UNIVERSIDAD DE LAS CIENCIAS INFORMÁTICAS
VICERRECTORÍA DE FORMACIÓN
DIRECCIÓN DE FORMACIÓN POSTGRADUADA

Patrones de diseño seguro para aplicaciones Web

Tesis presentada en opción al título de
Máster en Informática Aplicada

Autor: Joaquín Quintas Santiago
Tutor: Dr C Ángel Azoy Quintana

Ciudad de la Habana, febrero de 2010

DECLARACIÓN JURADA DE AUTORÍA Y AGRADECIMIENTOS

Yo Joaquín Quintas Santiago, con carné de identidad 459978, declaro que soy el autor principal del resultado que expongo en la presente memoria titulada: "Patrones de diseño seguro para aplicaciones Web", para optar por el título de Máster en Informática Aplicada.

Este trabajo fue desarrollado durante un año en colaboración con mi colega de equipo Mr. C Luis Ernesto Figueredo Santisteban, quien me reconocen la autoría principal del resultado expuesto en esta memoria.

A mi colega del equipo de trabajo le estoy muy agradecido. En especial deseo agradecer al Dr. C Ángel Azoy Quintana, quien fungió como tutor de mi formación como máster y contribuyó a mi crecimiento profesional y humano en general. A ellos, así como a otros colegas y amigos que no he mencionado por razones de espacio, les doy las más sinceras gracias.

Finalmente declaro que todo lo anteriormente expuesto se ajusta a la verdad, y asumo la responsabilidad moral y jurídica que se derive de este juramento profesional.

Y para que así conste, firmo la presente declaración jurada de autoría en Ciudad de la Habana a los 20 días del mes de febrero del año 2010.

____

Joaquín Quintas Santiago

Síntesis

Muchas de las aplicaciones que hoy en día se producen, contienen numerosos problemas de seguridad que podrían ser evitados si se incorporaran al ciclo de vida de desarrollo, un conjunto de actividades orientadas a evitar las vulnerabilidades que se introducen en cada una de las fases de dicho ciclo.

Emplear dichas actividades, implica un incremento considerable del esfuerzo que deben de realizar los equipos de desarrollo y requiere, además, que estos posean conocimientos y experiencia en el campo de la seguridad para que sean realmente efectivas.

Una alternativa, puede ser el empleo de los patrones de diseño seguro que, por sus características, son fáciles de utilizar y no requieren de tanta experiencia para su empleo. Para adecuarlos a las necesidades de las aplicaciones Web, se realizó un trabajo de selección, con el apoyo de un listado de las principales vulnerabilidades que afectan a las aplicaciones.

De la realización de este proceso, se obtuvieron ocho patrones de diseño seguro que, por sus características, pueden contribuir en la reducción de una parte importante de los problemas de seguridad que las aplicaciones Web deben enfrentar.

Índice

Introducción 1
Capítulo 1: Métodos de desarrollo de aplicaciones seguras 6
1.1 - Introducción 6
1.2 - Que es un software seguro 6
1.3 - Actividades que contribuyen a desarrollar un software seguro 7
1.3.1 - Requisitos de seguridad 8
1.3.2 - Casos de abuso 9
1.3.3 - Análisis de riesgo de arquitectura y diseño 11
1.3.4 - Revisión de código 12
1.3.5 - Pruebas de penetración 13
1.4 - Patrones de diseño seguro 15
1.4.1 - Repositorios de patrones 16
1.5 - Taxonomías de vulnerabilidades 17
1.5.1 - 2009 MITRE SANS Top 25 19
1.6 - Conclusiones 20
Capítulo 2: Patrones de diseño seguro 22
2.1 Introducción 22
2.2 Mecanismos para evitar o reducir las posibilidades de ataque de las principales vulnerabilidades 22
2.3 Patrones de diseño seguro que corresponden a los mecanismos de seguridad seleccionados 23
2.3.1 Patrón: Validador Interceptor 24
2.3.2 Patrón: Escape de las salidas 26
2.3.3 Patrón: Proxy confiable 28
2.3.4 Patrón: Canal seguro 30
2.3.5 Patrón: Identificador de autenticidad de transacciones 33
2.4 Otros patrones de diseño seguro 35
2.4.1 Patrón: Punto único de acceso 35
2.4.2 Patrón: Punto de chequeo 37
2.4.3 Patrón: Sesión dirigida 39
2.5 Efecto esperado del empleo de los patrones para la seguridad del sistema 41
2.6 Conclusiones 43
Capítulo 3: Beneficios del empleo de los patrones de diseño seguro 44
3.1 Introducción 44
3.2 Implicaciones de llevar a cabo un análisis de riesgo 44
3.3 Implicaciones de aplicar los patrones de diseño seguro 46
3.4 Ventajas del empleo de los patrones de diseño seguro propuestos 46
3.5 Validación experimental de los beneficios para la seguridad del empleo de los patrones 47
3.6 Conclusiones 48
Conclusiones 50
Recomendaciones 51
Referencias bibliográficas 52

Anexo 1: Solución a las principales vulnerabilidades propuestas en el listado "*MITRE CWE/SANS Top 25*" 56
Anexo 2: Estructuras estáticas y dinámicas de los patrones propuestos...............................64

Introducción

En el transcurso de la última década, las tecnologías de la informática y las comunicaciones han tenido un desarrollo vertiginoso. Cada vez es mayor el volumen de información que es procesado, almacenado o accedido de forma digital. Como consecuencia de este desarrollo, se ha incrementado nuestra dependencia de los sistemas informáticos que, paulatinamente, han comenzado a influir en numerosos aspectos de nuestras vidas.

En la misma medida en que hemos ido evolucionando hacia la era digital, es mayor la cantidad de recursos y procesos que se encuentran bajo el control de sistemas informáticos que, al ir ganando en importancia, han despertado el interés de toda una gama de delincuentes cibernéticos que se han aprovechado de numerosos problemas de seguridad, existentes en las aplicaciones, para llevar a cabo: desvíos de recursos, fraudes en los sistemas contables, robos en cuentas bancarias, entre otros muchos.

Estos problemas de seguridad, tienen su origen en debilidades en el diseño o errores en la implementación de las aplicaciones que posibilitan a los atacantes provocar comportamientos inesperados, distantes de lo establecido en las especificaciones funcionales [1] de dichas aplicaciones. Cuando esto ocurre, a estas debilidades y defectos se les denominará: vulnerabilidades.

Disciplinas como la Seguridad Informática tienen entre sus principales objetivos preservar tres propiedades específicas de la información [2]: (1) la confidencialidad, que exige que la información solo puede ser accedida por las personas autorizadas, (2) la integridad, que plantea que la información solo puede ser modificada por el personal autorizado y (3) la disponibilidad, que demanda que la información se encuentre accesible para las personas autorizadas cada vez que se requiera.

Cuando un atacante explota una vulnerabilidad presente en una aplicación, generalmente, impide que esta sea capaz de preservar las propiedades antes mencionadas para la información que se encuentra bajo su control. En otras ocasiones, la activación de una vulnerabilidad tiene como objetivo, brindar al atacante acceso a otros sistemas a los que no puede acceder directamente.

Algunas de estas vulnerabilidades, son conocidas hace cerca de 20 años. Tal es el caso del “desbordamiento de buffer” que, todavía hoy, encabeza las listas de los principales problemas de seguridad [3]. Esto, es una consecuencia directa de la falta de motivación de las organizaciones productoras de software que, por lo general, son premiadas por los consumidores por agregar nuevas funcionalidades y ser las primeras en el mercado con sus productos, contrario a premiarlas por un software que sea mejor o más seguro [4].

Afortunadamente, este escenario está comenzando a cambiar. Empresas como Microsoft han comenzado a tomar medidas [5] con el objetivo de ganar credibilidad y reconquistar mercados en los que habían sido desplazados (ej. Servidores Web), además, los usuarios han comenzado a volverse exigentes en lo que respecta a la seguridad de sus datos y la protección de su privacidad.

No obstante, todavía muchas organizaciones carecen de políticas que obliguen al desarrollo de aplicaciones seguras. Ya sea por ignorar que sus productos pueden ser afectados por problemas de seguridad, por falta de preparación de los equipos de desarrollo, por la cantidad de recursos que deben dedicar a su capacitación o el esfuerzo adicional que deben de realizar para construir una aplicación segura; lo cierto es, que muchos procesos de desarrollo, todavía no incorporan controles orientados a garantizar la seguridad de la aplicación a lo largo de todo su ciclo de vida [3].

Como consecuencia, se toman determinadas decisiones de diseño o se emplean prácticas incorrectas de programación que introducen defectos o debilidades [3] sin que exista ningún mecanismo para detectarlas a tiempo o evitarlas.

En nuestra institución, por lo general, se producen sistemas de gestión empleando aplicaciones Web y servidores de bases de datos. Dichas aplicaciones, controlan un volumen importante de recursos y procesos vitales, que hacen más eficiente y efectivo el funcionamiento de la institución. Sin embargo, todavía no se han incorporado controles a los procesos de producción que permitan garantizar la construcción de un software más seguro.

Al igual que en otras organizaciones, esto está motivado, fundamentalmente, por la cantidad de recursos y esfuerzos que serán necesarios dedicar a la capacitación del personal y a la modificación de los procesos productivos. Súmese a esto que, después que dichos procesos hayan sido modificados, se hará más complejo concluir satisfactoriamente muchos proyectos de desarrollo de software.

Por los motivos antes mencionados, nos encontramos frente a la siguiente **situación problémica**:

Como no existen actividades o controles incorporados a los ciclos de vida de desarrollo de las aplicaciones, la mayoría de las vulnerabilidades que se introducen en alguna de las fases de este ciclo, no son detectadas. Además, es difícil proteger a dichas aplicaciones de posibles amenazas internas como: introducción de puertas traseras, modificaciones indebidas y no autorizadas del código fuente, entre otras.

Como consecuencia, muchas de las aplicaciones que se han desarrollado, contienen debilidades y defectos que las hacen susceptibles a ataques e incapaces de proteger, adecuadamente, la información y los recursos que controlan. Además, pueden poner en riesgo otros sistemas que se encuentran en su entorno de explotación.

Por tanto, el **problema a resolver** durante el desarrollo de esta tesis será:

Como desarrollar aplicaciones Web más seguras, capaces de proteger la información y los recursos que controlan, sin que las modificaciones que se realicen a los procesos productivos impliquen un incremento notable del esfuerzo requerido para lograrlo.

Como **objeto de estudio** de esta investigación tendremos: los métodos que, incorporados a los procesos de desarrollo, permitan desarrollar aplicaciones Web más seguras.

Dentro de dichos métodos, el **campo de acción** donde se pretende incidir

será: en los patrones de diseño seguro y, más específicamente, en aquellos patrones que puedan ser empleados para aumentar la seguridad de las aplicaciones Web.

El **Objetivo general** del presente trabajo será:

Determinar cómo se pueden desarrollar aplicaciones Web más seguras con la realización de un esfuerzo razonable.

Objetivos específicos

- Determinar que puede ser considerado un software seguro.
- Determinar cuáles son los principales métodos que existen para evitar la aparición de problemas de seguridad en las aplicaciones Web.
- Seleccionar un método que no requiera mucho esfuerzo para su empleo.
- Adaptar el método seleccionado para que pueda ser empleado para el desarrollo de aplicaciones Web.

Tareas

- Investigar los principales métodos que existen para eliminar o reducir los problemas de seguridad en las aplicaciones Web.
- Determinar cuáles son los problemas de seguridad que con mayor frecuencia afectan a las aplicaciones Web.
- Seleccionar los patrones de diseño seguro que puedan contribuir a dar solución a los principales problemas de seguridad en las aplicaciones Web.
- Comprobar la efectividad de los patrones de diseño seguro seleccionados.

Hipótesis de trabajo

Los patrones de diseño seguro, aplicados al desarrollo de aplicaciones

Web, permitirán obtener aplicaciones más seguras requiriendo un esfuerzo menor que con el empleo de otros métodos.

Capítulo 1: Métodos de desarrollo de aplicaciones seguras

1.1 - Introducción

Durante el transcurso del presente capítulo, después de presentar una definición de lo que puede considerarse un software seguro, se expondrán algunos de los métodos que pueden emplearse para lograrlo. Una parte importante de dichos métodos, está representado por un conjunto actividades que pueden incorporarse a las diferentes fases de los ciclos de vida de las aplicaciones.

Otros métodos, como el empleo de los patrones de diseño seguro y las taxonomías de vulnerabilidades, también pueden ser considerados herramientas importantes para identificar y solucionar los principales problemas de seguridad que pueden afectar a una aplicación Web.

El análisis de dichos métodos, permitirá identificar cuáles de ellos pueden hacer un aporte a la seguridad, con un incremento razonable del esfuerzo requerido para lograrlo.

1.2 - Que es un software seguro

En el libro *"Software Security Assurance: A state of the art report (SOAR)"* [6] como resultado del estudio de varias definiciones dadas por diferentes instituciones, se llega a una definición general de lo que puede considerarse un software seguro:

"Un software seguro es aquel que no puede ser subvertido o forzado a fallar intencionalmente. Es un software que se mantiene correcto y predecible a pesar de los esfuerzos de comprometer su fiabilidad."

El software seguro es diseñado, implementado, configurado y soportado de forma tal que le posibilite:

- Continuar operando correctamente en la presencia de ataques ya sea resistiendo los intentos de explotarlo, provocar fallas u otras debilidades, o tolerando los errores y las fallas que resulten de esos intentos de explotarlo.

- Aislar, contener y limitar el daño resultante de cualquier falla provocada por ataques que el software fue incapaz de resistir o tolerar, y recuperarse tan rápido como sea posible de estas fallas.

1.3 - Actividades que contribuyen a desarrollar un software seguro

Existe consenso entre varios autores [7, 8, 9] de que no existe una única herramienta capaz de producir un software seguro y que, para obtenerlo, es necesario incorporar un conjunto de actividades en cada fase del ciclo de vida, orientadas a evitar la introducción de vulnerabilidades que puedan afectar la seguridad de la aplicación.

Siguiendo esta tendencia, han surgido varias metodologías como CLASP [8] (*Comprehensive Lightweight Application Security Process*) y Microsoft SDL [5] (*Security Development Lifecycle*) que agrupan a varias de dichas actividades, ubicándolas en la fase del ciclo de vida a la que corresponden y ofreciendo guías para realizarlas. Son, por lo general, independientes del proceso de desarrollo que se emplee.

Incorporar dichas metodologías a los procesos de desarrollo existentes en las instituciones, puede implicar un aumento importante del trabajo a realizar para la construcción de un aplicación [6]. En el caso de CLASP, por ejemplo, se proponen 24 nuevas actividades y 8 roles.

Por consiguiente, será necesario ser objetivos y emplear estas actividades de forma racional, teniendo en cuenta, que es imposible lograr un sistema cien por ciento seguro [3] y que, cada paso que se dé para alcanzar este objetivo, traerá aparejado un incremento en el costo del proyecto.

Siguiendo esta línea de pensamiento, el autor Gary McGraw en su libro "*Software Security: Building Security In*" [10] considera, que la seguridad debe de ser un ejercicio de gestión de riesgos que permita, al equipo de desarrollo, determinar el punto intermedio entre el gasto destinado a asegurar la aplicación y las pérdidas que podrá ocasionar la aparición de un problema de seguridad.

En el mismo libro, este autor, como resultado de un estudio de varias metodologías de desarrollo de software seguro, identifica siete actividades a las que considera como los puntos de contacto entre estas metodologías. Estas actividades, ordenadas atendiendo a su importancia, son: Revisión de código, Análisis de riesgos, Pruebas de penetración, Pruebas unitarias basadas en riesgos, Casos de abuso, Requisitos de seguridad y Operación segura [10].

Veamos a continuación la descripción de algunas de dichas actividades.

1.3.1 - Requisitos de seguridad

En la ingeniería de software cuando se habla de requisitos se hace referencia a una condición o necesidad de un usuario que debe estar presente en un sistema o componentes del sistema para satisfacer un contrato, estándar, especificación u otro documento formal [11].

En el caso de los requisitos de seguridad, estos se originan cuando, los interesados en un proyecto, establecen que algunos de los objetos del sistema que pueden ser tangibles (dinero en efectivo, medios materiales) o intangible (información), poseen algún valor y, por lo tanto, deben de ser protegidos [12].

El objetivo principal de capturar o determinar estos requisitos, es el de conceptualizar la seguridad de la aplicación en desarrollo desde el comienzo y de servir como referencia para los procesos subsiguientes.

El proceso de captura de requisitos en su forma tradicional, consiste en un conjunto de reuniones que se realizan con los interesados en el proyecto donde, a partir de un conjunto de técnicas (ej. "*Accelerated Requirements Method* [13]"), se definen los objetivos de seguridad con los que debe cumplir la aplicación (ej. un director de recursos humanos podría exigir la confidencialidad de la información relativa al personal que trabaja en la empresa). Concluida la captura, se procede a categorizar y a establecer prioridades para la satisfacción de los requisitos capturados.

Existen otras vías para llevar a cabo este proceso que eliminan los problemas asociados a la interacción con personas (diferencias en las definiciones, problemas de comunicación, grado de conocimientos, etc.).

Metodologías que basan la determinación de los requisitos de seguridad en los análisis de las interacciones entre los recursos del sistema y los roles del mismo [14]. A partir de los resultados de este análisis, se obtienen los requisitos de seguridad que, se materializarán como mecanismos de seguridad, en fases superiores del desarrollo de la aplicación.

Se hace bastante énfasis en la realización de este proceso, pues varios estudios han mostrado que, corregir errores en los requisitos cuando el software ya se encuentra en la fase de explotación, cuesta entre 10 y 200 veces más que cuando estos errores son detectados en la fase de análisis [15]. Agréguese a esto, el reconocimiento creciente, por parte de la industria del software, de que la ingeniería de requisitos es crítica en la culminación exitosa de cualquier proyecto de desarrollo de software.

Son evidentes las ventajas que acarrea la correcta determinación de los requisitos de seguridad pero, también ha de señalarse, que requieren la dedicación de recursos y tiempo, debido, fundamentalmente, a que la mayoría de los requisitos de seguridad son requisitos no funcionales y dependientes del contexto en el que se emplee la aplicación, lo que los hace complejos de determinar.

Súmese a lo anterior, que los requisitos deben de contar con determinadas propiedades para que puedan ser útiles [16] en la realización de los restantes procesos del ciclo de vida de un proyecto de software. Una de estas propiedades, demanda que los requisitos deben de ser medibles, o lo que es lo mismo, que debe ser posible comprobar cuando un requisito ha sido o no satisfecho. Pero, en el caso de los requisitos de seguridad, no existen por lo general criterios simples de satisfacción como un sí o un no [17], lo que obliga a los desarrolladores a realizar ciertas suposiciones para que los requisitos de seguridad puedan ser medibles. Suposiciones que, en caso de no cumplirse o ser incorrectas, pueden generar problemas de seguridad.

1.3.2 - Casos de abuso

Los casos de uso, empleados en las metodologías de desarrollo de

software tradicionales, reflejan los objetivos perseguidos por el usuario en la forma de funcionalidades que brindará el sistema para que estos puedan realizar su trabajo. Sirven, además, para documentar los escenarios de uso normal del sistema [11].

Los casos de abuso, persiguen un objetivo similar pero, describiendo los escenarios de empleo del sistema, desde el punto de vista de un atacante. Su descripción, constituye una documentación de las acciones malintencionadas que podría llevar a cabo, un intruso, en los diferentes escenarios [15]. Pueden ser vistos, también, como un conjunto de operaciones que deberán estar estrictamente restringidas para los usuarios.

Para su determinación, se analizarán los casos de uso del sistema en construcción desde el punto de vista de un atacante, intentando encontrar, funcionalidades que permitan hacer un empleo incorrecto del sistema. Implica cuestionar, cada elemento de dicho sistema, con preguntas tales como: ¿Cómo puede el sistema determinar que una solicitud proviene de un usuario legítimo del sistema y no de un atacante?, ¿Cómo puede el sistema distinguir entre buenos y malos parámetros de entrada?

Durante la realización de este proceso, puede ser de gran utilidad apoyarse en la realización de tormentas de ideas donde, los diseñadores asumiendo el rol de los atacantes, ataquen virtualmente al sistema, apoyándose en los problemas conocidos de seguridad para evaluar la fortaleza de la aplicación. Es posible, también, analizar la forma en que se accede a los recursos del sistema, las relaciones de estos accesos con los mecanismos de seguridad [8] o aquellos elementos, que la mayoría de los desarrolladores pasan por alto, asumiendo que no pueden o no serán hechos (ej. "los usuarios no entienden el formato de la cache, por lo que, no pueden modificarla").

Los resultados de este proceso, serán empleados para enriquecer los requisitos de seguridad del sistema y como ayuda para la posterior selección de los mecanismos de defensa contra los problemas potenciales detectados [10]. Contribuyen a diferenciar cuál es el empleo apropiado del sistema y cual no lo es,

dejándolo claramente documentado para su utilización como guía en fases sucesivas.

Su correcta determinación, exige gran experiencia en el campo de la seguridad por parte de aquellos que participen en el proceso. Exige además, salvar las dificultades que afrontan los diseñadores de un sistema para observarlo desde la postura de un atacante.

1.3.3 - Análisis de riesgo de arquitectura y diseño

El análisis de riesgos, es una técnica de ingeniería cuyo objetivo es ayudar a identificar y planificar la mejor manera de mitigar las posibles amenazas a las que deberá enfrentarse una aplicación o sistema informático. Permitirá a las organizaciones tomar decisiones más efectivas y eficientes en cuanto al empleo de sus limitados recursos (tiempo, personal, dinero) [18].

Para la realización de este proceso, la aplicación será dividida en sus partes componentes y serán analizados, cada uno de sus puntos de entrada y flujos de datos, con el objetivo de identificar posibles debilidades del sistema [9]. La realización de este análisis, se apoyará en modelos de amenazas y listados de ataques conocidos.

Este momento del proceso, permitirá encontrar problemas tanto, en la lógica del negocio como en la arquitectura del sistema [19]. Debilidades de diseño, conflictos en los requerimientos, interfaces expuestas, omisiones a la hora de forzar las políticas de seguridad, etc.

Identificadas, las posibles amenazas, se realiza un proceso de catalogación de las mismas, atendiendo a: la facilidad para encontrarlas, los conocimientos requeridos, el impacto para la seguridad del sistema, entre otros aspectos [5]. El resultado final, es un listado ordenado, según el riesgo, de los problemas de seguridad potenciales, a los que podrá estar expuesta la aplicación.

Dicho listado, ayudará al equipo de desarrollo a determinar, cuáles riesgos mitigar y cuáles no, permitiéndole enfocarse en la creación de mecanismos de seguridad para aquellas amenazas que verdaderamente requieran de atención, en

vez de desgastarse tratando de dar solución a todos los problemas o a problemas complejos, cuya solución, no aporta prácticamente nada a la seguridad del sistema.

Estos resultados, también enriquecerán la lista de requisitos de seguridad del sistema, contribuirán a la reducción del costo del proyecto, por concepto de corrección de problemas de seguridad, y permitirán la creación de pruebas de seguridad más eficaces.

Entre sus inconvenientes, se puede mencionar, que implican un volumen de trabajo importante, además, requieren de gran experiencia y conocimientos de seguridad para el personal que los realice [10]. De estos aspectos, dependerá, en gran medida, la calidad de los resultados.

1.3.4 - Revisión de código

Tal y como su nombre lo indica, la revisión de código, consiste en analizar de forma manual o automática el código fuente de la aplicación en desarrollo, intentando encontrar, errores de implementación que deriven en problemas de seguridad.

Se le presta gran atención a este proceso, pues varios autores [20, 21, 22] consideran, que es en el código del programa donde se introducen cerca del 50% de las vulnerabilidades. Por lo tanto, detectarlas mediante la revisión del código de la aplicación, puede constituir una notable contribución al incremento de la seguridad del sistema.

La inspección manual de código, permite encontrar, además de construcciones riesgosas, otros problemas de implementación relacionados con la lógica de la aplicación (ej. “puertas traseras”). Este tipo de auditoría, es altamente consumidora de tiempo. Según estadísticas de Microsoft, un especialista entrenado puede revisar 1500 líneas de código diarias de C o 1000 de C++ [5]. Ir más rápido, solamente degradaría la calidad del proceso.

Si se tiene en cuenta los millones de líneas de código con que cuenta cualquier programa hoy en día y la exigencia, para el personal que vaya a efectuar la tarea, de poseer una alta preparación acerca de los problemas de seguridad

que pueden encontrarse en la implementación de una aplicación, antes de comenzar a examinarla, la revisión manual de código fuente, se convierte en una tarea prácticamente imposible de realizar.

Una alternativa viable, puede encontrarse en el empleo de herramientas de revisión automática de código. Estas herramientas emplean bases de datos, en las que se incluyen los diferentes tipos de vulnerabilidades, y pueden reconocerlas a partir de un análisis estático del código. Estas bases de datos, pueden actualizarse en la misma medida en la que las vulnerabilidades aparecen y, mediante un proceso de reinspección, determinar si existen afectaciones.

No son perfectas, por lo que, cada análisis vendrá generalmente acompañado de un grupo de falsos positivos y falsos negativos. Los primeros, representaran una pérdida de tiempo. Los segundos, representarán vulnerabilidades existentes en la aplicación, que pasarán desapercibidas para la herramienta, lo que puede crear una falsa sensación de seguridad [23].

Otro problema con estas herramientas, es la dificultad que entraña lograr, que sean capaces de entender el objetivo para el cual se crea un determinado programa [24]. Aspecto, en el que se encuentran en desventaja respecto al especialista humano, que estará en una mejor posición para comprender la lógica de la aplicación.

1.3.5 - Pruebas de penetración

Las pruebas de penetración (*penetration test* o *pentest*), se realizan para verificar, el correcto funcionamiento del sistema desde el punto de vista de seguridad, en su entorno de operación. Para llevarlas a cabo, se ejecutarán contra la aplicación acciones propias de los atacantes, cuyo objetivo será, el de intentar evadir los mecanismos de seguridad o detectar alguna vulnerabilidad, existente en el sistema, que le posibilite a un atacante tomar ventaja del mismo [22].

Además de contribuir a detectar problemas de seguridad, permiten demostrarlos de manera concreta y valorar la magnitud de los mismos. Algo que no es posible lograr con los resultados abstractos de un análisis de riesgo.

Como su objeto de análisis son las aplicaciones, su empleo queda

prácticamente relegado a las fases finales del ciclo de vida. Fases, en las que intentar corregir los errores detectados implicará, por lo general, rediseñar y reprogramar desde pequeños componentes hasta la aplicación completa. Esto último, puede traducirse como un incremento significativo del costo del proyecto.

Por esta causa, se hace hincapié en que, las comprobaciones de seguridad, no deben ser consideradas, el punto de partida, de una entidad que pretenda desarrollar un software seguro. Debe de considerársele, como el último elemento dentro del ciclo de vida de la seguridad, que será aplicado al sistema para verificar que no existen problemas asociados a errores de configuración [25].

Tiene como limitantes, al igual que otras herramientas, la alta dependencia de los resultados de estas pruebas con la habilidad y experiencia del personal que las lleva a cabo. Es un proceso que, generalmente, consume grandes cantidades de tiempo cuando se realiza de forma manual y, para el cual, las herramientas automáticas que se han desarrollado, no han tenido demasiado impacto [24].

Es importante mencionar además, que no existe garantía de que una aplicación que haya pasado exitosamente una comprobación de seguridad, sea segura. Esto, es consecuencia directa del problema que entraña la comprobación de negativos (ej. "la aplicación no tiene vulnerabilidades de desbordamiento de buffer") que, por lo general, es un reto mucho mayor que la comprobación de positivos (ej. "al introducir la contraseña correcta, el usuario puede ingresar al sistema").

Cuando se realizan comprobaciones de positivo, puede obtenerse un alto grado de confianza de que el componente de software ejecutará la funcionalidad tal y como se desea. Sin embargo, a la hora de comprobar los negativos, el número de acciones a realizar para detectar las posibles fallas, puede llegar a ser infinito. Si se tiene en cuenta que, las comprobaciones solo pueden realizar un conjunto finito de estas acciones, la tarea de demostrar una aplicación es segura, es prácticamente imposible. Lo único que puede ser demostrado a partir de las comprobaciones de seguridad es que, para determinadas acciones dentro de

determinadas condiciones de entorno, no es posible encontrar ningún problema de seguridad [3].

Como ha podido apreciarse, casi todas las actividades analizadas son altamente consumidoras de tiempo, requieren de la realización de un esfuerzo considerable y del empleo de personal con conocimientos y experiencias en seguridad para obtener resultados relevantes.

1.4 - Patrones de diseño seguro

Ante los problemas que deben enfrentar los equipos de desarrollo para incorporar las metodologías de diseño seguro de software y sus actividades, aparece como alternativa el empleo de los "Patrones de diseño seguro". Estos patrones, pueden reducir de forma significativa, el esfuerzo que se requiere para desarrollar aplicaciones seguras, así como, el tiempo y los recursos necesarios para lograrlo [26]. Lo que a su vez se revierte, en una disminución del costo del proyecto.

Los patrones se definen como: "una solución genérica, bien probada, que resuelve un problema recurrente que aparece en un contexto dado" [27]. Surgen de la idea, de que es poco común que cada nuevo proyecto de desarrollo, tenga que enfrentar problemas de seguridad, que demanden soluciones verdaderamente novedosas [28]. Lo que generalmente ocurre, es que los desarrolladores intentan recordar problemas similares que fueron resueltos de forma satisfactoria en otras situaciones, reutilizar su esencia y adaptar los detalles para resolver un nuevo problema [28].

Cuando la solución a estos problemas se encapsula en la forma de un patrón y se emplea para reducir las debilidades y vulnerabilidades presentes en una aplicación, se estará aplicando la experiencia colectiva de diseñadores experimentados en seguridad. Esto permitirá, a los principiantes, actuar como expertos en seguridad y evitará que se improvisen los mecanismos de seguridad [29].

Otra ventaja del empleo de los patrones para la seguridad, es que contribuyen a aplicar algunos de los “principios de diseño seguro”, al diseño del proyecto en desarrollo [30]. Estos principios, se ofrecen como lineamientos en varias metodologías, libros y artículos de seguridad. Tienen como objetivo, obtener el diseño de una aplicación más segura y resistente a ataques. Sin embargo, carecen de una actividad bien definida que sirva como guía para su incorporación al diseño de la aplicación.

Aunque tratan de proveer una solución auto-contenida a un problema, los patrones no son independientes entre sí. Existen varias relaciones entre ellos, una de las más importantes, es el refinamiento, que permite que, la solución propuesta por un patrón, pueda ser implementada con la ayuda de otros patrones para resolver subproblemas del problema original. Sin ellas, los patrones verían limitado su empleo a resolver problemas aislados, sin grandes efectos en el diseño o la arquitectura completa de un software [28].

La principal dificultad que entraña, el empleo de los patrones, es que los equipos de desarrollo deberán ser capaces de identificar los contextos en los que dichos patrones deben ser aplicados. Otra dificultad, puede encontrarse a la hora de decidir cuál será la estrategia de implementación más adecuada a utilizar. Por lo que, requerirán cierto entrenamiento previo a su empleo.

1.4.1 - Repositorios de patrones

Una forma muy común de encontrar a los patrones, es agrupados en la forma de repositorios donde, en ocasiones, es posible hallar decenas de estos adecuadamente clasificados y con abundante información sobre su intención, motivación, estructura, comportamiento, estrategias de implementación, entre otros aspectos. En algunos casos, es posible hallar también ejemplos de código.

En el libro “*Security Patterns – Integrating security and system engineering*” [28], Markus Schumacher aporta más de treinta patrones que ofrecen soluciones para la incorporación de mecanismos de seguridad tales como: la identificación y autenticación, el control de acceso, la autorización y la auditoría. Además, pueden

encontrarse patrones para construir aplicaciones seguras destinadas a ser empleadas en Internet y patrones para la construcción de cortafuegos.

Si bien los patrones recogidos en el repositorio antes mencionado son bastante generales y pueden aplicarse a diferentes tipos de aplicaciones, los libros "*Security Patterns Repository – Version 1.0*" [31] y "*Core Security Patterns: Best Practices and Strategies for J2EE™, Web Services, and Identity Management*" [32] ofrecen, cada uno, más de veinte patrones dirigidos, específicamente, a las aplicaciones Web.

Otros libros como: "*OWASP ASDR: The Application Security Desk Reference*" [33] y "*A Guide to Building Secure Web Applications and Web Services*" [34], aunque no emplean las plantillas tradicionales para definirlos, también ofrecen numerosos patrones destinados a reducir los problemas de seguridad presentes en las aplicaciones Web.

Como ha podido apreciarse, existen varios repositorios de patrones disponibles. Esto implica que los desarrolladores, además de tener que lidiar con los problemas asociados a su empleo, deberán conocer los patrones que componen a cada uno de los repositorios y salvar las diferencias de nomenclatura que existe entre ellos.

1.5 - Taxonomías de vulnerabilidades

Como se mencionó durante la introducción de este trabajo, las vulnerabilidades se encuentran entre las causas principales de los problemas de seguridad que afectan a las aplicaciones. Conocerlas y saber identificarlas, puede contribuir a que los equipos de desarrollo sean capaces de detectarlas tempranamente y puedan emprender acciones para su mitigación.

Por ello, se han hecho numerosos esfuerzos con el objetivo de facilitar la identificación de los posibles defectos y debilidades que pueden introducirse en el proceso de desarrollo de un software, constituyendo esta, una de las áreas más activas en las investigaciones de seguridad del software [6].

Existen numerosas taxonomías de vulnerabilidades. Algunas de ellas, como "*Comprehensive Lightweight Application Security Process (CLASP) Vulnerability Root Cause Classification*" [8] y "*CVE: Common Vulnerabilities and Exposures*" [37], tienen como característica contar con varias categorías cargadas con la descripción de numerosas vulnerabilidades.

Seguir la pista a todas las vulnerabilidades descritas por este tipo de taxonomías, sería el ideal para lograr un software prácticamente libre de problemas de seguridad. Sin embargo, esta aproximación puede tener grandes implicaciones en la duración y el costo del proyecto. Además, por lo general, no cuentan con una guía clara que permita establecer prioridades a la hora de enfrentar los diferentes tipos de problemas. Como consecuencia, los desarrolladores por desconocimiento, pueden dedicar un gran volumen de tiempo y esfuerzos para tratar de solucionar problemas de poca relevancia para la seguridad del sistema y prestar, una atención superficial, a otros de mayor envergadura.

Otras taxonomías siguen un enfoque diferente. Seleccionan y clasifican las vulnerabilidades atendiendo a su frecuencia de aparición y a las consecuencias potenciales, en caso de que sean descubiertas y explotadas. Constituyen una alternativa más viable, reduciendo considerablemente el esfuerzo y enfocándolo a los problemas de seguridad que poseen mayor relevancia.

Una de estas taxonomías es "*19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*" [35]. En ella, se relacionan los 19 errores de seguridad que más comúnmente se cometen en el desarrollo de aplicaciones. Su selección se basa en un reporte del "*National Cyber Security Division at the U.S. Department of Homeland Security*" donde se plantea, que estos 19 errores, representan la causa del 95% de los problemas de seguridad presentes en las aplicaciones.

Otra de estas taxonomías, es "*OWASP Top 10: The Ten Most Critical Web Application Security Vulnerabilities 2007*" [36]. Surge de un estudio realizado por la *"Open Web Application Security Project"* (OWASP), donde se encuentran

recogidas, por orden de prevalencia, las vulnerabilidades más importantes de las aplicaciones web y se han agrupado en 10 categorías.

Ambas taxonomías poseen dificultades en las definiciones de las categorías a las que pertenecen las vulnerabilidades catalogadas y, en el caso de la segunda, las descripciones son de alto nivel, lo que reduce, en cierto grado, su utilidad [37].

1.5.1 - 2009 MITRE SANS Top 25

Con el objetivo de resolver estos problemas y unificar las diferentes taxonomías existentes para evitar que los usuarios deban dedicar un tiempo y un esfuerzo significativo a seleccionar, correlacionar y fusionar diferentes nombres, definiciones y clasificaciones asignadas a las mismas vulnerabilidades en diferentes taxonomías, surge el proyecto "MITRE CWE (*Common Weakness Ennumeration*)" [38].

Este proyecto, intenta proveer un diccionario donde se incluyan las 20 000 vulnerabilidades de un proyecto precedente muy exitoso: CVE (*Common Vulnerabilities and Exposures*), organizadas en categorías que faciliten su identificación. En él, se ha efectuado un trabajo de normalización y solución de conflictos para proveer un lenguaje común que permita describir, de manera estándar, las debilidades de seguridad. Ofrece además, un listado de los 25 problemas más importantes de seguridad "*2009 CWE/SANS Top 25 Most Dangerous Programming Errors*" [39], de los cuales, la gran mayoría, es posible encontrarlos en las aplicaciones Web.

Este listado, surge como resultado de un estudio realizado a las vulnerabilidades presentes en la taxonomía "MITRE CWE", donde se evaluó la facilidad para descubrirlas y explotarlas, el nivel de conocimientos requeridos para lograrlo, el impacto a la seguridad del sistema en caso de ser explotadas y cuan frecuente era su aparición en las aplicaciones.

Por sus características, este listado será empleado durante el desarrollo del presente trabajo, como una guía que permita seleccionar, entre los diferentes repositorios de patrones de seguridad mencionados, que patrones son los más

adecuados para mitigar o evitar la aparición de estas vulnerabilidades, reduciéndolos a una cantidad mínima que, a la vez que permita evitar los principales problemas de seguridad y reducir el esfuerzo que deberán realizar los equipos de desarrollo para crear aplicaciones Web más seguras.

1.6 - Conclusiones

Durante el presente capítulo, se ha realizado un breve recorrido por los principales métodos empleados para producir aplicaciones más seguras. Uno de los métodos más promisorios, es el de incorporar un conjunto de actividades a cada una de las diferentes fases del ciclo de vida de un proyecto de software, con el objetivo de garantizar que la seguridad sea tenida en cuenta a la hora de concebir, diseñar, implementar y explotar el software. Varias metodologías, han incorporado, en mayor o menor grado, varias de estas actividades.

El principal problema que enfrenta un equipo de desarrollo que las emplee, es el considerable esfuerzo que se deberá realizar para aplicarlas y para alcanzar los conocimientos y la experiencia que permitan emplearlas de forma efectiva. Requieren, además, que se les dedique tiempo y recursos.

Una alternativa a las metodologías de desarrollo de software seguro, se encuentra en el empleo de los patrones de diseño seguro que, aunque no llegan al nivel de dichas metodologías para garantizar la seguridad de una aplicación, pueden dotarla de una seguridad razonable con la realización de un esfuerzo mucho menor. Su principal problema está, en que los equipos de desarrollo deben conocer varios patrones y saber reconocer el contexto donde estos pueden ser aplicados. Tareas, que no siempre son sencillas y que pueden complejizarse, en la misma medida en la que se incrementan el número de patrones presentes en los repositorios.

Las taxonomías de vulnerabilidades, pueden emplearse como guías que les permitan a los desarrolladores, detectar los problemas de seguridad que han sido introducidos durante el desarrollo de sus aplicaciones. Tienen el inconveniente, de que en ocasiones son muy extensas y complejas y, por lo general, no ofrecen una

guía clara de cómo resolver los problemas de seguridad en ellas recogidos. Aspecto este, que puede limitar en cierta medida su utilidad.

Estas taxonomías, pueden emplearse para determinar que patrones, de entre todos los presentes en los diferentes repositorios, serán los más convenientes para reducir los problemas de seguridad más comunes de las aplicaciones Web.

Capítulo 2: Patrones de diseño seguro

2.1 Introducción

Durante el desarrollo de este capítulo, se analizarán las vías o mecanismos que pueden emplearse para eliminar algunas de las 25 vulnerabilidades propuestas en el listado "*2009 CWE/SANS Top 25 Most Dangerous Programming Errors*" y reducir las posibilidades de los atacantes para explotar otras.

A partir de la determinación de estos mecanismos, se seleccionarán un conjunto de patrones de diseño seguro, cuya descripción y comportamiento, se corresponda con los mecanismos identificados. Además, se analizarán otros que contribuirán a hacer más eficaz el aporte, de los patrones antes mencionados, y contribuirán a brindar una mayor resistencia a ataques a la aplicación.

2.2 Mecanismos para evitar o reducir las posibilidades de ataque de las principales vulnerabilidades

Para determinar qué mecanismos pueden emplearse para reducir la cantidad de vulnerabilidades presentes en una aplicación y hacerla más resistente a los ataques, dirigidos a explotar aquellas vulnerabilidades que no hayan podido ser eliminadas, se analizaron cada una de las 25 vulnerabilidades relacionadas en el listado "*2009 CWE/SANS Top 25 Most Dangerous Programming Errors*" para identificar mecanismos de diseño, que pudieran contribuir a la elaboración de una aplicación más segura.

A partir de los resultados de este análisis, que se encuentran recogidos en el Anexo 1, se seleccionaron cinco mecanismos que, atendiendo a sus características y a las de las vulnerabilidades analizadas, son los que pueden realizan una mayor contribución a la seguridad de una aplicación.

Los mecanismos seleccionados son:

- M1: Correcta validación de los datos
- M2: Escape de las salidas
- M3: Empleo de intermediarios (*proxy*)

- M4: Empleo de un canal seguro para la transmisión de los datos
- M5: Identificador de autenticidad de las solicitudes

Estos mecanismos, aparecen representados en la Tabla 1 (columnas), con el objetivo de mostrar su posible contribución en la eliminación de las vulnerabilidades en estudio (filas). El aporte de cada mecanismo, se representará empleando una letra (D) cuando su incorporación al diseño de una aplicación, contribuya de forma directa a evitar la vulnerabilidad, o sea, la elimine. En aquellos casos, en los que el mecanismo no elimine la vulnerabilidad pero pueda reducir la posibilidad de éxito de alguno de los ataques que se emplean para explotarla, aparecerá una letra (I).

Vuln	M1	M2	M3	M4	M5	Vuln	M1	M2	M3	M4	M5
CWE-20	D					CWE-94	D	D	D	I	I
CWE-116		D				CWE-494				I	
CWE-89	D	D	D	I	I	CWE-404					
CWE-79	D	D		I		CWE-665					
CWE-78	D	D	D	I	I	CWE-682	I			I	
CWE-319				D		CWE-285					
CWE-352		I		I	D	CWE-327					
CWE-362						CWE-259	I		I		I
CWE-209				I		CWE-732	I				
CWE-119	I					CWE-330					
CWE-642				I		CWE-250					
CWE-73	D		D	I	I	CWE-602	D	D			D
CWE-426	I										

Tabla 1: Efecto sobre las vulnerabilidades de los mecanismos seleccionados.

2.3 Patrones de diseño seguro que corresponden a los mecanismos de seguridad seleccionados

Partiendo de la descripción de los mecanismos seleccionados, se eligieron, entre los patrones presentes en algunos de los repositorios, cinco, cuyas descripciones y comportamiento, se corresponden con la de dichos mecanismos.

Estos patrones, serán brevemente descritos a continuación, empleando la plantilla: “Intención”, “Motivación”, “Estructura”, “Participantes”, “Comportamiento” y “Estrategias”.

2.3.1 Patrón: Validador Interceptor

Intención

Garantizar que los datos provenientes de cualquier entidad externa, cumplan con un conjunto de reglas de sintaxis, tipo de dato, longitud, negocio, etc. [40].

Motivación

Es mediante el intercambio de información con el sistema, la forma en que un atacante, por lo general, interactúa con el mismo y, por lo tanto, varias estrategias de ataques, intentan comprometer el sistema mediante el envío de solicitudes con datos inválidos o códigos maliciosos [28].

Contar con un mecanismo simple y flexible, que permita detectar malformaciones en los datos y limitar las posibles entradas, solamente a aquellas esperadas por el sistema, puede contribuir, en gran medida, a reducir las posibilidades de un atacante.

Estructura

En la Fig. 1 (Anexo 2), aparece la estructura estática de este patrón.

Participantes

Cliente: Representa a todas las entidades externas que interactúan con la aplicación (usuarios, servicios, etc.).

Objetivo: Componente interno del sistema, encargado de dar respuesta a la solicitud del cliente.

Validador: Encargado de verificar que se cumplan las reglas definidas para

un tipo de datos específico.

Validador Interceptor: Es el encargado de determinar que validadores deben ser empleados para verificar una solicitud.

Aplicación: Encargada de recibir y dar respuesta a las solicitudes del cliente.

Comportamiento

Cuando un cliente hace una solicitud al sistema, esta es interceptada por el "Validador Interceptor". Si existe alguna regla de validación para la solicitud entrante, invocará al mecanismo o a los mecanismos de validación correspondientes para determinar si no existen problemas en: la estructura, sintaxis, tipo de datos, contenido, etc. En caso de que no existan problemas, permite el paso de la solicitud hacia su objetivo Fig. 2 (Anexo 2). Caso contrario, la descarta Fig. 3 (Anexo 2).

Estrategias

Las validaciones del lado del cliente, son inherentemente inseguras. Es relativamente fácil manipular una página Web y evadir cualquier mecanismo de validación de la página original. Para que el empleo de este patrón sea considerado seguro, la validación deberá de realizarse del lado del servidor [34].

La validación del lado del cliente, tiene sentido para las reglas de negocio, brindándole retroalimentación de los errores cometidos al usuario, antes de que la solicitud, sea enviada. Esto aumenta el rendimiento percibido y ahorra tiempo de procesamiento, por concepto de errores, en el servidor [33]. Pueden contribuir, además, a la detección de intrusos. Una solicitud que fue previamente validada en el cliente, no tiene ninguna razón para generar errores si se le aplica el mismo mecanismo de validación en el servidor. Si esto ocurre, puede considerarse con un porciento elevado de certeza, que la aplicación se encuentra bajo ataque.

En el lado del servidor, este patrón puede ubicarse a la entrada de cada una de las operaciones que realiza la aplicación. Esta variante de implementación, implica tener un número importante de mecanismos de validación diseminados por

el sistema y, como consecuencia, pueden quedar lugares donde: no hayan sido aplicados, sus implementaciones difieran o no se encuentren debidamente actualizadas.

Una aproximación más centralizada, colocando todos los mecanismos de validación en un punto que constituya, paso obligatorio de todas las solicitudes que llegan a la aplicación dentro de la frontera de confianza establecida en el diseño, puede reducir, eficazmente, los problemas mencionados con anterioridad.

Por lo general, el principal vector de ataque de una aplicación Web, lo constituyen las entradas provenientes del usuario [41], no obstante, para cualquiera sea la variante de implementación que se emplee, será necesario recordar que, tanto los ficheros de configuración, las variables de entorno y otras fuentes de datos externas a la aplicación [38], también pueden constituir vectores de ataque y, por tanto, será conveniente validarlas antes de su empleo.

2.3.2 Patrón: Escape de las salidas

Intención

Evitar que, caracteres o información de control presentes en los datos, tengan un significado especial para el intérprete hacia el que van dirigidos [42].

Motivación

De forma general, los navegadores de los usuarios, la *shell* del sistema, el servidor de bases de datos, etc. son, en esencia, intérpretes de comandos que ejecutan determinadas operaciones, ante la presencia de ciertos caracteres o información de control, en los datos que reciben.

Si estos caracteres o información, no son convenientemente escapados, un atacante tendrá la posibilidad de llevar a cabo acciones que modifiquen la lógica de la aplicación como: la ejecución de código malicioso [43], la inyección SQL [35], la inyección de comandos del sistema [44], entre otras.

Contar con un mecanismo, que se encargue de eliminar el significado especial de dichos caracteres y expresiones para toda la información que va dirigida hacia algún tipo de intérprete, puede contribuir a reducir las posibilidades de éxito de varios tipos de ataques.

Estructura

En la Fig. 4 (Anexo 2), aparece la estructura estática de este patrón.

Participantes

Componente: Elemento de la aplicación que, a partir de los datos que recibe del usuario o de otros componentes, genera información que puede contener elementos peligrosos para los intérpretes hacia los que va dirigida.

Aplicación: Encargada de recibir y dar respuesta a las solicitudes del cliente.

Destino: Elemento del sistema, a quien va dirigida la información generada por el componente. Puede ser el navegador del usuario, un programa del sistema operativo, el gestor de bases de datos, etc.

Escape Concreto: Contiene un grupo de reglas específicas, que permiten escapar, todos aquellos elementos que poseen un significado especial para un intérprete determinado (ej. HTML o SQL).

Escape de las Salidas: Encargado de ejecutar el “Escape Concreto”, que requiere la información procesada.

Comportamiento

Cuando el cliente realiza una solicitud a la aplicación, esta entrega la solicitud al componente responsable de procesarla. El componente, como resultado del proceso, genera información que será entregada a “Escape de las Salidas”, el que, dependiendo del destino de dicha información, ejecutará algún “Escape Concreto”. Realizada esta operación, la información regresa a la aplicación, que la empleará para preparar la respuesta y entregársela al cliente Fig. 5 (Anexo 2).

Estrategias

Es posible aplicar este patrón, a todas aquellas salidas generadas por componentes que, posteriormente, serán utilizadas por algún tipo de intérprete. Sin embargo, es bastante complejo mantener actualizados todos los mecanismos de este tipo que se encuentren diseminados por el sistema, en la misma medida en la que este se desarrolla y los flujos de información cambian. Como resultado, pueden aparecer casos en los que, determinados flujos de información, no se encuentren convenientemente escapados atendiendo al componente hacia el que van dirigidos.

Una aproximación más efectiva, sería la de centralizar los mecanismos de escapes de las salidas en un punto de la aplicación, donde ocurra el paso obligado, de la información que fluye hacia los componentes y desde ellos. En este punto, se aplicarían los procedimientos de escapes correspondientes, según requiera el componente de destino.

Para cualquiera de los casos, es aconsejable emplear las funciones de escapes que provee la plataforma en la que se esté trabajando (ej. “htmlentities()” de PHP [45]), siempre que existan. Contrario a realizar una implementación desde cero de los algoritmos de escape.

2.3.3 Patrón: Proxy confiable

Intención

Proveer una interface segura para acceder a: componentes externos, de terceros o a recursos empleados por el sistema, mediante la restricción de las funcionalidades que pueden ser accedidas o forzando el cumplimiento del protocolo requerido para acceder a los mismos [32].

Motivación

Generalmente, es necesario exponer un conjunto de componentes, que no se encuentran adecuadamente protegidos, a la interacción con los usuarios. Estos componentes, por lo general, son fabricados por terceros (ej. html2pdf), complejos y poseen muchas funcionalidades que pueden ser objeto de abuso, empleo malicioso o pueden contener debilidades.

En el caso de los recursos, es común que se expongan accesos directos a los mismos, dándole la posibilidad a un atacante, de manipularlos para acceder a otros recursos protegidos o confidenciales del sistema que, de otra forma, no podrían ser accedidos.

El "Proxy confiable", actúa como un intermediario. Intercepta y filtra todas las comunicaciones entre los usuarios y el componente en cuestión. De esta forma, puede compensar la existencia de debilidades en el componente protegido y garantizar que, las políticas apropiadas, sean hechas cumplir de forma consistente.

Estructura

En la Fig. 6 (Anexo 2), aparece la estructura estática de este patrón.

Participantes

Cliente: Representa a todas las entidades externas que interactúan con la aplicación (usuario, servicio, etc.).

Aplicación: Encargada de recibir y dar respuesta a las solicitudes del cliente.

Protocolo de Seguridad: Conjunto de reglas que una solicitud debe de cumplir para ser enviada al componente protegido.

Proxy Confiable: Encargado de verificar que, las solicitudes arribantes, cumplan el "Protocolo de Seguridad" y enviarlas al componente protegido.

Componente Protegido: Servicio de la empresa, aplicaciones externas, componentes de terceros, etc. a quien va dirigida la solicitud.

Comportamiento

Cuando el cliente envía una solicitud que debe ser atendida por un “Componente Protegido”, el “Proxy Confiable”, verifica que la solicitud contenga los parámetros requeridos, que estos posean la estructura adecuada, que hayan sido cumplidas un conjunto de precondiciones necesarias para llevar a cabo la solicitud, etc. En el caso de que todos los requerimientos hayan sido satisfechos, envía la solicitud hacia el “Componente Protegido” o crea una nueva solicitud desde cero, copiando solamente, los datos relevantes Fig. 7 (Anexo 2). De no cumplir con los requisitos, se descarta la solicitud y se reporta el error Fig. 8 (Anexo 2).

Estrategias

Para la implementación de este patrón, en muchos casos, será necesario estudiar el componente en cuestión que se desea proteger y, a partir de sus especificaciones, desarrollar el proxy que se empleará para brindar un acceso protegido al mismo. En otros casos, será conveniente emplear mecanismos ya definidos y bien probados como los “Procedimientos almacenados” o la sentencia “PREPARE”, empleados para el acceso a las bases de datos [38].

Otra vía de implementación de este patrón, facilita la protección durante el acceso a los recursos, a la vez que brinda a los usuarios, una manera sencilla de acceder, solamente, a los recursos que les están permitidos. Un ejemplo de este tipo de implementación, aparece en los “Mapas de referencias de acceso” [45]. Es importante destacar que, la implementación mencionada, puede hacer una contribución a la seguridad similar a la del método de validación: “Aceptar bueno conocido” [34], pero con un costo mucho menor de rendimiento.

2.3.4 Patrón: Canal seguro

Intención

Proteger la confidencialidad y la integridad de los datos en tránsito, entre dos áreas seguras del sistema, separadas por zonas en las que no existe garantía de la seguridad de los datos [32].

Motivación

Cuando la información transita por canales inseguros en texto claro, es susceptible a ser capturada, analizada o modificada por usuarios no autorizados. Esto, pone en riesgo los atributos de confidencialidad e integridad de la información.

Dichos atributos, deben ser preservados para la información que almacena o procesa la aplicación pero, también, deben ser preservados para aquella información requerida para su funcionamiento como: datos de estado, identificadores de sesión, etc. Si estos elementos son expuestos, pueden facilitar a un atacante la realización de un conjunto de ataques de: robo de identidad, secuestro de sesiones, escalado de privilegios, entre otros.

Con el empleo de comunicaciones seguras (comunicaciones con el empleo de criptografía), es posible proteger la información de las miradas ajenas. Además, es posible detectar cuando ha sido comprometida la integridad de dicha información.

Este tipo de comunicaciones, también, puede brindar la posibilidad de autenticar, tanto al cliente como al servidor, dificultando la realización de varios ataques de suplantación.

Estructura

En la Fig. 9 (Anexo 2), aparece la estructura estática de este patrón.

Participantes

Cliente: Realiza una solicitud de intercambio de información sensible a la aplicación.

Aplicación: Se encarga de crear el canal seguro para enviar la información

al cliente.

Canal Seguro: Es el mecanismo encargado de proteger los datos en tránsito.

Comportamiento

Cuando el cliente realiza una solicitud de acceso a la aplicación, esta realiza una solicitud al sistema para crear un canal seguro. El sistema, durante el proceso de creación del canal, negocia con el cliente las características del mismo y la llave a emplear. Después de establecido el canal, todas las solicitudes desde el cliente se procesarán por el canal seguro (proceso de cifrado) y serán enviadas al servidor. Un proceso similar ocurrirá con las respuestas del servidor. El canal se mantendrá activo hasta que el usuario solicite salir de la aplicación Fig. 10 (Anexo 2).

Estrategias

Como el empleo de la criptografía puede afectar el rendimiento del servidor, algunos autores sugieren reservar su empleo, solamente para aquellas transacciones que contengan información confidencial (ej. una aplicación que establece un canal seguro para autenticar al usuario y después cierra el canal) [28].

Sin embargo, para aquellas aplicaciones que requieren intercambiar con el cliente datos de estado e identificadores de sesiones, será necesario mantener el canal seguro durante el transcurso de toda la sesión. Esto se debe, como ya se ha planteado, a que dichos identificadores deben ser considerados confidenciales debido a la función que cumplen en el sistema.

Muchos servidores Web, ya ofertan la posibilidad de crear canales seguros (ej. empleo de SSL/TLS en Apache). Esta posibilidad, puede ser empleada para reducir la complejidad del sistema, evitando tener que implementar la lógica que requiere este patrón. Además, excepto casos en los que haya un amplio conocimiento de criptografía, es mejor emplear algoritmos y aplicaciones

comerciales que hayan sido probadas extensamente, a la creación de un algoritmo propio de cifrado. Súmese a esto, que muchas de estas implementaciones realizan un empleo más óptimo del hardware y, para los casos en que exista hardware de cifrado, pueden emplearlo para reducir las afectaciones al rendimiento del sistema.

2.3.5 Patrón: Identificador de autenticidad de transacciones

Intención

Brindar un mecanismo que permita determinar si, las transacciones o solicitudes que llegan al servidor, son genuinas o falsificadas [45].

Motivación

Como consecuencia del funcionamiento de las aplicaciones Web y los navegadores de los usuarios, un atacante puede generar transacciones falsas y persuadir al usuario para que las envíe (sin su conocimiento) a la aplicación como si se tratasen de transacciones genuinas. Estas transacciones, al ser procesadas en el servidor, gozarán de los mismos privilegios y les estarán permitidas todas aquellas operaciones a las que el usuario, según sus privilegios, tenga acceso.

El empleo de un identificador de autenticidad de las transacciones, permite obtener un grado mayor de confianza de que, las transacciones que llegan al sistema, fueron generadas por el usuario (son autenticas) y no son falsificaciones generadas por un atacante.

Estructura

En la Fig. 11 (Anexo 1), aparece la estructura estática de este patrón.

Participantes

Cliente: Representa a todas las entidades externas que interactúan con la aplicación (usuario, servicio, etc.).

Aplicación: Encargada de recibir y dar respuesta a las solicitudes del cliente.

Objetivo: Módulo o componente, encargado de llevar a cabo la transacción.

Forma: Componente que facilita el intercambio de información, entre el cliente y la aplicación.

Identificador: Número aleatorio, generado en el momento de crear una "Forma" para intercambiar información con el cliente.

Administrador de Identificadores: Componente que se encarga de crear los identificadores y mantener una colección de ellos para verificar la autenticidad de las transacciones que llegan al sistema.

Comportamiento

Cuando el cliente ejecuta la aplicación, al generarse la interfase de usuario, se generan "Identificadores" que se introducen, cual si fueran marcas, en las "Formas" antes de ser enviadas al "Cliente" Fig. 12 (Anexo 2).

Cuando el "Cliente" llene la "Forma" y envíe la solicitud, la aplicación verificará si, la "Forma" recibida, contiene un "Identificador" válido antes de permitir que se ejecute la transacción Fig. 13 (Anexo 2). De no ser así, descartará la transacción Fig. 14 (Anexo 2).

Estrategias

Si bien este patrón puede emplearse para validar la autenticidad de cualquier operación que el cliente solicite a la aplicación, es posible, para reducir la complejidad y las afectaciones al rendimiento, limitar su aplicación solamente a aquellas transacciones que impliquen la modificación de la información que contiene la aplicación.

Es importante, si se decide emplear este patrón de forma limitada, tener en cuenta que las solicitudes de información que se realicen al sistema, aunque no representan un peligro para la integridad de los datos, pueden poner en riesgo la confidencialidad de los mismos.

2.4 Otros patrones de diseño seguro

La incorporación de los patrones anteriormente mencionados al diseño de una aplicación, introduce cinco mecanismos que pueden hacer una importante contribución a la seguridad del sistema. No obstante, algunas de las estrategias de implementación de dichos patrones, pueden afectar su eficacia.

Por tanto, a continuación se analizarán tres patrones de diseño seguro que pueden contribuir a forzar el empleo de las estrategias de implementación más eficaces y, adicionalmente, contribuir a la reducción de las posibilidades de los atacantes de explotar otras vulnerabilidades.

2.4.1 Patrón: Punto único de acceso

Intención

Definir una interface común y única para la comunicación de las entidades externas con el sistema y ejercer un mejor control y monitoreo de dichas comunicaciones [32].

Motivación

Contar con varios puntos de acceso a un sistema, implica implementar mecanismos para hacer cumplir las políticas de seguridad en cada uno de ellos. Como consecuencia, en ocasiones será posible encontrar varias implementaciones diferentes de un mismo mecanismo (algunas de estas con problemas de seguridad). En otros casos, podrán encontrarse diferencias e inconsistencias en las políticas de seguridad que dichos mecanismos pretenden hacer cumplir. Esto provoca que, muchos sistemas, no puedan ser defendidos contra los ataques externos de forma efectiva.

En el caso específico de las aplicaciones Web, si no se ha implementado ningún mecanismo para evitarlo, es posible, por lo general, acceder al sistema o a

algunas de sus funcionalidades solicitándolas directamente desde la barra de navegación del navegador del usuario. Esto implica, repetir los mismos mecanismos de seguridad para cada una de las páginas que componen la aplicación y exponerse a los riesgos antes mencionados.

Con el empleo de este patrón, se fuerzan todos los accesos a la aplicación a través de un canal único, permitiendo centralizar la implementación de mecanismos de monitoreo y control, haciéndolos más efectivos y desacoplándolos de la lógica de la aplicación. Previene, además, que entidades externas del sistema se comuniquen directamente con los componentes del mismo [30].

También, contribuye a reforzar la eficacia de los patrones 2.3.1 y 2.3.2 y a reducir la cantidad de posibles ataques a vulnerabilidades como: CWE-665, CWE-285, CWE-259.

Estructura

En la Fig. 15 (Anexo 2), aparece la estructura estática de este patrón.

Participantes

Cliente: Representa a todas las entidades externas que interactúan con la aplicación (usuario, servicio, etc.).

Frontera del sistema: Límite que separa, al sistema protegido, de todas las entidades externas.

Sistema protegido: Conjunto de entidades internas, protegidas por la frontera del sistema.

Punto único de acceso: Provee una interfase para que las entidades externas puedan comunicarse con los componentes internos del sistema.

Comportamiento

Ninguna comunicación puede ser hecha de forma directa, desde un componente externo, hacia algún componente interno del sistema Fig. 16 (Anexo 2). Las entidades externas que lo deseen, deberán contactar con el punto único de

acceso y entregarle su solicitud para que este las haga llegar a las entidades internas Fig. 17 (Anexo 2).

Estrategias

En el caso de las aplicaciones Web, además de concebir en el diseño un punto único de entrada al sistema, debe restringirse el acceso directo a las páginas que componen el mismo.

Una variante, puede ser la de emplear los mecanismos de control de acceso del servidor Web y restringir el acceso al sistema desde cualquiera de sus páginas, excepto, para la que esté concebida como punto de entrada al sistema. Es la más sencilla de todas, pero es susceptible a errores de configuración en el servidor, principalmente, durante la instalación o actualización de la aplicación.

La otra posibilidad consiste en, como primera acción para cada página que contenga la aplicación, comprobar que el acceso a la misma se está produciendo desde una entidad interna. En ese caso, el acceso a la página será permitido, en caso contrario, será rechazado. Esta aproximación, puede tener como inconveniente que no se introduzcan los mecanismos de control en algunas páginas por olvido o que, la lógica implementada, no sea todo lo restrictiva que se desea.

2.4.2 Patrón: Punto de chequeo

Intención

El objetivo de este patrón es comprobar cada una de las solicitudes que llegan al sistema para determinar, dependiendo de si cumplen o no con las políticas de seguridad de la aplicación, si se les permite el acceso al mismo [30].

Motivación

Que una aplicación posea definidas políticas de seguridad y reglas de autorización, no es suficiente para evitar violaciones y accesos no autorizados. Debe existir un mecanismo encargado de hacerlas cumplir y de tomar medidas en casos de violaciones.

Para impedir accesos no autorizados, es vital comprobar, cada vez que se intente acceder a una operación o recurso, quien está intentando acceder y de qué forma pretende hacerlo [28].

En los casos en los que se detecten intentos de violaciones, tener la capacidad de discernir entre un error del usuario y la realización de un ataque, puede facilitar el empleo de medidas preventivas.

Estructura

En la Fig. 18 (Anexo 2), aparece la estructura estática de este patrón.

Participantes

Cliente: Representa a todas las entidades externas que interactúan con la aplicación (usuario, servicio, etc.).

Aplicación: Encargada de recibir y dar respuesta a las solicitudes del cliente.

Objetivo: Componente interno de la aplicación hacia donde va dirigida la solicitud del cliente.

Políticas de seguridad: Conjunto de reglas que, permiten determinar, cuándo una condición de acceso es permitida o no.

Contramedidas: Provee un grupo de acciones a ejecutar como respuesta ante una violación de las políticas de seguridad.

Punto de chequeo: Implementa un método que permite validar si, las solicitudes que llegan al sistema, cumplen las políticas de seguridad. Dispara acciones, en caso necesario, para proteger al sistema de los atacantes.

Comportamiento

Cuando se ha implementado un punto de chequeo, todas las solicitudes de

las diferentes entidades externas deben pasan a través de él. Al llegar una solicitud, el punto de chequeo, verifica si esta cumple con las políticas de seguridad establecidas.

Cuando se detectan violaciones, además de descartar la solicitud entrante, el punto de chequeo puede tomar un conjunto de medidas como: cerrar la sesión, bloquear el acceso por espacio de algunos minutos, bloquearlo de forma permanente, enviar un mensaje al administrador, etc. Fig. 19 (Anexo 2). En caso de que no se detecten violaciones, se permitirá el tránsito normal de la solicitud hacia su destino Fig. 20 (Anexo 2).

Estrategias

El patrón, punto de chequeo, puede ser empleado para verificar el apego a las políticas de seguridad en todos los puntos, que se estime conveniente, dentro de la aplicación. Sin embargo, se corre el riesgo de que algunas operaciones importantes no sean revisadas por desconocimiento u olvido.

Quizás, la forma más eficaz de emplear el punto de chequeo, es colocándolo exactamente después del punto único de acceso a la aplicación e incorporándole todas las verificaciones relevantes para la seguridad (patrón "validador interceptor", patrón "escape de las salidas", etc.). De esta forma, se podrán verificar todas las solicitudes que lleguen al sistema y se reducirán al mínimo las posibilidades de que pueda ser evadido.

Aunque complejiza un poco la implementación, sería recomendable emplear este patrón, para verificar el apego a las políticas de seguridad de las comunicaciones e intercambios de información que ocurren entre los componentes internos.

2.4.3 Patrón: Sesión dirigida

Intención

Evitar que el usuario pueda hacer saltos arbitrarios dentro de la aplicación y, de esta forma, evadir mecanismos de validación y mecanismos de control de acceso [28].

Motivación

Muchas aplicaciones, necesitan capturar datos de los usuarios y, para esto, implementan un conjunto de páginas que guían al usuario durante este proceso. Los atacantes, pueden intentar saltar a páginas arbitrarias para intentar evadir comprobaciones de integridad, validaciones de datos u otros mecanismos de seguridad.

Verificar que hayan sido satisfechas determinadas precondiciones antes de procesar una solicitud, puede contribuir a evitar, que un atacante, evada determinados mecanismos de seguridad o explote vulnerabilidades que, tienen su origen, en la correcta inicialización de alguna variable.

Al igual que el patrón “Punto único de acceso”, puede contribuir a reforzar la eficacia de los patrones 2.3.1 y 2.3.2 y a reducir los posibles ataques a vulnerabilidades como: CWE-665, CWE-285, CWE-259.

Estructura

En la Fig. 21 (Anexo 2), aparece la estructura estática de este patrón.

Participantes

Cliente: Representa a todas las entidades externas que interactúan con la aplicación (usuario, servicio, etc.).

Aplicación: Encargada de recibir y dar respuesta a las solicitudes del cliente.

Objetivo: Módulo o componente encargado de llevar a cabo la transacción.

Precondiciones: Eventos que deben de haber ocurrido satisfactoriamente, antes de poder llevar a cabo determinada acción.

Sesión Dirigida: Encargada de validar, a partir de las reglas y

precondiciones, si una operación puede o no realizarse.

Comportamiento

Para cada solicitud que realice el cliente, se verificará que esta ha cumplido con determinadas precondiciones. En el caso, en el que todas las precondiciones necesarias hayan sido satisfechas, se enviará la transacción a su destino Fig. 22 (Anexo 2). En caso de que otras operaciones deban ser realizadas, se descartará la solicitud y se generará un error Fig. 23 (Anexo 2).

Estrategias

En los casos, en los que todas las páginas de la aplicación puedan ser accedidas de forma directa y una transacción requiera del empleo de varias páginas para completarse, el usuario no podrá solicitar la segunda página hasta que hayan culminado exitosamente los procesos requeridos en la primera. Datos de sesión almacenados en el servidor, serán empleados para determinar que página debe de ser mostrada en cada momento. En caso de que no existan los datos, será mostrada la página principal.

En los casos donde se aplica el patrón "Punto Único de Acceso", aunque se reduce la posibilidad de que un usuario malintencionado pueda llamar a páginas arbitrarias, será todavía posible invocar a funcionalidades arbitrarias de la aplicación, por tanto, aunque disminuye la utilidad de este patrón, aún puede ser preciso emplearlo.

2.5 Efecto esperado del empleo de los patrones para la seguridad del sistema

En la Tabla 2 aparece recogido el efecto esperado, con la incorporación de los patrones de diseño seguro, en la eliminación de determinadas vulnerabilidades y la reducción de las posibilidades de un atacante para explotar aquellas, que no hayan podido ser eliminadas.

Al igual que en la Tabla 1, las filas representan las vulnerabilidades relacionadas en el Anexo 1 y las columnas los patrones:

- P1: Validador interceptor
- P2: Escape de las salidas
- P3: Proxy confiable
- P4: Canal seguro
- P5: Identificador de autenticidad de transacciones
- P6: Punto único de acceso
- P7: Punto de chequeo
- P8: Sesión dirigida

En la intercepción, se representa con una letra (D), aquellos patrones que influyen directamente en la eliminación de determinada vulnerabilidad y con una (I) aquellos patrones que, aunque no eliminan la vulnerabilidad, reducen las posibilidades de un atacante para explotarla.

Vuln	P1	P2	P3	P4	P5	P6	P7	P8	Vuln	P1	P2	P3	P4	P5	P6	P7	P8
CWE-20	D					I	I	I	CWE-94	D	D	D	I	I	I	I	I
CWE-116		D				I	I	I	CWE-494				I				
CWE-89	D	D	D	I	I	I	I	I	CWE-404								I
CWE-79	D	D		I		I	I		CWE-665								I
CWE-78	D	D	D	I	I	I	I	I	CWE-682	I			I		I	I	I
CWE-319				D					CWE-285						I	I	I
CWE-352		I		I	D				CWE-327								
CWE-362									CWE-259	I		I		I	I	I	I
CWE-209				I					CWE-732	I					I	I	I
CWE-119	I					I	I	I	CWE-330								
CWE-642				I					CWE-250								
CWE-73	D		D	I	I	I	I	I	CWE-602	D	D			D			
CWE-426	I							I									

Tabla 2: Efecto en la reducción de las vulnerabilidades esperado de los patrones propuestos.

Como puede apreciarse, si se aplican adecuadamente estos ocho patrones al diseño de la aplicación, de las 25 vulnerabilidades propuestas, será posible eliminar 10, reducir las posibilidades de atacar otras 11 y no se ofrecerá ningún tipo de protección para 4.

2.6 Conclusiones

Durante este capítulo, se han propuesto un grupo de cinco patrones que fueron seleccionados a partir de un conjunto de mecanismos para prevenir la aparición de algunas vulnerabilidades. Fueron propuestos, además, tres patrones que pueden contribuir a la eficacia de la implementación de los primeros cinco propuestos y realizar un aporte adicional en la reducción de las posibilidades de ataque de otras vulnerabilidades.

Por tanto, se espera que con una adecuada incorporación al diseño de una aplicación Web de estos ocho patrones, se eliminen un conjunto de vulnerabilidades y se incremente la resistencia a ataques de dicha aplicación. Lográndose con ello, el objetivo de hacerla más segura con la realización de un esfuerzo menor por parte de los equipos de desarrollo y sin que se requieran grandes conocimientos y experiencia en seguridad.

Capítulo 3: Beneficios del empleo de los patrones de diseño seguro

3.1 Introducción

En el presente capítulo, se llevará a cabo una comparación entre el esfuerzo requerido para incorporar los patrones al diseño de una aplicación en desarrollo y el esfuerzo requerido para realizar de un análisis de riesgos.

Además, se mencionarán de forma breve los resultados obtenidos en la realización de una prueba experimental que permitió validar el aporte para la seguridad de una aplicación Web, de los patrones propuestos en este trabajo.

3.2 Implicaciones de llevar a cabo un análisis de riesgo

Como se mencionó anteriormente en el Capítulo 1, algunas metodologías como “CLASP” y “SDL” recomiendan llevar a cabo, concluido el diseño de la aplicación y antes de comenzar la etapa de implementación, un “análisis de riesgos” con el objetivo de identificar las vulnerabilidades presentes en el diseño de la aplicación.

Los pasos para llevar a cabo este tipo de análisis, son los siguientes [19]:

1. Crear un modelo, con la menor granularidad posible, de la aplicación y sus partes componentes.
2. Crear, a partir de este modelo, un DFD (Diagrama de Flujo de Datos) para representar los intercambios de información entre cada una de sus partes componentes.
3. Determinar, para cada flujo de datos, cuáles de las amenazas existentes pueden materializarse.
4. Para cada amenaza determinar, que vulnerabilidad en el diseño del sistema, hace posible su materialización.
5. Clasificar los riesgos de que cada amenaza pueda llegar a materializarse atendiendo a: la dificultad que entraña detectar la vulnerabilidad, los conocimientos requeridos para detectarla y el

daño potencial que puede traer para los usuarios y para el negocio al ser explotada.

6. Determinar, para aquellas vulnerabilidades que posean un riesgo de ser explotadas más elevado, qué mecanismos de mitigación pueden emplearse.

Conocidos los pasos para llevar a cabo un análisis de riesgo, veamos las implicaciones de efectuar algunos de los puntos que mayor esfuerzo, experiencia y conocimientos requieren.

Durante la realización del punto 3, para determinar a qué amenazas se encuentra sujeto determinado flujo de datos, será necesario valorar cómo reaccionará cada componente ante la presencia de un ataque. El personal que realice la evaluación, deberá valorar la mayor cantidad de ataques posibles apoyándose para esto, en el empleo de los “arboles de ataque” (algunos pueden llegar a tener hasta 287 tipos de ataques diferentes [46]). Por cada variante de ataque que sea factible realizar, se deberá determinar a qué amenazas deberemos enfrentarnos. Por consiguiente, mientras más variantes se evalúen, mayor será la posibilidad de detectar posibles amenazas.

Prestando atención ahora al punto 5, en el que será necesario clasificar los riesgos atendiendo a cuán fácil será descubrir y explotar determinada vulnerabilidad, aparece nuevamente la necesidad de contar con conocimientos y experiencias para que el proceso sea lo más efectivo posible. Esta necesidad de personal preparado se debe, sobre todo, a cuan dependiente de la subjetividad de cada individuo resulta el proceso de evaluación.

Por último, al igual que en los demás puntos analizados, en el punto número 6, durante el cual se deben determinar qué mecanismos contribuirán a mitigar los principales riesgos encontrados, también requiere que se hayan desarrollado determinadas habilidades y competencias por parte del equipo de trabajo. Estos conocimientos, serán necesarios para evitar la improvisación de los mecanismos de defensa. Comenzar a idearlos desde cero, puede generar defensas defectuosas, que no sean tan efectivas como aparentan, o, lo que es

peor, pueden propiciar la introducción de nuevas vulnerabilidades.

3.3 Implicaciones de aplicar los patrones de diseño seguro

Como ya se ha explicado, los patrones, por lo general, se encuentran formando parte de repositorios que contienen a muchos de ellos con descripciones detalladas del contexto donde pueden ser aplicados, los problemas que resuelven, la forma en que lo hacen, entre otros aspectos.

Durante su aplicación, aparecen dos momentos fundamentales:

1. Identificación del contexto donde debe de ser aplicado un patrón
2. Aplicación de los patrones al diseño de la aplicación

Para su aplicación, la principal complejidad surge del hecho de que será necesario identificar los contextos, en el diseño de una aplicación, donde pueden aplicarse (punto 1). Evidentemente, esto implica que el personal encargado de identificar dichos contextos, debe de estar familiarizado con la mayor cantidad posible de patrones para que este proceso sea mucho más rápido y efectivo.

Identificados los contextos, solo resta modificar el diseño de la aplicación para incorporar los patrones correspondientes (punto 2). Para esto, no se requiere de ningún esfuerzo adicional, pues la mayoría de los patrones vienen con especificaciones claras de cómo deben de ser integradas al diseño. Además, en algunos casos, se ofrece el código fuente de su implementación y, en otros, pueden emplearse *frameworks* como: "*OWASP ESAPI Validation API*" [33], que ya contienen la implementación de alguna variante (este *framework* se refiere a los "Validadores Concretos" que aparecen en la estructura del patrón "Validador Interceptor").

3.4 Ventajas del empleo de los patrones de diseño seguro propuestos

Si se analizan los patrones propuestos en el presente trabajo, recordaremos que la propuesta surge del análisis de los mecanismos que se emplean para

mitigar varias de las principales vulnerabilidades que afectaron a las aplicaciones durante el año 2009.

La selección de estás vulnerabilidades, es el resultados de la realización de un análisis de riesgo que involucró a varios cientos de aplicaciones y que permitió determinar que estas eran, generalmente, las más prevalentes y que, a su vez, requerían de un menor esfuerzo y preparación por parte de un atacante para encontrarlas y explotarlas.

Esto implica, comparativamente hablando, que no será necesario realizar los cinco primeros puntos que comprende la realización de un análisis de riesgo. Además, como los patrones de diseño ya ofrecen un mecanismo para la mitigación de las vulnerabilidades, prácticamente se elimina la necesidad de realizar el paso número 6.

Todo lo anteriormente planteado implica que, al aplicar los patrones al diseño de una aplicación, se estarán mitigando varias de las principales vulnerabilidades que afectan a las aplicaciones Web, sin que se haya requerido la experiencia y el esfuerzo que hubieran sido necesarios para realizar un análisis de riesgos.

Pudiera cuestionarse que las vulnerabilidades analizadas, representan solo una parte de los cientos de tipos diferentes que existen, o que, los patrones seleccionados, son incapaces de mitigarlas todas. Sin embargo, algo similar ocurre al efectuar un análisis de riesgos. Estos, como ya se ha expresado, intentan identificar la mayor cantidad de problemas, pero solo resuelven los más importantes. Sería prácticamente imposible resolverlos todos, además de ser económicamente inviable.

3.5 Validación experimental de los beneficios para la seguridad del empleo de los patrones

Para validar el beneficio que el empleo de los patrones de diseño seguro tiene para la seguridad de las aplicaciones Web, se llevó a cabo un experimento para verificar que, al aplicar los patrones propuestos, era posible mitigar varias de las vulnerabilidades mencionadas y hacer la aplicación más resistente a ataques.

Para esto, se seleccionó una aplicación Web en la que fueron introducidas 13 vulnerabilidades (10 de ellas podían eliminarse directamente) y se verificó, llevando a cabo 9 tipos de ataques diferentes (los más comunes para las aplicaciones Web [25, 43]), que era posible explotar estas 13 vulnerabilidades. Cada vulnerabilidad fue atacada empleando uno o varios de los ataques seleccionados, para un total de 48 ataques efectivos realizados.

Posteriormente, esta misma aplicación con las vulnerabilidades introducidas, se modificó para aplicarle los patrones de diseño seguro propuestos en el presente trabajo. Nuevamente, se evaluó el comportamiento de la aplicación frente a los 48 ataques antes mencionados.

Como resultado, de los 48 ataques realizados, solamente 3 fueron efectivos. De las 13 vulnerabilidades introducidas, solamente 2 fueron parcialmente explotadas (no todos los ataques llevados a cabo contra estas 2 vulnerabilidades fueron efectivos).

Teniendo en cuenta estos resultados, puede considerarse que el empleo de los patrones tuvo un impacto significativo en el incremento de la seguridad de la aplicación Web en estudio, eliminó una parte sustancial de las vulnerabilidades presentes en la aplicación y prácticamente eliminó la posibilidad de realizar algún ataque exitoso.

El proceso realizado se describe con más detalles en el artículo “Patrones de diseño seguro para aplicaciones Web” [47].

3.6 Conclusiones

Durante el presente capítulo se comparó, atendiendo al esfuerzo y a la experiencia requeridos, la realización de un análisis de riesgos y el uso de los patrones de diseño seguro para construir una aplicación Web más segura.

En este análisis, se llegó a la conclusión de que, al emplearse dichos patrones, se estaban empleando los resultados de un análisis de riesgos que identificaba las principales y más comunes vulnerabilidades presentes en las aplicaciones Web. Por tanto, se reducía la necesidad de llevar a cabo este proceso y, por consiguiente, el esfuerzo y la experiencia en seguridad que

requeriría un equipo de desarrollo para evitar los problemas de seguridad en una aplicación en construcción.

Además, se presentaron de forma muy breve, los resultados de las pruebas de penetración realizadas a una aplicación Web, a la que le fueron aplicados los patrones de diseño seguro. En esta prueba pudo apreciarse que, de 48 ataques efectuados para explotar 13 vulnerabilidades presentes en la aplicación, tan solo 3 fueron satisfactorios. Esto permite considerar que, los patrones de diseño seguro para las aplicaciones Web presentados en este trabajo, pueden ser empleados satisfactoriamente para desarrollar aplicaciones Web más seguras con la realización de un esfuerzo menor por parte de los equipos de desarrollo.

Conclusiones

Después de haber transitado por los diferentes capítulos del presente trabajo, podemos concluir que: la hipótesis de trabajo que nos planteamos al inicio de esta investigación ha sido satisfecha, pues se obtuvieron ocho patrones de diseño que permiten construir aplicaciones Web más seguras con un esfuerzo menor que otros métodos y sin que se requiera gran experiencia en seguridad por parte de los equipos de desarrollo que los empleen.

Además, todos los objetivos propuestos para la realización de este trabajo se cumplieron satisfactoriamente. Se identificó una definición de lo que podría considerarse un software seguro, se determinaron los principales métodos existente para reducir los problemas de seguridad durante el desarrollo de las aplicaciones, de estos métodos se seleccionó el empleo de los patrones de diseño seguro como el más factible a emplear y se verificó que, realmente, podía contribuir a crear aplicaciones más seguras, realizando un esfuerzo menor y requiriendo menos experiencia de los equipos de desarrollo que otros métodos.

Recomendaciones

Como recomendaciones a este trabajo, se puede sugerir continuar investigando que otros patrones de diseño pueden emplearse para la construcción de aplicaciones más seguras pues, existen todavía, vulnerabilidades que no pueden ser mitigadas por los patrones propuestos. Además, dichos patrones solo aportan mecanismos para construir aplicaciones más resistentes a ataques, pero también serían necesarios otros patrones que contribuyesen a crear aplicaciones tolerantes a fallos y capaces de recuperarse en ante la ocurrencia de errores.

Además, es importante recomendar que se mantenga una estrecha vigilancia de las nuevas vulnerabilidades que aparezcan, con el objetivo de, si los patrones hasta ahora definidos no representan solución, definir nuevos patrones que contribuyan a mitigarlas.

Por último, recomendar la extensión del estudio de los patrones a otras fases dentro del ciclo de vida de la aplicación. Patrones de implementación o de configuración, podrían ser definidos para facilitar la tarea de garantizar la seguridad de las aplicaciones y dedicar mayor cantidad de tiempo a satisfacer las necesidades de los usuarios.

Referencias bibliográficas

1. *Basic Concepts and Taxonomy of Dependable and Secure Computing.* **Avizienis, Algirdas, y otros.** 1, Enero-Marzo de 2004, IEEE Transactions on Dependable and Secure Computing, Vol. 1, págs. 11-33.
2. **Bishop, Matt.** *Computer Security: Art & Science.* s.l. : Addison Wesley, 2002. 0-201-44099-7.
3. *Security In The Software Lifecycle - Making Software Development Processes - and Software Produced by Them - More Secure.* US Department of Homeland Security. 2006.
4. **Ozment, Andy.** Research Statement. [En línea] [Citado el: 17 de 01 de 2009.] http://www.cl.cam.ac.uk/~jo262/research.html.
5. **Howard, Michael y Lipner, Steve.** *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software.* s.l. : Microsoft Press, 2006.
6. **Goertzel, Karen Mercedes, Winograd, Theodore y McKinley, Holly Lynne.** *Software Security Assurance: A state of the art report (SOAR).* Defense Tecnical Information Center. 2007.
7. **Redwine, Samuel T., Baldwin, Rusty O. y Polydys, Mary L.** *Secure Software Assurance: A Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software v0.9.* US Department of Homeland Security. 2006.
8. **Secure Software, Inc.** *CLASP - Comprehensive Lightweight Application Security Process v2.0.* 2006.
9. **Redwine, Samuel T. y Davis, Noopur.** *Processes for Producing Secure Software.* National Cyber Security Partnership. 2004.
10. **McGraw, Gary.** *Software Security: Building Security In.* s.l. : Addison Wesley Professional, 2006.

11. **Rumbaugh, James, Jacobson, Ivar y Booch, Grady.** *The Unified Modeling Language Reference Manual.* s.l. : Addison Wesley, 1998.

12. **Haley, Charles B.** Arguing Security: A Framework for Analyzing Security Requirements. [Doctor of Philosophy in Computer Science Thesis]. 2007.

13. **Hubbard, R.** Design, Implementation, and Evaluation of a Process to Structure the Collection of Software Project Requirements. [PhD Thesis]. 1999. Colorado Technical University.

14. *Building Security Requirements with CLASP.* **Viega, John.** 2005.

15. **Mead, Nancy R.** *How To Compare the Security Quality Requirements Engineering (SQUARE) Method with Other Methods.* SEI, Carnegie Mellon University. 2007.

16. **Mannion, M. y Keepence, B.** *SMART Requirements.* s.l. : ACM SIGSOFT, 1995.

17. **Talukder, Asoke K. y Chaitanya, Manish.** *Architecting secure software systems.* s.l. : CRC Press.

18. **Caralli, Richard A., y otros.** *Introducing OCTAVE Allegro: Improving the Information Security Risk Assessment Process.* SEI, Carnegie Mellon University.

19. **Saitta, Paul, Larcom, Brenda y Eddington, Michael.** Trike v.1 Methodology Document [Draft]. 2005.

20. **Graff, Mark G. y Wyk, Kenneth R. van.** *Secure Coding: Principles & Practices.* s.l. : O'Reilly, 2003.

21. **Chess, Brian y West, Jacob.** *Secure Programming with Static Analysis.* s.l. : Addison-Wesley, 2007.

22. **Gallagher, Tom, Jeffries, Bryan y Landauer, Lawrence.** *Hunting Security Bugs.* s.l. : Microsoft Press, 2006.

23. **OWASP Foundation.** *OWASP Testing Guide v2.0.* 2007.

24. **OWASP Foundation.** *OWASP Application Security Verification Standart 2008 – Web Application Edition.* 2008.

25. **McClure, Stuart, Shah, Saumil y Shah, Shreeraj.** *Web Hacking: Attacks and Defense.* s.l. : Addison & Wesley, 2002. 0-201-76176-9.

26. **Weiss, M.** Modelling Security Patterns Using NFR Analysis. *Integrating security and software engineering: advances and Future Visions.* s.l. : Idea Group Publishing, 2006, 6, págs. 127-141.

27. **Dougherty, Chad, y otros.** *Secure Design Patterns.* SEI, Carnegie Mellon University. 2009.

28. **Schumacher, Markus, y otros.** *Security Patterns: Integrating Security and Systems Engineering.* s.l. : John Wiley & Sons, 2006.

29. **Schumacher, Markus y Roedig, Utz.** Security Engineering with Patterns. 2001. Department of Computer Science, Darmstadt University of Technology.

30. **Wassermann, Ronald y Cheng, Betty H.C.** Security Patterns. 2003. Department of Computer Science and Engineering Michigan State University.

31. **Kienzle, Darrell M., y otros.** Security Patterns Repository - Version 1.0. 2002.

32. **Steel, Christopher, Nagappan, Ramesh y Lai, Ray.** Core Security Patterns: Best Practices and Strategies for J2EE™, Web Services, and Identity Management. s.l. : Prentice Hall, 2005.

33. **OWASP Foundation.** *OWASP ASDR: The Application Security Desk Reference.* 2008.

34. **OWASP Foundation.** *A Guide to Building Secure Web Applications and Web Services.* 2006.

35. **Howard, Michael, LeBlanc, David y Viega, John.** *19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them.* s.l. : McGraw-Hill/Osborne, 2005.

36. **OWASP Foundation.** *OWASP Top 10: The Ten Most Critical Web Application Security Vulnerabilities.* 2007.

37. **MITRE.** CVE: Common Vulnerabilities and Exposures. [En línea] [Citado el: 18 de 11 de 2009.] http://cve.mitre.org/.

38. CWE: Common Weakness Enumeration. A community-developed Dictionary of Software Weakness Types. [En línea] [Citado el: 02 de 05 de 2009.] http://cwe.mitre.org/index.html.

39. 2009 CWE/SANS Top 25 Most Dangerous Programming Errors. [En línea] [Citado el: 02 de 05 de 2009.] http://cwe.mitre.org/top25/index.html.

40. **Netland, Lars-Helge, Espelid, Yngve y Mughal, Kahlid Azim.** Security Pattern for Input Validation. 2006. Department of informatics, University of Bergen.

41. **Grossman, Jeremiah, y otros.** *Cross Site Scripting Attacks: XSS Exploits and Defense.* s.l. : Syngress, 2007.

42. **Microsoft Corporation.** *Improving Web Application Security: Threats and Countermeasures.* s.l. : Microsoft Press, 2003. 0735618429.

43. **Alshanetsk, Ilia.** *PHP architect's Guide to Security.* s.l. : Marco Tabini & Associates, 2005.

44. **Shiflett, Chris.** *Essential PHP Security.* s.l. : O'Reilly, 2005.

45. **Shiflett, Chris.** PHP Security. 2004. O'Reilly Open Source Convention.

46. CAPEC: Common Attack Pattern Enumeration and Classification. A community knowledge resource for building secure software. [En línea] 2009. [Citado el: 10 de 02 de 2010.] http://capec.mitre.org/data/index.html.

47. *Patrones de diseño seguro para aplicaciones Web.* **Quintas, Joaquín.** C. Habana: s.n., 2010, Revista Militar de ciencia y Tecnología.

Anexo 1: Solución a las principales vulnerabilidades propuestas en el listado "*MITRE CWE/SANS Top 25*"

CWE-20: Incorrecta validación de las entradas

Para mitigar o resolver este tipo de vulnerabilidad, será necesario entender las áreas por donde, información insegura, puede entrar a la aplicación. Variables de entorno, ficheros que emplea la aplicación, entradas del usuario, bases de datos, sistemas externos que proveen información, entre otros, constituyen vectores que deberán ser validados teniendo en cuenta su longitud, sintaxis, tipo y reglas de negocio.

Se prefiere emplear el principio de "Aceptar bueno conocido" (ej. "Listas blancas") al de "Descartar malo conocido" (ej. "Listas negras"). Se recomienda además, que los datos a ser validados, transiten solamente una vez por los mecanismos de validación y, para el caso de las aplicaciones Web, que no se confíe en los mecanismo de validación del lado del cliente y se dupliquen o solamente se implementen en el lado del servidor.

CWE-116: Codificación o escape de las salidas defectuoso

Para dar solución a esta vulnerabilidad, se recomienda que, a todos los intercambios de información entre los diferentes componentes de la aplicación, especialmente si provienen o son conformados empleando fuentes de datos externas, se les aplique una correcta codificación y que sean escapadas adecuadamente la información de control que viene asociada a los datos.

CWE-89: Fallo para preservar la estructura de una consulta SQL (Inyección SQL)

Este tipo de vulnerabilidad, es posible resolverla a partir de la realización de una adecuada codificación y escape de la información que será enviada al componente de base de datos. Contribuyen, además, una correcta validación de los datos de entrada y el empleo de procedimientos almacenados y consultas preparadas (aquellas que se crean con la sentencia PREPARE QUERY).

CWE-79: Fallo para preservar la estructura de una página Web (*Cross-Site Scripting*)

Contribuyen a reducir o a solucionar este problema, la realización de una correcta validación de los datos de entrada a la aplicación, una adecuada codificación de la información y el escape de los caracteres de control. Además, es importante prevenir la modificación de la información en tránsito entre el servidor y la aplicación cliente.

CWE-78: Fallo para preservar la estructura de un comando del Sistema Operativo (Inyección de comandos del SO)

Para evitar este tipo de vulnerabilidad, al igual que en otros casos, se recomienda realizar una adecuada validación de las entradas y una correcta codificación y escape de las salidas dirigidas al intérprete de comandos del sistema operativo. Adicionalmente, deben emplearse jaulas (*jails*) y cajas de arena (*sandboxes*) para reducir las posibilidades de interacción del comando con el sistema.

También, es aconsejable emplear mecanismos que, como los procedimientos almacenados y las consultas preparadas, limiten al mínimo indispensable las opciones con las que puedan ser invocados dichos comandos.

CWE-319: Transmisión en texto claro de información sensible

Para resolver esta vulnerabilidad, la principal recomendación es el empleo de algoritmos o sistemas de cifrado que, aplicados a la información que deberá transitar por caminos inseguros, garanticen la confidencialidad de la misma.

El empleo de estos algoritmos o sistemas, puede contribuir a validar, tanto a los clientes como al servidor, y a garantizar la integridad de los datos en tránsito.

CWE-352: Cross-Site Request Forgery (CSRF)

Este es un problema bastante común y complejo de resolver. Una de las posibles soluciones, puede ser el empleo de firmas digitales para garantizar la

autenticidad e integridad de las solicitudes que llegan al sistema. La otra, es la generación en el servidor de un identificador, criptográficamente seguro, que permita verificar la autenticidad de las solicitudes cuando estas regresan del navegador del usuario.

Se hace hincapié en que, para ninguno de los dos casos, la aplicación puede ser vulnerable a XSS (CWE-79) y, en el caso del segundo, la información debe de ser protegida en tránsito para evitar que un atacante pueda capturar el identificador y emplearlo para generar una solicitud genuina.

CWE-362: Condiciones de carrera

Aunque todavía no existe una solución cien por ciento eficaz para resolver este problema, se recomienda emplear adecuadamente las primitivas de sincronización entre procesos que proveen los lenguajes de programación, las capacidades *threat-safe* como la abstracción de acceso a datos y reducir al mínimo los recursos compartidos con el objetivo de eliminar la mayor cantidad de complejidad posible del control de flujo y reducir, de esta forma, la posibilidad de aparezcan o se den condiciones inesperadas.

CWE-209: Fuga de información de mensajes de error

La solución para este tipo de vulnerabilidad, se encuentra en hallar un punto intermedio entre los mensajes de error crípticos y mensajes de error demasiado habladores. Cuando la aplicación se encuentra en producción, todos aquellos mensajes de *debugging* que brindan información sobre la arquitectura interna y el estado de la aplicación, deben de ser eliminados. Solamente se le debe brindar al usuario la información mínima necesaria para que comprenda que está haciendo mal y pueda retroalimentarse y corregirlo.

CWE-119: Fallo para restringir las operaciones a las fronteras de un buffer de memoria (Desbordamiento de *buffer*)

Para reducir las posibilidades de explotar este tipo de vulnerabilidad, no solo será necesario evitar el empleo de funciones con efectos similares a los de la

función “*strcpy*” en C y C++, sino que, además, será necesario realizar una adecuada validación de los parámetros de entrada y del espacio reservado en el *buffer* al que van destinados los datos, pues, aunque lenguajes como: PHP, JAVA, PERL y otros, no son susceptibles a este tipo de vulnerabilidades, los comandos del sistema y la implementación de los mismos lenguajes mencionados, si pueden ser susceptibles a este tipo de vulnerabilidad.

CWE-642: Control externo de datos críticos de estado

Para evitar esta vulnerabilidad, lo más aconsejable es guardar los datos de estado de los clientes en el servidor y, solo en caso de que fuese en extremo necesario, guardarlos en el cliente con el empleo mecanismos de encriptación y chequeos de integridad.

CWE-73: Control externo de nombres de ficheros y caminos

Para evitar esta vulnerabilidad, debe evitarse emplear datos externos a la aplicación en la generación de nombres de ficheros o caminos. Para los casos en que el número de ficheros sea limitado, es posible emplear un “mapa de referencias de acceso” o emplear un mecanismo de validación que siga el principio de “aceptar bueno conocido”.

También, puede resultar de mucha utilidad emplear funciones de canonicalización que permitan determinar, a qué fichero exactamente se intentará acceder y, aunque solo limiten el daño que se ocasionará al sistema operativo, el empleo de jaulas o cajas de arena, pueden contribuir como mecanismo de defensa en profundidad.

CWE-426: Caminos de búsqueda inseguros

Este tipo de vulnerabilidad es posible evitarla si, al acceder a ficheros de la aplicación o al ejecutar comandos del sistema, se emplea el camino completo del recurso o caminos de búsqueda estáticos definidos en la aplicación. Es posible también, validar adecuadamente los caminos de búsqueda antes de emplearlos.

CWE-94: Fallo para controlar la generación de código (Inyección de código)

La forma principal de evitar esta vulnerabilidad, es evitar la generación de código dinámico. En los casos, en los que sea extremadamente necesario llevar a cabo este tipo de implementación, el empleo de un adecuado mecanismo de validación y de escape de la información de control que acompaña a los datos, puede reducir de forma considerable la posibilidad de que surja una vulnerabilidad.

También, es viable el empleo de mecanismos que, al igual que los procedimientos almacenados para las bases de datos, actúen como intermediarios y limiten las posibilidades a la hora de generar código.

CWE-494: Descarga de código sin chequeos de integridad

Aunque no representan soluciones completas, es posible reducir las posibilidades de un atacante, con el empleo de algún mecanismo que permita garantizar la integridad de los datos y medidas para detectar ataques de *DNS-spoofing*. Una solución más efectiva, aunque, en ocasiones más compleja, puede ser la de emplear las infraestructuras de claves públicas y la firma digital de los contenidos descargados.

CWE-404: Liberación de recursos inapropiada

Con la realización de una correcta implementación, que libere adecuadamente los recursos reservados, es posible reducir o eliminar las posibilidades de que este tipo de vulnerabilidad pueda ser explotada. Otra posibilidad, es el empleo de lenguajes que incorporen un recolector de basura.

CWE-665: Inicialización inapropiada

Para evitar este tipo de vulnerabilidad, es posible emplear lenguajes que automáticamente asignen un valor por defecto a las variables que no han sido inicializadas o generen errores en tiempo de compilación al encontrar este tipo de variables. Si esto no es posible, la inicialización explicita de variables, tanto en el momento de su declaración como en el momento de su primer empleo, es la solución más recomendable.

CWE-682: Cálculos incorrectos

Para reducir las posibilidades de aparición de este tipo de vulnerabilidad, se debe de ser cuidadoso a la hora de llevar a cabo la implementación y, prestar especial atención, a las discrepancias entre el tamaño en *bytes* de las variables que se emplean, la precisión, las conversiones entre tipos y otros factores. Puede contribuir también, la utilización de mecanismos de validación que verifiquen que, los valores procedentes de fuentes externas a la aplicación, se encuentran dentro del rango esperado.

CWE-285: Control de acceso incorrecto (Autorización)

La aparición de este tipo de vulnerabilidad, puede reducirse mediante una correcta división de la aplicación en áreas normales, privilegiadas y de administración y, con el empleo de un control de acceso basado en roles (*RBAC* por sus siglas en inglés), asignar los privilegios mínimos requeridos para cada usuario. En el caso de las aplicaciones Web, el control de acceso debe de realizarse del lado del servidor, en cada una de las páginas que componen la aplicación para que los usuarios no puedan acceder a la información o las funcionalidades mediante la solicitud directa de las páginas.

CWE-327: Empleo de algoritmos criptográficos riesgosos o defectuosos

Para reducir las posibilidades de aparición de este tipo de vulnerabilidades, a la hora de emplear algoritmos criptográficos, calcular resúmenes o generar números aleatorios, se recomienda no inventar este tipo de algoritmos y emplear aquellos, bien conocidos y probados, que existen en el mundo. Será necesario, además, verificar periódicamente que no se están empleando algoritmos obsoletos.

CWE-259: Contraseñas en el código

Aunque es un problema complejo de resolver, debe de evitarse por todos los medios posibles dejar contraseñas en el código. Para esto, es posible emplear

ficheros o bases de datos fuertemente protegidos y cifrados para almacenar las contraseñas

CWE-732: Asignación de permisos inseguros a recursos críticos

Para reducir las posibilidades de aparición de este tipo de vulnerabilidad, lo más recomendable es verificar los permisos de recursos críticos para la aplicación como en el caso de los ficheros de configuración y, si tienen permisos inseguros (como acceso a escritura para un usuario normal del sistema), generar un error y salir de la aplicación. También, al igual que en el caso de los problemas con el control de acceso, la aplicación debe de ser particionada según los diferentes privilegios necesarios y asignar adecuadamente dichos privilegios a cada usuario o rol del sistema. Mientras mayor sea el grado de granularidad con que se definan los privilegios de acceso a cada recurso, menor será la probabilidad de que algún usuario tenga más privilegios de los necesarios para cumplir con el rol que la ha sido asignado en la aplicación.

CWE-330: Empleo de valores aleatorios con poca entropía

Para evitar esta vulnerabilidad, será necesario emplear algoritmos adecuados de generación de números pseudo-aleatorios, con una semilla grande (256 bits o más), generada a partir de números pseudo-aleatorios de alta calidad obtenidos de los dispositivos de hardware.

CWE-250: Ejecución con excesivos privilegios

Para reducir la posibilidad de aparición de este tipo de vulnerabilidad, es recomendable identificar las funcionalidades que requieren privilegios adicionales y aislarlas, tanto como sea posible, del resto del código. Elevar los privilegios, solo cuando sea necesario emplear estas funcionalidades, y bajarlos, tan pronto como sea posible. También, es aconsejable realizar una extensa validación de los datos de entrada de las funciones privilegiadas.

CWE-602: Forzado de la seguridad del servidor en el lado del cliente

Cualquier mecanismo de chequeo en el lado del cliente puede ser evadido con relativa facilidad, por tanto, para evitar este tipo de vulnerabilidad, se aconseja realizar todo tipo de validaciones y chequeos del lado del servidor.

Anexo 2: Estructuras estáticas y dinámicas de los patrones propuestos

Validador Interceptor

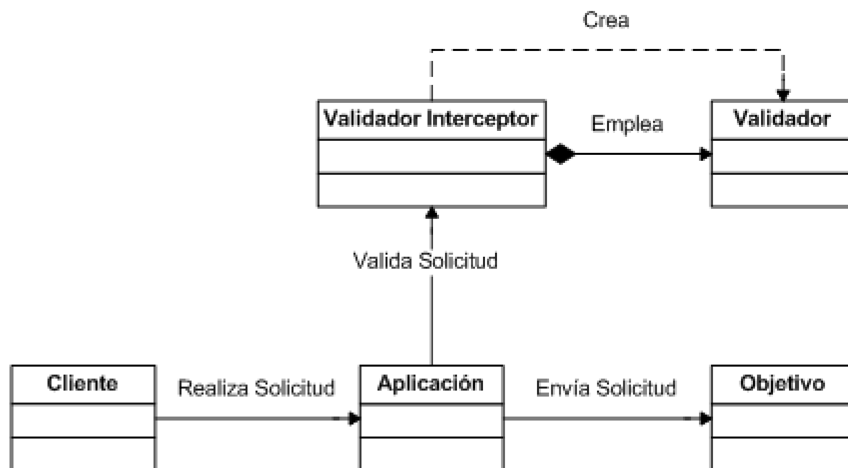

Figura 1: Estructura estática del patrón "Validador Interceptor".

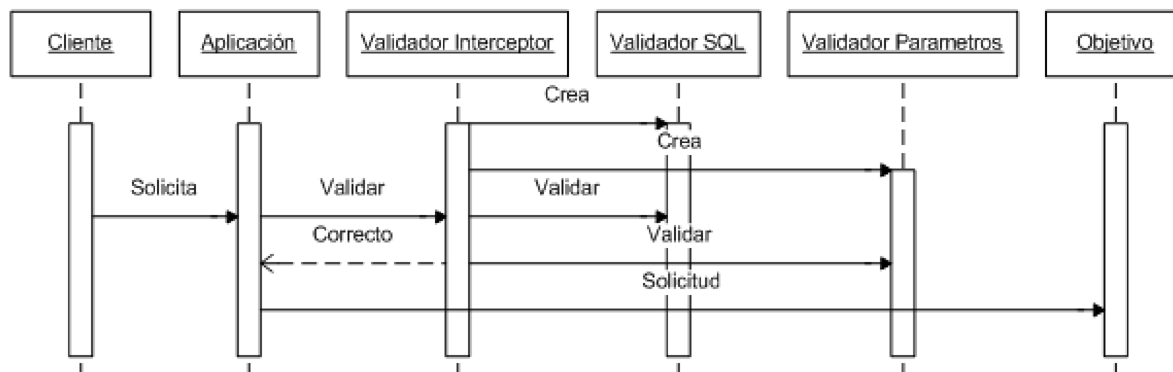

Figura 2: Tránsito normal de una solicitud cuyos parámetros están formados acorde a las reglas del negocio o de tipos de datos.

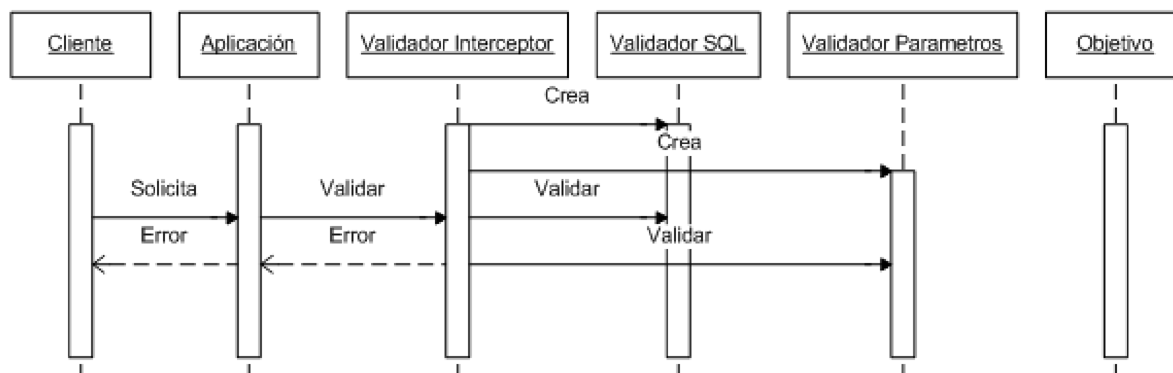

Figura 3: Rechazo de una solicitud que no cumple con las reglas del negocio o de tipo de datos.

Escape de las salidas

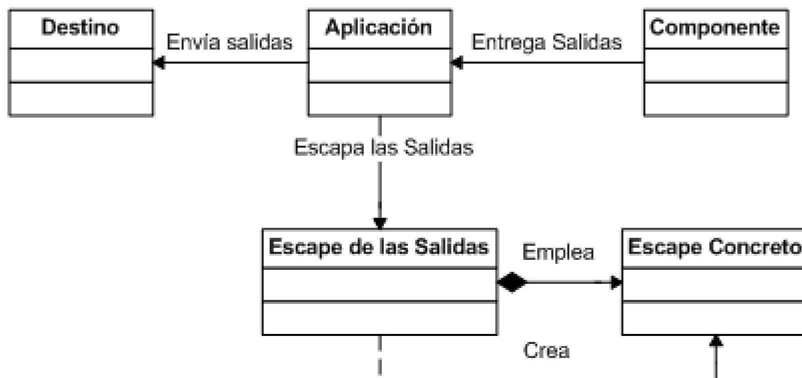

Figura 4: Estructura estática del patrón escape de las salidas.

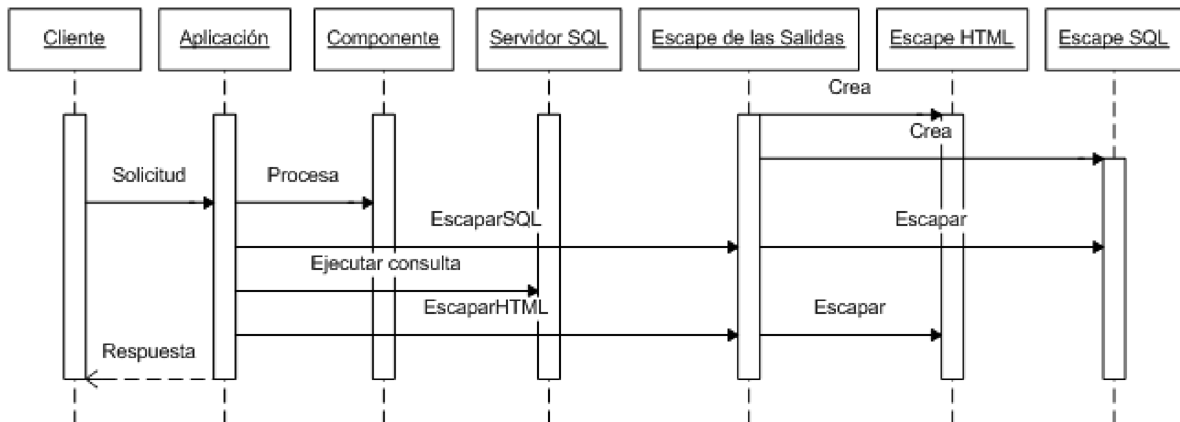

Figura 5: Escape de las salidas de los componentes internos de la aplicación antes de ser enviados a los componentes externos.

Proxy Confiable

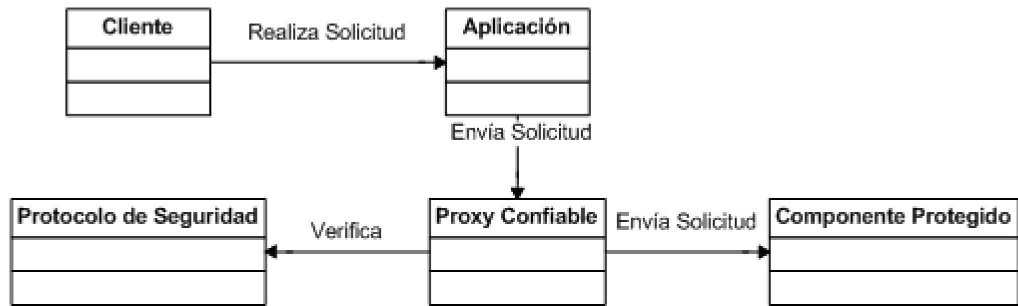

Figura 6: Estructura estática del patrón "Proxy Confiable".

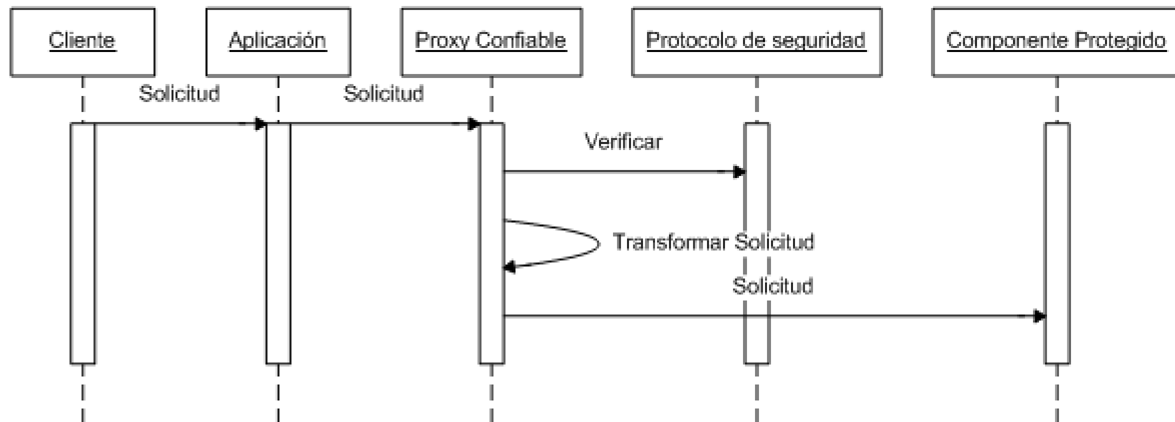

Figura 7: Tránsito normal de una solicitud hacia un componente protegido.

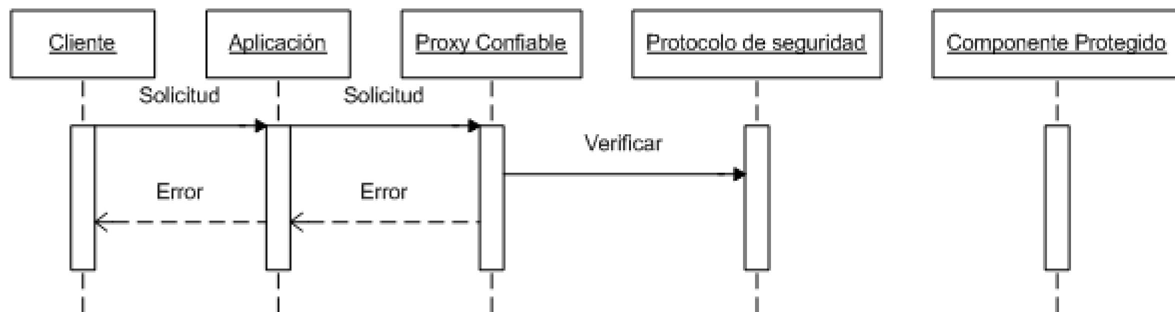

Figura 8: Rechazo de una solicitud que no cumple el protocolo de seguridad.

Canal Seguro

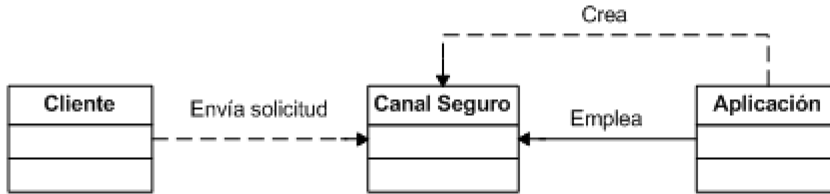

Figura 9: Estructura estática del patrón "Canal Seguro".

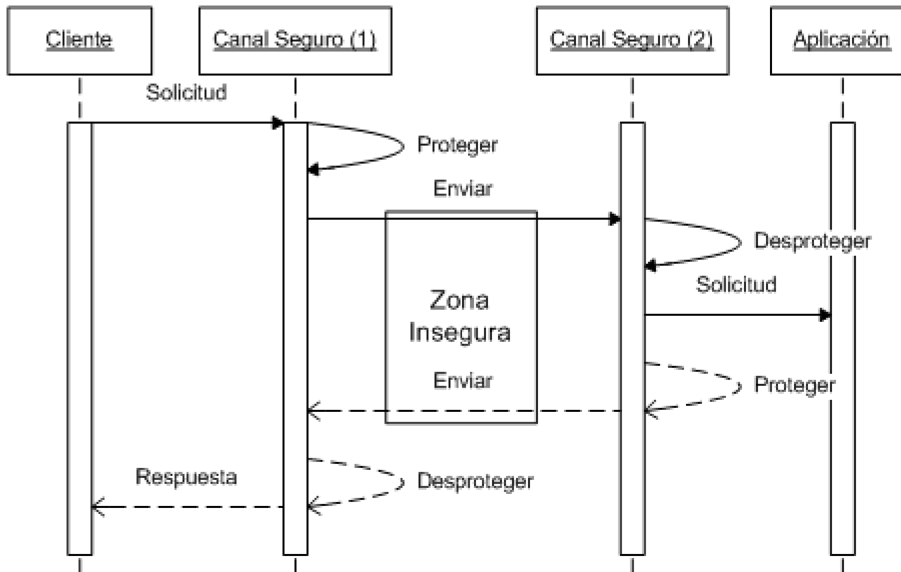

Figura 10: Protección de la información en tránsito por el Canal Seguro.

Identificador de Autenticidad de Transacciones

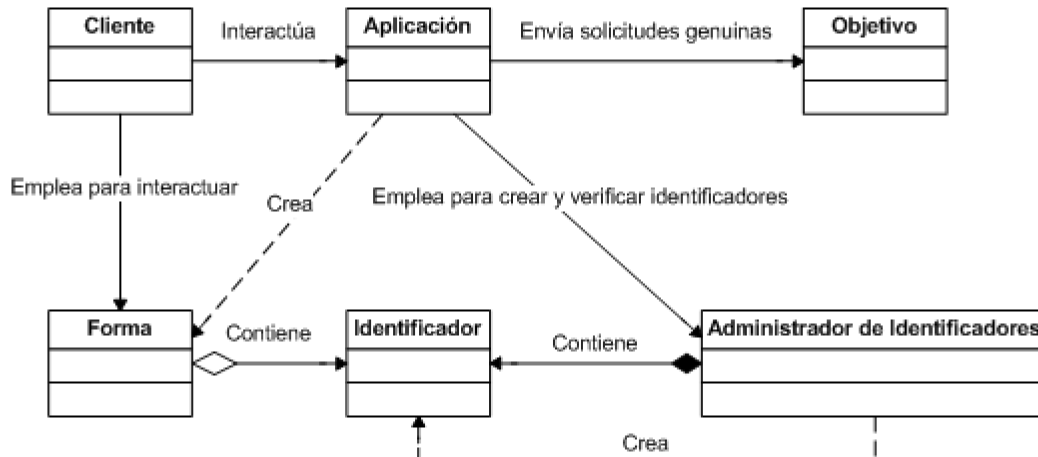

Figura 11: Estructura estática del patrón "Identificador de Autenticidad de las Transacciones".

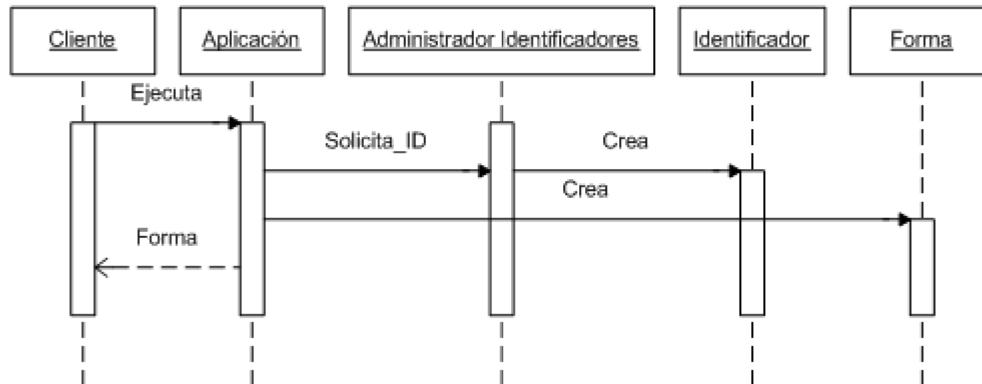

Figura 12: Creación de una forma con identificador de autenticidad.

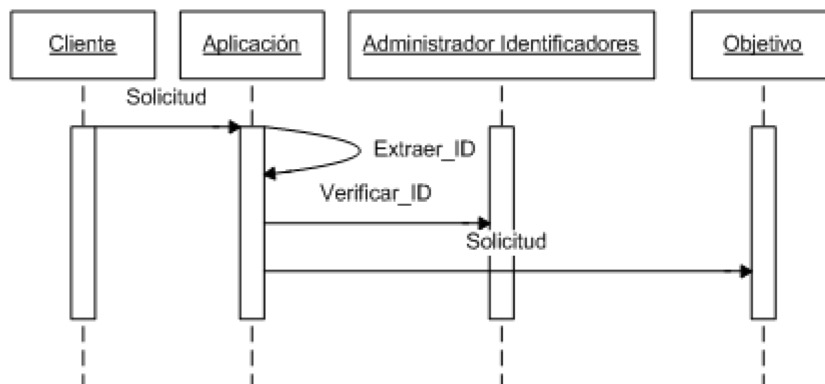

Figura 13: Envío de una solicitud genuina a su objetivo.

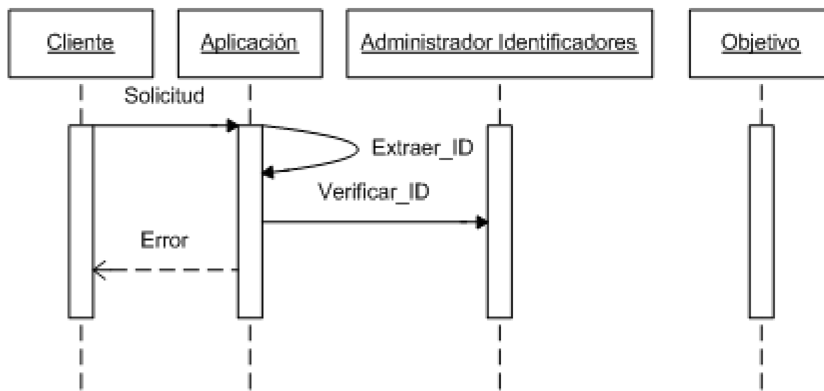

Figura 14: Rechazo de una solicitud falsificada.

Punto Único de Acceso

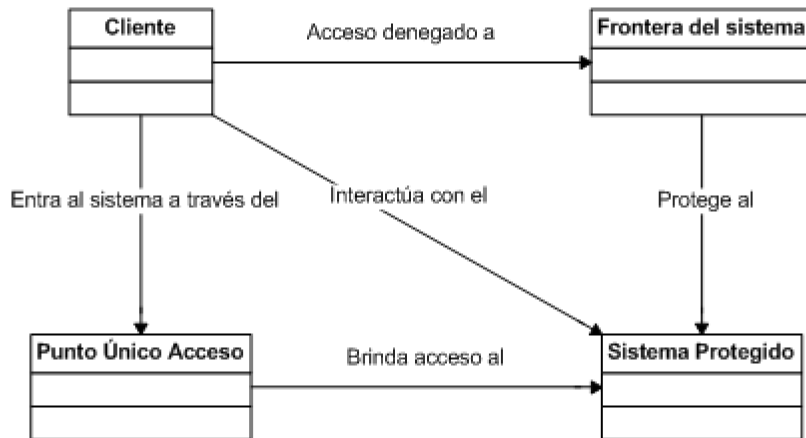

Figura 15: Estructura estática del patrón "Punto único de Acceso".

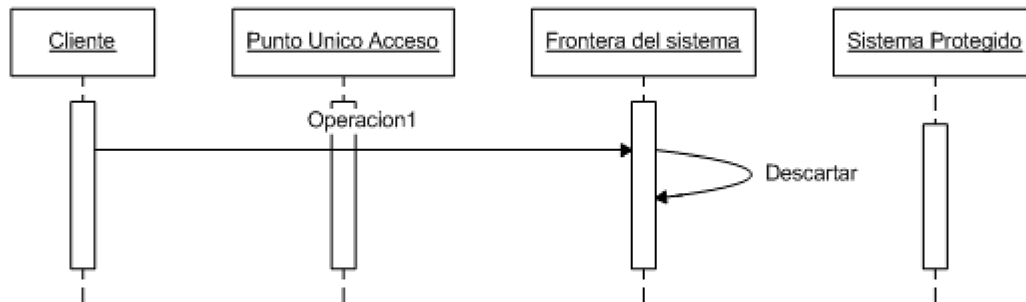

Figura 16: Denegación de las solicitudes que no pasan por el "Punto Único de Acceso".

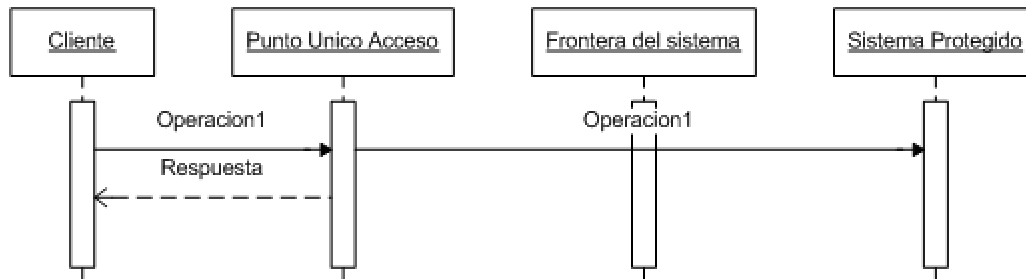

Figura 17: Acceso de las solicitudes que pasan por el "Punto Único de Acceso".

Punto de Chequeo

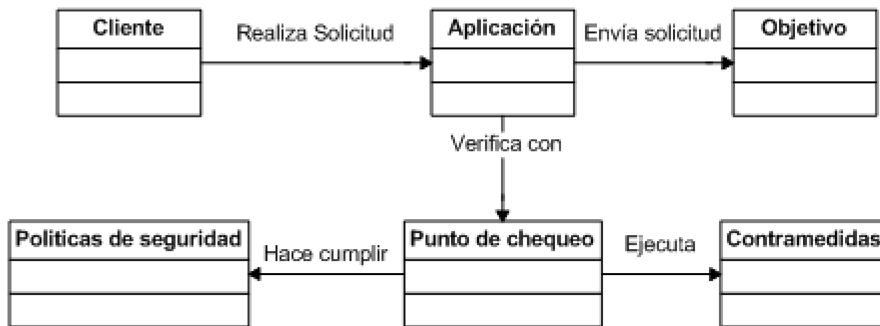

Figura 18: Estructura estática del patrón "Punto de Chequeo".

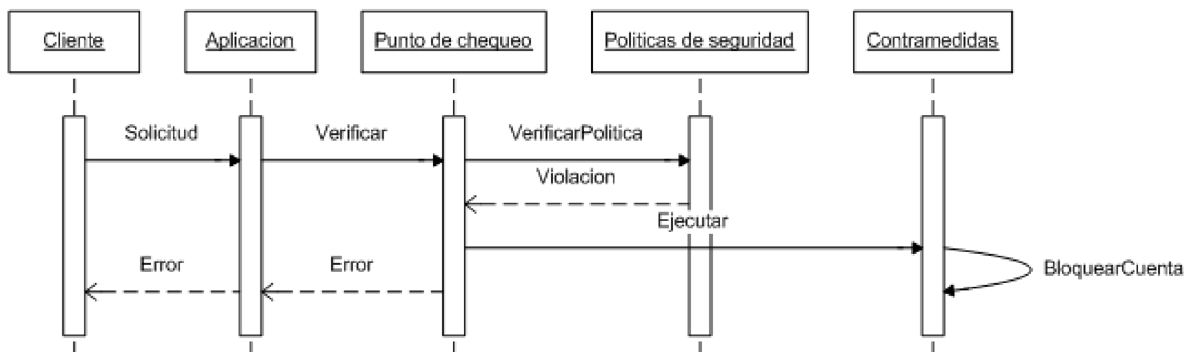

Figura 19: Rechazo y toma de medidas ante una solicitud que viola las políticas de seguridad.

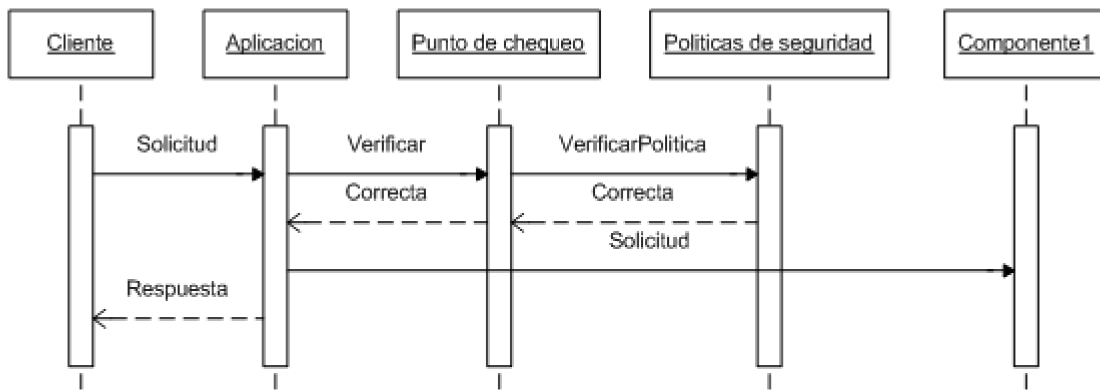

Figura 20: Tránsito normal de una solicitud que no viola las políticas de seguridad.

Sesión dirigida

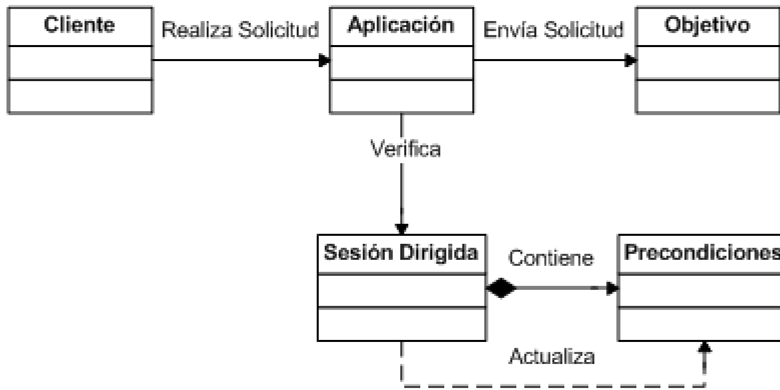

Figura 21: Estructura estática del patrón "Sesión Dirigida".

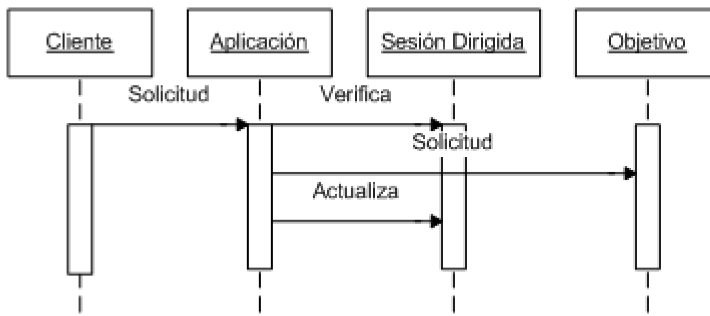

Figura 22: Tránsito normal de una solicitud que cumple con las precondiciones.

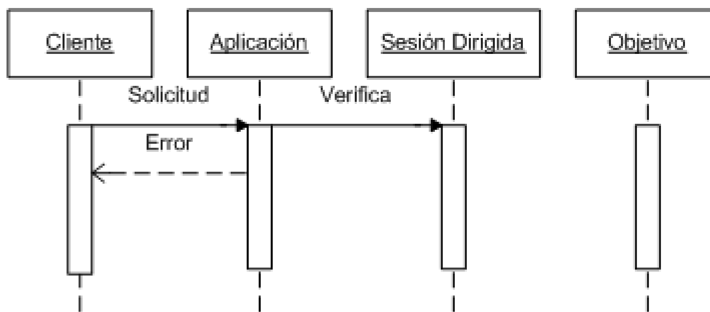

Figura 23: Rechazo de una solicitud que no ha satisfecho todas las precondiciones para ser aceptada.