos de prueba

TestClientAuthentication

Ejecutados, satisfactoriamente- 0 casos de pruebas.

Ejecutados, no satisfactoriamente- 14 casos de prueba.

No ejecutado, no satisfactoriamente – 0 casos de prueba

A continuación se muestra las no conformidades de las pruebas realizadas al módulo Seguridad.

no	No conformidad	Aspecto correspondiente	Significativa	Estado NC
1	El algoritmo de Base64EncodeString de la biblioteca SSLmm presenta errores en la codificación.	El resultado del algoritmo Base64EncodeString cuando es comparado con el resultado del algoritmo Base64EncodeString de la Crypto.Cipher, el resultado no es el mismo, el Base64EncodeString genera caracteres extraños demás.	X	Pendiente
2	El algoritmo aesEncrypt de la biblioteca SSLmm presenta errores de codificación.	El resultado del algoritmo aesEncrypt cuando es comparado con el resultado del algoritmo aesEncrypt de la Crypto.Cipher, el resultado no es el mismo, porque el aesEncrypt llama internamente al Base64EncodeString.	X	Pendiente

3	El algoritmo aesDecrypt de la biblioteca SSLmm presenta errores de decodificación.	El resultado del algoritmo aesDecrypt cuando es comparado con el resultado del algoritmo aesDecrypt de la Crypto.Cipher, el resultado no es el mismo.	X	Pendiente

Tabla 35 *No conformidades de la biblioteca SSLmm*

3.5 Conclusiones parciales.

- En este capítulo se explicaron paso a paso los algoritmos implementados para probar cada uno de los componentes del módulo Seguridad dándole así cumplimiento al objetivo fundamental del trabajo.
- Se hace un análisis detallado de los resultados obtenidos y se realiza una tabla de las no conformidades para la corrección de los errores.

CONCLUSIONES

Con la realización del presente trabajo de diploma se obtienen las siguientes conclusiones:

- Para poder obtener una aplicación capaz de ejecutar pruebas de manera automatizada al módulo Seguridad se hizo necesario la elaboración de un plan de pruebas que se ajustara a las necesidades del mismo y los diseños de casos de prueba se realizaron usando una técnica que pudiera dar una medida de la complejidad del módulo.
- Como producto del desarrollo de esta investigación se obtuvo una aplicación capaz de detectar vulnerabilidades en el módulo Seguridad y hacerles un reporte de las mismas para ser corregidas. Además de que estas pruebas sirven de base para poder realizar otras más complejas.

RECOMENDACIONES

Después de aplicado el proceso de prueba al módulo Seguridad y comprobado los beneficios que trae consigo se recomienda:

- Cuando se le realice una interfaz al módulo de Seguridad hacerle pruebas de caja negra para obtener mejores resultados de funcionamiento.
- Utilizando como base las pruebas de unidad que fueron hechas en esta investigación realizarles al módulo pruebas de integración continua, ya que estas pruebas son más efectivas pues se realizan cada vez que el código sufre algún cambio.
- Utilizar como referencia para futuros trabajo de pruebas automatizadas el trabajo de diploma.

BIBLIOGRAFÍA

1. **Error de software** [En línea] [Citado el: diciembre 7, 2012.] http://riveraortizz.blogspot.com/p/mantenimientopreventivo-software_28.html.

2. **Tello, Eduardo A. Rodríguez.** *Importancia de las pruebas de software.* 2011.

3. **EVA**. [En línea][Citado el: 10 de Diciembre de 2012] http:// eva.uci.cu/ mod/ resource /view.php?id=4179&subdir=/Temas Generales

4. **Fontela, Pablo Carlos Suárez.** *Documentación y pruebas antes del paradigma de objeto.* 2003.

5. **EVA.** [En línea] [Citado el: 14 de Enero de 2013.] http://eva.uci.cu/mod/resource/view.php?id=8500&subdir=/Ediciones_del_Pressman/Pressman_6ta_edicion.

6. **López, Abel Quintana.** *Herramienta de Pruebas Automatizadas para el módulo HMI del editor Phoenix.* UCI: s.n., 2011.

7. Buenas Tareas. [En línea] [Citado el: 20 de Enero de 2013.] http://www.buenastareas.com/ensayos/Prueba-De-Integraci%C3%B3n-De-Software/900311.html.

8. Ecured. [En línea] [Citado el: 5 de Diciembre de 2012.] http://www.ecured.cu/index.php/Automatizaci%C3%B3n_de_pruebas.

9. **S. Pressman**, **R**. (2005). INGENIERÍA DEL SOFTWARE: UN ENFOQUE Práctico Quinta edición.

10. [En línea] [Citado el: 20 de Enero de 2013.] http://www.inei.gob.pe/biblioineipub/bancopub/Inf/Lib5103/Libro.pdf.

11. **OpenUP** [En línea] [Citado el: 3 de Febrero de 2013.] http://todotecnology.blogspot.com/2009/11/metodologia-open-up.html.

12. **Metodología OpenUP** [En línea] [Citado el: 3 de Febrero de 2013.] http://kasyles.blogspot.com/2008/09/openup-como-alternativa-metodolgica.html.

13. Gestión de proyectos. [En línea] [Citado el: 3 de Febrero de 2013.] http://openup.es/metodologia-beneficios-de-hacerlo-con-open-up/.

14. **Sánchez, María A. Mendoza.** *Metodologías De Desarrollo De Software.* 2004.

15. **Molpeceres, Alberto.** *Proceso de desarrollo:RUP, XP, FDD.* 2002.

16. **Metodología XP** [En línea] [Citado el: 8 de Febrero de 2013] http://es. scribd.com/doc/72420606/METODOLOGIA-XP.

17. **Díaz, Toni de la Fuente.** *Usando OpenSSL en el mundo real. Quítate el miedo a usar OpenSSL.* 2006.

18. **IEEE.** Computer Dictionary. [Trad]. Inglés. 1990. Standard 610.

19. *CxxTest User Guide.* 2012.

20. **Estrada, Marlin Mariana Briones.** *Sistema de Información administrativo-contable para la planta Agro-Industrial “Casanova" en el cantón el Empalme.* 2011.

21. **Marzal, Andrés and García, Isabel.** *Introducción a la programación con Python.*

22. **autores, Conjunto de.** *Aprenda a Pensar Como un Programador con Python.* 2002.

23. **Carrillo Pérez, Isaías, Pérez González, Rodrigo and Rodríguez Martín, Aureliano David.** *METODOLOGÍA DE DESARROLLO DEL SOFTWARE.* 2008.

24. *Capítulo 3 PROCESO DE DESARROLLO OPEN UP/OAS Roles y artefactos del proceso.* 2013.

25. **Rodríguez Corbea, Maite and Ordóñez Pérez, Meylin.** *Metodología XP Aplicable al Desarrollo de Educativo en Cuba.* 2007.

26. **Delgado**, **R**. (2008). *0516_ROLES Y RESPONSABILIDADES.* Cuba.

27. **Letelier**, **P**. *Pruebas del Software.* Departamento Sistemas Informáticos y Computación.

28. **Aguilar**, **V**. **H**., & **González Jorrín**, **M**. (n.d.). *Proceso de pruebas de caja negra basado en la descripción de casos de usos.* Cuba.

29. **Estudio-frameworks** [En línea] [Citado el: 3 de mayo de 2013.] autor: Irina Elena Argota Vega http://www.ilustrados.com/tema/12008/Estudio-frameworks-para-desarrollo-pruebas-software.html

30. **EVA** [En línea] [Citado el: 25 de Diciembre de 2012] http://eva.uci.cu/mod/resource/view.php?id=9065/Materiales básicos/Documentación sobre pruebas.pdf

31. **Beyris Soulary, Liliam Celia y Reyes Bermúdez, Enrique.** *Informe de análisis y diseño para el módulo de Seguridad del Sistema de Supervisión y Control Guardián del ALBA.* 2013.

GLOSARIO

A

ASSERT es una función que comprueba si la expresión que le pasas como argumento es falsa y en tal caso, escribe un mensaje de error en el dispositivo de error estándar y llama a la señal *abort.* Mediante esta función se puede controlar que variables importantes con las que se trabaja se mantengan en un cierto rango de valores.

I

IEEE corresponde a las siglas en español Instituto de Ingenieros Eléctricos y Electrónicos: es una asociación técnico-profesional mundial dedicada a la estandarización. Es la mayor asociación internacional, sin ánimo de lucro, formada por profesionales de las nuevas tecnologías.

P

Pruebas automatizadas es un método de ejecutar una prueba sin intervención humana, que de lo contrario lo requeriría. En términos prácticos puede significar una prueba de interfaces gráficas de usuario (GUI para sus siglas en inglés) usando una herramienta de prueba comercial o una prueba escrita con un lenguaje de *scripting* interno de una aplicación.

ANEXOS

```
void testHashCode()
{
    char cadin[255];
    char cant[10];
    char cadout[255];
    int can;
    EncClass obj(true);
    ifstream fin("Cadenas_Entrada.txt");
    ifstream foutmd5("outHash_md5.txt");
    ifstream foutsha1("outHash_sha1.txt");
    ifstream foutsha256("outHash_sha256.txt");
    ifstream foutsha512("outHash_sha512.txt");
    fin.getline(cant, 10);
    can = atoi(cant);
    for (int i = 0; i < can; ++i)
    {
        fin.getline(cadin, 255);
        //md5
        foutmd5.getline(cadout, 255);
        TS_ASSERT_EQUALS(obj.HashCodemd5(cadin), cadout);
        //sha1
        foutsha1.getline(cadout, 255);
        TS_ASSERT_EQUALS(obj.HashCodesha1(cadin), cadout);
        //sha256
        foutsha256.getline(cadout, 255);
        TS_ASSERT_EQUALS(obj.HashCodesha256(cadin), cadout);
        //sha512
        foutsha512.getline(cadout, 255);
        TS_ASSERT_EQUALS(obj.HashCodesha512(cadin), cadout);
    }
```

Anexo 1 *Código del testHashCode*

```
void testBase64E()
{
    EncClass base(true);
    char cadin[255];
    char cadout[255];
    ifstream fin("inB64.txt");
    ifstream fout("outB64.txt");
    while(!fin.eof() && !fout.eof())
    {
        fin.getline(cadin, 255);
        fout.getline(cadout, 255);
        TS_ASSERT_EQUALS(base.Base64E(cadin), cadout);
    }
    fin.close();
    fout.close();

}
```

Anexo 2 *Código del testBase64E*

```
void testAesE()
{
    EncClass obj(true);
    int can;
    char cant[10];
    char cadin[255];
    char cadout[255];
    char cadpass[255];
    ifstream fin("Cadenas_Entrada.txt");
    ifstream fout("outAES.txt");
    ifstream fpass("Pass_Entrada.txt");
    fin.getline(cant, 10);
    can = atoi(cant);
    for (int i = 0; i < can; ++i)
    {
        fin.getline(cadin, 255);
        fout.getline(cadout, 255);
        fpass.getline(cadpass, 255);
        TS_ASSERT_EQUALS(obj.AesE(cadin, cadpass), cadout);
    }
    fin.close();
    fout.close();
    fpass.close();
}
```

Anexo 3 *Código del estAesE*

```
void testAesD()
    {
        EncClass obj(true);
        int can;
        char cant[10];
        char cadin[255];
        char cadout[255];
        char cadpass[255];
        ifstream fin("Cadenas_Entrada.txt");
        ifstream fout("outAES.txt");
        ifstream fpass("Pass_Entrada.txt");
        fin.getline(cant, 10);
        can = atoi(cant);
        for (int i = 0; i < can; ++i)
        {
            fin.getline(cadin, 255);
            fout.getline(cadout, 255);
            fpass.getline(cadpass, 255);
            TS_ASSERT(obj.AesD(cadout, cadpass, cadin));
        }
        fin.close();
        fout.close();
        fpass.close();
    }
```

Anexo 4 Código del testAesD

```
void TestPrivateKey()
{
    EncClass obj(true);
    string pkey, pkey2;
    //Crear Guardar y cargar una llave privada
    for (int i = 0; i < 100; ++i)
    {
        pkey=obj.PrivateKey(512);
        TS_ASSERT(obj.SavePKey(pkey, "/home/miguel/workspace/EncryptationTest/llave.pem"));
        pkey2 = obj.LoadPKey("/home/miguel/workspace/EncryptationTest/llave.pem");
        TS_ASSERT_EQUALS(pkey, pkey2);
    }

}
```

Anexo 5 *Código de TestPrivateKey*

```
void TestCertificate()
{
    //Crear Guardar y cargar un certificado
    EncClass obj(true);
    string cer1, cer2, pkey;
    pkey = obj.PrivateKey(512);
    X509Name* name = new X509Name("Miguel", "UCI", "Cuba", "Habana");
    X509mm* x509 = new X509mm(123456789, 9, name, 1234, 12345);
    cer1 = obj.Certificate(pkey, x509);
    TS_ASSERT(obj.SaveCertificate(cer1, "/home/miguel/workspace/EncryptationTest/certificado.pem"));
    cer2 = obj.LoadCertificate("/home/miguel/workspace/EncryptationTest/certificado.pem");
    TS_ASSERT_EQUALS(cer1, cer2);

}
```

Anexo 6 *Código de TestCertificate*

```
class Generator():
    def __init__(self):
        self.a=1

    "Generador aleatorio de una cadena de long n (Numeros y letras)"
    def GenString(self,n):
        return ''.join([choice(string.letters + string.digits) for i in range(n)])

    "Generador de cadenas de entrada y pass hasta 1000"
    def GenCandenasEntrada(self):
        cantpalabras = random.choice(range(100))
        fpass = open('Pass_Entrada.txt', 'w+')
        fout = open('Cadenas_Entrada.txt', 'w+')
        fout.write(str(cantpalabras))
        fout.write('\n')
        for i in range(cantpalabras):
            fout.write(self.GenString(48)+'\n')
            fpass.write(self.GenString(32)+'\n')
        fout.close()
        fpass.close()
```

Anexo 7 *Generador de cadena parte 1*

```
"Codificador de Base64"
def B64(self):
    fin = open('Cadenas_Entrada.txt', 'r')
    fout = open('outB64.txt', 'w+')
    palabra = fin.readline()
    palabra = fin.readline()
    palabra = palabra[: -1]
    while palabra != "":
        s = base64.encodestring(palabra)
        fout.write(s)
        palabra = fin.readline()
        palabra = palabra[: -1]
    fin.close()
    fout.close()
```

Anexo 8 *Generador de cadena parte 2*

```
"Codificoador de AES"
def AESEncryp(self):
    mode = AES.MODE_CBC
    fin = open('Cadenas_Entrada.txt', 'r')
    fpass = open('Pass_Entrada.txt', 'r')
    fout = open('outAES.txt', 'w+')
    palabra = fin.readline()
    palabra = fin.readline()
    palabra = palabra[: -1]
    llave = fpass.readline()
    llave = llave[: -1]
    while palabra != "":
        encryptor = AES.new(llave, mode)
        cambio = base64.encodestring(encryptor.encrypt(palabra))
        fout.write(cambio)
        palabra = fin.readline()
        palabra = palabra[: -1]
        llave = fpass.readline()
        llave = llave[: -1]
    fin.close()
    fpass.close()
    fout.close()
```

Anexo 9 *Generador de cadena parte 3*

```
"Hash Code"
def HashCode(self):
    fin = open('Cadenas_Entrada.txt', 'r')
    foutmd5 = open('outHash_md5.txt', 'w+')
    foutsha1 = open('outHash_sha1.txt', 'w+')
    foutsha256 = open('outHash_sha256.txt', 'w+')
    foutsha512 = open('outHash_sha512.txt', 'w+')
    palabra = fin.readline()
    palabra = fin.readline()
    palabra = palabra[: -1]
    while palabra != "":
        foutmd5.write(hashlib.md5(palabra).hexdigest()+'\n')
        foutsha1.write(hashlib.sha1(palabra).hexdigest()+'\n')
        foutsha256.write(hashlib.sha256(palabra).hexdigest()+'\n')
        foutsha512.write(hashlib.sha512(palabra).hexdigest()+'\n')
        palabra = fin.readline()
        palabra = palabra[: -1]
    fin.close()
    foutmd5.close()
    foutsha1.close()
    foutsha256.close()
    foutsha512.close()
```

***Anexo 10** Generador de cadena parte 4*

```
void TestServerAuthentication()
{

    ISecurityServer* server = new ISecurityServer(nameServer1IP, nameServer1Port, nameServer2IP, nameServe
                                                   localIP, nameServer, timeOut, debug);
    server->addService(0);
    string session;
    bool auth = server->authentication(HDB_MODULE, "/home/miguel/scadanac/share/security.d/X509", session)
    TS_ASSERT(auth);
}
```

***Anexo 11** Código de TestServerAuthentication*

```
void TestClientAuthentication()
{

        ISecurityServer* server = new ISecurityServer(nameServer1IP, nameServer1Port, nameServer2IP, nameS
                                                       localIP, nameServer, timeOut, debug);
        ISecurityClient* client = new ISecurityClient(nameServer1IP, nameServer1Port, nameServer2IP, nameS
                                                                  localIP, nameServer, timeOut, debug);
        bool auth = client->authentication("kike", "erb20044");
        TS_ASSERT(auth);
        ServiceId id = 0;
        string ticket = client->getTicketsForService(id);
        bool verif = client->verifyServiceForClientTicket(id, ticket);
        TS_ASSERT(verif);
}
```

***Anexo 12** Código de TestClientAuthentication*