UNA REVISIÓN DE METODOLOGÍAS SEGURAS EN CADA FASE DEL CICLO DE VIDA DEL DESARROLLO DE SOFTWARE

César Marulanda L.
Italtel S.P.A.
Medellín, Colombia
cesar.marulanda@italtel.com.co

Julián Ceballos H.
Productora de Software S.A. PSL
Medellín, Colombia
jceballos@psl.com.co

(Tipo de Artículo: **Reflexión**. Recibido el 03/11/2011. Aprobado el 11/01/2012)

RESUMEN
El desarrollo de software seguro es un asunto de alta importancia en las compañías, debido a que la mayoría de ellas dependen altamente de sus aplicaciones para su operación normal. Es por esto que se hace necesario implementar efectivamente metodologías de desarrollo seguro que sean aplicadas en cada fase del ciclo de vida: requisitos, diseño, desarrollo y pruebas. Es importante tener presente la seguridad desde las etapas más tempranas del proceso de desarrollo y no dejarla en un segundo plano. Además, se requiere investigar el estado del arte en desarrollo de aplicaciones seguras y así escoger metodologías de acuerdo con las necesidades de cada aplicación y los requisitos de los interesados en el producto final. El objetivo de este artículo es recopilar una serie de metodologías y herramientas existentes que se puedan implementar, añadiendo seguridad en toda la aplicación y desarrollando no sólo software de alta calidad sino también de alta seguridad.

Palabras clave
Ciclo de Vida de Desarrollo de Sistemas, Desarrollo Seguro, Pruebas Seguras, Arquitectura Segura.

A REVIEW OF SECURE METHODOLOGIES FOR ALL THE STAGES OF LIFE CYCLE OF SOFTWARE DEVELOPMENT

ABSTRACT
Secure Software Development is a high importance matter in all companies, because most of them are highly-dependent on their applications for normal operation, for this reason is necessary to effectively implement secure development methodologies to be applied in every phase of Software Development Life Cycle; Requirements, Design, Development and Tests, because is important to keep in mind that Security starting from the earlier stages in the development process, and do not put it away for late stages. Thus is important to research the state of art in secure development for applications and choose methodologies in order to satisfy the needs for the application and the requirements of final product stakeholders. The main goal of this article is to collect a set of methodologies and tools, which can be implemented, adding security in the whole application, thus developing high quality and high security software.

Keywords
Systems development life cycle, secure development, secure tests, secure architecture.

UNE REVISION DES METHODOLOGIES SÛRES POUR TOUS LES PHASES DU CYCLE DE VIE DU DEVELOPPEMENT DES LOGICIELS

RÉSUMÉ
Le développement de logiciels sécurisés est un sujet très important pour les entreprises, á cause de que la majorité d'entre eux dépendent fortement de leurs applications, pour avoir une opération normale. C'est pour cela que il est nécessaire d'implémenter effectivement des méthodologies de développement sécurisé qui soient appliqués pendant chaque phase du cycle de vie du développement logiciel ; requêtes, conception, développement et essais, puisque il est important de considérer la sécurité pendant toutes les étapes du processus de développement, au lieu de l'avoir en second plan. Par conséquent, il est important de faire des recherches sur l'état de l'art au sujet de développement d'applications sécurisées, et de choisir des méthodologies, selon la nécessité de chaque application et les requêtes actuelles des intéressés par le produit final. L'objectif de cet article est de rassembler des méthodologies et outils existantes qui pourraient être implémentés, en fournissant de sécurité supplémentaire sur tout l'application, et en développant non seulement de logiciel de haute qualité mais encore très sûr.

Mots-clés
Cycle de vie de développement de systèmes, développement sécurisé, essais sécurisés, architecture sécurise.

1. INTRODUCCIÓN

La seguridad de los sistemas es un tema que ha despertado amplio interés desde hace mucho tiempo, porque no sólo es necesario tener aplicaciones de alta calidad sino también que tengan seguridad. Debido a que actualmente la mayor parte de ataques están enfocados a las aplicaciones y teniendo en cuenta que la mayoría de desarrolladores no tienen las habilidades y el conocimiento necesario para desarrollar código seguro [2], es necesario que las empresas apliquen metodologías y herramientas que permitan desarrollar aplicaciones seguras que cumplan con las exigencias de seguridad en este tiempo.

De acuerdo con Allen et al. [1], el Ciclo de Vida de Desarrollo de Software CDVS se puede definir como un proceso iterativo en el cual se identifican cuatro etapas principales: Requisitos, Diseño, Desarrollo y Pruebas. Existen otros modelos, como CMMI [7], que permiten definir buenas prácticas para el desarrollo de software y plantean básicamente que el CVDS se considere como un micro proyecto, donde un cambio en alguna de las etapas se debe ver reflejado en todo el proyecto, que se irá refinando cada vez hasta lograr el objetivo deseado. Pero el modelo CMMI tradicional no tiene a la seguridad implementada en sus definiciones de procesos y buenas prácticas, por lo que es necesario agregar esquemas de seguridad adicionales.

Lo primero que se debe hacer para comenzar a implementar seguridad en las aplicaciones es identificar cuáles son los activos de información que se deben proteger, cómo protegerlos, cuáles son las vulnerabilidades de los elementos que interactúan con la información y como mitigarlas, al punto de reducirlas a un nivel de riesgo aceptable mediante un proceso iterativo que analice profundamente los riesgos durante todo el CVDS. También es muy importante identificar patrones de ataque en todas las fases y almacenarlos en una base de conocimiento que permita prevenir futuros ataques en otras aplicaciones [1].

La seguridad debe estar implícita desde la misma concepción del software, porque es un error dejarla para etapas posteriores del desarrollo. Se debe comenzar de los requisitos, que no deben ser simples listas de chequeos de implementación de controles, como firewalls y antivirus, sino que deben ir más enfocados a la protección de los activos críticos [1]. Para la aplicación específica que se está desarrollando se debe tener en cuenta la perspectiva del agresor. En este artículo, con el objetivo de lograr una buena aproximación al desarrollo de requisitos seguros, se describe la metodología de mala definición de casos uso, que son el caso inverso a los buenos casos de uso [1] y que definen lo que el sistema no puede permitir. También se mencionará el proceso SREP [5], que provee un método para elicitar, categorizar y priorizar requisitos de seguridad para aplicaciones y sistemas de información tecnológica. En la etapa de diseño se describe una herramienta de modelado de aplicaciones, llamada UMLsec [3], que es una extensión de UML que permite añadir características de seguridad a los diagramas del modelado; luego se hace una comparación de ésta con otras metodologías de diseño propuestas, se describen los pros y contras encontrados con el objetivo ofrecer una mejor perspectiva de lo que se puede utilizar en esta fase. En la etapa de desarrollo o codificación, se analizan las vulnerabilidades más comunes al código, como SQL Injection, Cross Site Scripting y otras de aplicaciones Web y se muestra cómo mitigar el riesgo de que sean explotadas [1]. Por último, se aborda el tema de las pruebas de seguridad [9], donde además de mencionar los diferentes tipos que existen, se abordan temas específicos como analizadores de chequeo de código estático, *sniffers* y analizadores de métricas, los cuales permiten medir la carga de los componentes asociados al desarrollo y para establecer posibles puntos de quiebre o cuellos de botella en las aplicaciones.

Este artículo está enfocado no sólo a evaluar y mostrar estas metodologías y herramientas utilizadas actualmente en el desarrollo seguro, sino también en hacer una propuesta formal de cómo implementarlas en un caso práctico.

2. JUSTIFICACIÓN

Antes se pensaba sólo en asegurar infraestructuras utilizando sistemas de seguridad, con firewalls, sistemas de detección de intrusos IDS o sistemas de prevención de intrusos IPS y herramientas para detección de virus, entre otras, pero se dejaba de lado a las aplicaciones. En los últimos años los ataques a las aplicaciones se han incrementado constantemente. Un estudio del SANS Institute, publicado en 2005 [2], revela que se estaban incrementando los ataques a las aplicaciones de antivirus y de *backup*, porque son aplicaciones que no se actualizan regularmente y corren con altos privilegios dentro del sistema operativo. Otro tipo de ataques en incremento se observa en las aplicaciones Excel y Word, en las que se aprovechan ciertas vulnerabilidades y sólo se necesitaba enviar un e-mail con uno de estos archivos adjuntos y se podía infectar una máquina y obtener su control.

La Figura 1 muestra que a través de los años la aparición de vulnerabilidades en Word y Excel ha ido incrementado notablemente, lo mismo pasa con muchas otras aplicaciones, debido a que progresivamente se desarrollan nuevas técnicas de ataque y se incrementan tanto los atacantes como las vulnerabilidades en las aplicaciones. Actualmente y debido al creciente y extendido uso de Internet, las aplicaciones Web se han convertido en los blancos muy perseguidos, porque siempre están expuestas en la red. La mayor parte de estos ataques son de Cross Site Scripting y SQL Injection, como se observa en la Figura 2. Todas estas fallas en los programas se deben a errores de desarrollo, porque la mayoría de desarrolladores no saben lo que es código seguro, porque nadie se los ha dicho y nadie se los ha exigido,

no conocen los ataques al código ni las herramientas que explotan esas vulnerabilidades y, simplemente, al no saber cómo se puede explotar el código, no saben cómo escribir de forma segura [2].

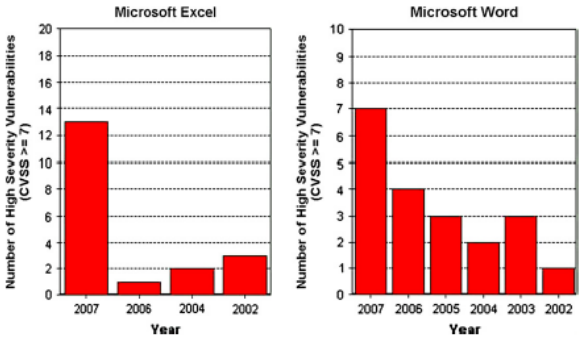

Fig. 1: Incremento de vulnerabilidad en Excel y Word [2]

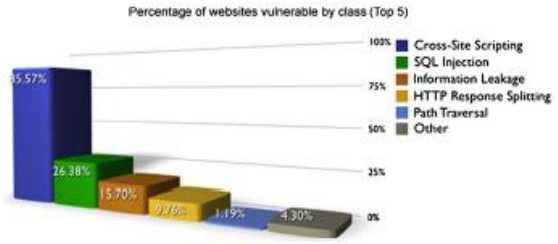

Fig. 2: Vulnerabilidades expuestas en los sitios Web [2]

De acuerdo con este panorama, se hace necesario crear metodologías, estándares y procesos y desarrollar herramientas que sean reconocidas y aceptadas de forma general para el desarrollo de aplicaciones seguras, que permitan no sólo formar a los desarrolladores en técnicas y procesos, sino que además permita evaluar y mantener seguro al código en el tiempo, con el objetivo de brindarles a los clientes no sólo software de alta calidad, sino también de alta seguridad.

3. AMENAZAS A LAS APLICACIONES SEGURAS

Es necesario definir inicialmente las condiciones que debe cumplir un sistema seguro [1]:

- Que sea poco vulnerable y libre de defectos tanto como sea posible.
- Que limite el resultado de los daños causados por cualquier ataque, asegurando que los efectos no se propaguen y que puedan ser recuperados tan rápido como sea posible.
- Que continúe operando correctamente en la presencia de la mayoría de ataques.

El software que ha sido desarrollado con seguridad generalmente refleja las siguientes propiedades a través de su desarrollo:

- **Ejecución Predecible.** La certeza de que cuando se ejecute funcione como se había previsto. Reducir o eliminar la posibilidad de que entradas maliciosas alteren la ejecución o las salidas.

- **Fiabilidad.** La meta es que no haya vulnerabilidades que se puedan explotar.

- **Conformidad.** Actividades planeadas, sistemáticas y multidisciplinarias que aseguren que los componentes y productos de software están conforme a los requisitos, procedimientos y estándares aplicables para su uso específico.

Algunas propiedades fundamentales que son vistas tanto como atributos de seguridad, como propiedades del software son:

- **Confidencialidad.** Debe asegurar que cualquiera de sus características —incluyendo sus relaciones con ambiente de ejecución y usuarios—, sus activos y/o contenidos no sean accesibles para no autorizados.

- **Integridad.** El software y sus activos deben ser resistentes a subversión. Subversión es llevar a cabo modificaciones no autorizadas al código fuente, activos, configuración o comportamientos, o cualquier modificación por entidades no autorizadas. Tales modificaciones incluyen sobre-escritura, corrupción, sabotaje, destrucción, adición de lógica no planeada —inclusión maliciosa— o el borrado. La integridad se debe preservar tanto en el desarrollo como durante su ejecución.

- **Disponibilidad.** El software debe ser funcional y accesible por usuarios autorizados cada vez que sea necesario. Al mismo tiempo, esta funcionalidad y estos privilegios deben ser inaccesibles por usuarios no autorizados.

Dos propiedades adicionales, comúnmente asociadas con usuarios humanos, se requieren en entidades de software que actúan como usuarios, por ejemplo, agentes *proxy* y servicios Web.

- **Responsabilidad.** Todas las acciones relevantes de seguridad del software o de los usuarios se deben almacenar y rastrear, con atribución de responsabilidad. Este rastreo debe ser posible en ambos casos, es decir, mientras y después de que la acción registrada ocurra. Según la política de seguridad para el sistema se debería indicar cuáles acciones se consideran como seguridad relevante, lo que podría hacer parte de los requisitos de auditoría.

- **No repudio.** Esta propiedad le permite al software y a los usuarios refutar o denegar responsabilidades de acciones que ha ejecutado. Esto asegura que la responsabilidad no puede ser derribada o evadida.

Otras propiedades influyentes en el software seguro son la exactitud, la capacidad de pronóstico, la fiabilidad y la protección, las cuales están también influenciadas por el tamaño, la complejidad y la trazabilidad del software.

- **Análisis de riesgos.** Debido a las constantes amenazas, toda organización es vulnerable a riesgos debido a la existencia de vulnerabilidades. De acuerdo con la Figura 3, para determinar el riesgo se puede evaluar la probabilidad asociada con los agentes de amenaza, el vector de ataque, las

debilidades de seguridad, los controles de seguridad, los impactos técnicos y el impacto al negocio cuando se materialice la amenaza [11].

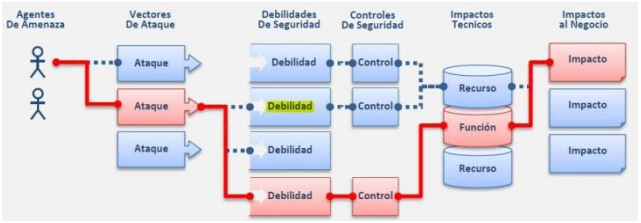

Fig. 3: Rutas atreves de las aplicaciones [11]

El OWASP —The Open Web Application Security Project [10]—, publica anualmente el top 10 de los riesgos más serios que se presentan en las organizaciones, el cual se puede observar en la Tabla I para 2010. Cada organización realiza su análisis de riesgos mediante una evaluación de éstos para detectar su severidad.

TABLA I
OWASP Top 10 2010 [10]

Top	Riesgo
A1	Injection
A2	Cross-Site Scripting (XSS)
A3	Broken Authentication and Session Management
A4	Insecure Direct Object References
A5	Cross-Site Request Forgery (CSRF)
A6	Security Misconfiguration
A7	Insecure Cryptographic Storage
A8	Failure to Restrict URL Access
A9	Insufficient Transport Layer Protection
A10	Unvalidated Redirects and Forwards

Antes de comenzar a desarrollar código seguro, lo primero que se debe hacer es un análisis de riesgos. Primero se identifican cuáles son los activos críticos de la organización que están involucrados en el proceso de desarrollo o que de alguna forma van a estar expuestos en la aplicación y luego se identifican cuáles son las posibles amenazas que pueden explotar las vulnerabilidades de estos activos. En la seguridad de la información la amenaza, la fuente del peligro, a menudo es una persona intentando hacer daño con uno o varios agentes maliciosos [1].

El software está sujeto a 2 categorías generales de amenazas:

Amenazas durante el desarrollo. Un ingeniero de software puede sabotear el programa en cualquier punto de su desarrollo.

Amenazas durante la operación. Un agresor intenta sabotear el sistema.

El software en la red está comprometido por el aprovechamiento de sus debilidades, las cuales se derivan de las siguientes fuentes:

1. Complejidad o cambios incluidos en el modelo de procesamiento
2. Suposiciones incorrectas del ingeniero, incluyendo las relacionadas con la capacidad, las salidas, el comportamiento de los estados del ambiente de ejecución del software o con entradas esperadas de entidades externas.
3. Implementación defectuosa en el diseño o en los requisitos de interfaces del software con otras entidades externas o en ambiente de ejecución de los componentes del software.
4. Interacción no planeada entre componentes de software, incluyendo los de otros proveedores [1].

Teniendo esto definido, se puede proceder a categorizar o priorizar los riesgos, con el objetivo de definir cuáles son los más pernicioso y, de esa forma, lograr una categorización que permita definir el orden en que se deben atender y para identificar dónde invertir mayor esfuerzo o dinero para mitigar el riesgo asociado.

4. METODOLOGÍAS INVESTIGADAS

4.1 Funcionamiento con carga compartida [2]

Con el objetivo de definir estándares de desarrollo de código seguro varias compañías de todo el mundo, como Siemens en Alemania, Tata en India, Nomura Research en Japón, CIBC en Londres y las americanas Kaiser Permanente, Boeing, Cisco, Symantec, Intel y American Express, se unieron a esta iniciativa. El proyecto pretende no sólo mejorar los procesos en cada una de las empresas sino el desarrollo de software en sí. Todas tienen como patrón común que sus desarrollos dependen de la calidad del software, lo que permitirá además que los clientes reciban productos seguros, ya sean desarrollados por terceros o propios. A este grupo de empresas se unieron también desarrolladores de otras organizaciones y universidades, como Amazon, Secure Compass, Universidad Carnegie Mellon y el equipo OWASP, para dar lugar al "Concilio de Programación segura" que, desde el 2007, se reúne para definir estándares y documentos que indiquen las habilidades necesarias para escribir código seguro.

Debido a la necesidad de evaluar el código y también impulsados por buscar la certificación de los desarrolladores, SANS lanzó el Software Security Institute SSI, que está enfocado en proveer información, entrenamiento y una librería de investigaciones e iniciativas para ayudarles a desarrolladores, arquitectos y administradores a proteger sus aplicaciones, incluyendo las Web. Adicionalmente, el SSI lanzó GSSP —Secure Software Programmer— que certifica a los usuarios que cumplen con todos los requisitos y habilidades requeridas para realizar codificación segura y cuenta con una evaluación online para medir sus capacidades.

4.2 Security Requirements Engineering Process [5]

El SREP es un método basado en activos y orientado a riesgos, que permite el establecimiento de requisitos de seguridad durante el desarrollo de las aplicaciones.

Básicamente lo que define este método es la implementación de Common Criteria durante todas las fases del desarrollo de software —CC es un estándar internacional ISO/IEC 15408 para seguridad informática, cuyo objetivo es definir requisitos seguros que les permitan a los desarrolladores especificar atributos de seguridad y evaluar que dichos productos si cumplan con su cometido—. En este proceso también se apunta a integración con el Systems Security Engineering Capability Maturity Model (ISO/IEC 21827), que define un conjunto de características esenciales para el éxito de los procesos de ingeniería de seguridad de una organización. En contraste con su derivado, el CMM, SSE-CMM establece una plataforma para mejorar y medir el desempeño de una aplicación, en cuanto a principios de ingeniería de seguridad, en vez de centrarse en las 22 Process Areas.

Esta metodología trata cada fase del CVDS como a un mini proceso o iteración, dentro de las cuales se aplican las actividades SREP que permiten identificar y mantener actualizados los requisitos de seguridad de la fase, permitiendo mitigar efectivamente los riesgos asociados a cada una. El proceso se detalla en la Figura 4.

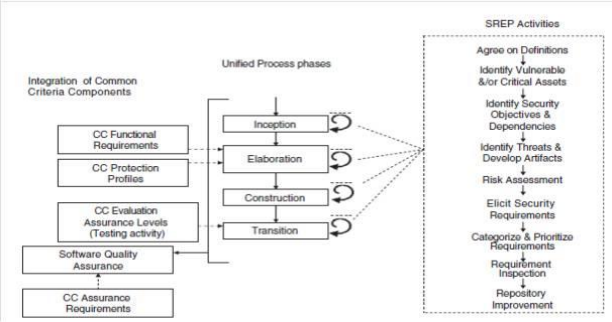

Fig. 4: El proceso SREP [5]

Las nueve actividades definidas para cada iteración son las siguientes: (1) acuerdo en las definiciones, donde se verifica que todos los participantes del software aprueben y estén de acuerdo en la definición de un conjunto de requisitos de seguridad que se adapte a las políticas de la compañía; (2) identificación de activos críticos o vulnerables, que consiste en realizar una lista detallada de los activos de información más importantes para la organización, con base en el Security Resources Repository, que debe ser obtenido previamente; (3) identificar los objetivos y dependencias de seguridad, donde se debe tener en cuenta las políticas de seguridad y los aspectos legales para definir correctamente hacia qué se apunta para definir el nivel necesario de seguridad requerido; (4) identificar amenazas y desarrollar artefactos, aquí es necesario identificar todas las amenazas que puedan afectar cada uno de los activos para desarrollar artefactos; (5) evaluación del riesgo, en esta etapa se seleccionan los riesgos a tratar y cuáles a mitigar o transferir, teniendo presente los objetivos definidos y los activos críticos de la aplicación; (6) elicitar los requisitos de seguridad, donde se documentan los requisitos de seguridad que se deben cumplir, de acuerdo con las amenazas encontradas en el nivel necesario para la aplicación, también se definen las pruebas de seguridad que se aplicarán a cada requisito o conjunto de requisitos; (7) categorizar y priorizar los requisitos, porque una vez identificados se deben ordenar por importancia de acuerdo al impacto que tengan en el cumplimiento de los objetivos; (8) inspección de requisitos, para garantizar la calidad y la consistencia de los hallados deben ser validados por el equipo de trabajo, para que se acepten y refinen de acuerdo con las observaciones encontradas y (9) mejora el repositorio, aquí se deben recopilar todos los documentos, diagramas y modelos que se hayan encontrado durante el ciclo de desarrollo y deben quedar consignados en el SRR.

4.3 SQUARE [4]

El modelo SQUARE —Security Quality Requirements Engineering— propone varios pasos para construir modelos de seguridad desde las etapas tempranas del ciclo de vida del software. En el proceso del modelo se hace un análisis enfocado a la seguridad, los patrones de ataque, las amenazas y las vulnerabilidades y se desarrollan malos casos de uso/abuso. Los pasos son:

1. *Acuerdo en las definiciones*. Se debe generar un acuerdo con el cliente en cuanto a las definiciones de seguridad, como en el control de acceso, la lista de control de acceso, el antivirus, entre otras.
2. *Identificar metas de seguridad*. Que se trazan con base en las propiedades de seguridad definidas en el documento.
3. *Desarrollar artefactos.* Diagramas de arquitectura, casos de mal uso, casos de uso de seguridad, identificar activos y servicios esenciales, árboles de ataques, patrones de ataque, requisitos de seguridad, mecanismos de seguridad.
4. *Evaluación de los riesgos*. Se analizan las amenazas y vulnerabilidades y se definen estrategias de mitigación.
5. *Elicitación de los requisitos*.
6. *Clasificar los requisitos*. De acuerdo con el nivel o las metas de seguridad.
7. *Priorizar los requisitos*. (1) Esenciales, si el producto no puede ser aceptado si él, (2) condicionales, si requerimiento incrementa la seguridad, pero el producto se acepta sin él y (3) opcionales, si es de baja prioridad frente a los esenciales y condicionales.
8. *Revisión por pares*.

4.4 CoSMo [6]

Debido a la necesidad de integrar aspectos de seguridad en el proceso de modelado de software, el modelado conceptual debe abarcar los requisitos y los mecanismos de seguridad de alto nivel. Los autores trabajan en el desarrollo de un método de modelamiento conceptual de seguridad al que denominan CoSMo —Conceptual Security Modeling—. Antes de tener la visión de que los mecanismos de seguridad pueden hacer cumplir los requisitos, se elaboran cuestiones fundamentales de políticas de seguridad; las cuales consisten de un conjunto de

leyes, normas y prácticas que regulan cómo una organización gestiona, protege y distribuye información sensible. Cada requisito de seguridad lo puede ejecutar uno o más mecanismos de seguridad, resultando en una matriz de requisitos y mecanismos. Ambos se definen genéricamente porque se usan para modelar la seguridad en el nivel conceptual. En primer lugar, los autores de CoSMo tratan de mostrar cómo integrar las consideraciones de seguridad en el marco de modelado de procesos conceptuales. Después, enumeran de forma sistemática los requisitos de seguridad frecuentes e indican claramente cuáles son los mecanismos para hacerlos cumplir.

En general, un requisito de seguridad exigido no es parte del diagrama de casos de uso, sino de la descripción del caso de uso. En CoSMo, es posible modelar este requisito en el nivel conceptual, incluso en un diagrama de casos de uso.

4.5 UMLSec [3]

Es una metodología de desarrollo basada en UML para especificar requisitos de seguridad relacionados con integridad y confidencialidad. Mediante mecanismos ligeros de extensión de UML es posible expresar los estereotipos, las etiquetas, las restricciones y el comportamiento de un subsistema en presencia de un ataque. El modelo define un conjunto de operaciones que puede efectuar un atacante a un estereotipo y trata de modelar la actuación del subsistema en su presencia. Esta metodología define varios estereotipos que identifican requisitos de seguridad, como *secrecy*, en el que los subsistemas deben cumplir con que ningún atacante pueda ver su contenido mientras este esté en ejecución y *secure link*, con el que se asegura el cumplimiento de los requisitos de seguridad en la comunicación a nivel físico; lo que pretende evaluar es que un atacante en una dependencia de un subsistema estereotipada, como *secrecy*, que tenga dos nodos *n*, *m* que se comunican a través de un enlace *l* nunca pueda leer el contenido del subsistema.

Stereotype	$\text{Threats}_{default}()$
Internet	{delete, read, insert}
encrypted	{delete}
LAN	∅
wire	∅
smart card	∅
POS device	∅
issuer node	∅

Fig. 5: Amenazas que pueden ser explotadas por un atacante dependiendo del canal [3]

La Figura 5 presenta varios escenarios con un atacante por defecto representado por uno externo con capacidad modesta. Se puede observar cuáles pueden ser las amenazas que puede explotar dependiendo del enlace de comunicación. Secure Dependency trata de asegurar que las dependencias *call* y *send* entre dos componentes cumplan con la integridad y confidencialidad, además, que los mensajes ofrecidos por un componente *C* sean seguros si y sólo si también son seguros en otro componente *D*.

El estereotipo *Critical* se usa para marcar datos que necesitan ser protegidos a toda costa; esta protección está apoyada por los estereotipos *data security* y *no-down flow*. El primero establece básicamente que toda la información establecida como *secrecy* debe permanecer igual ante cualquier amenaza.

La metodología UMLSec pretende reutilizar los diseños ya existentes, aplicando patrones para formar nuevos a partir de una transformación al existente, tal como se observa en las Figuras 6 y 7.

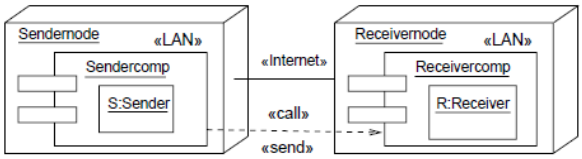

Fig. 6: Canal de Comunicación Internet [3]

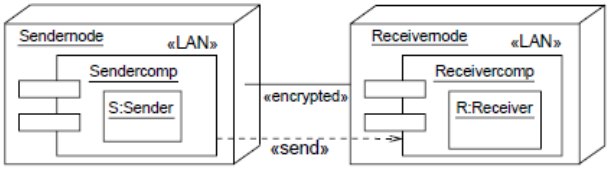

Fig. 7: Canal de Comunicación Encriptado [3]

En la Figura 6 se muestra un diagrama de componentes UML que no cumple con el estereotipo de *secrecy* para el subsistema, porque permite la lectura del atacante por defecto en el canal de Internet; mientras que en la Figura 7 se utiliza un patrón de canal seguro para encriptar la comunicación entre ambos componentes para cumplir con el estereotipo *secrecy*.

4.6 Casos de Mal Uso

Es el caso inverso de un caso de uso UML y es lo que el sistema no debería permitir. Así como en un caso de uso se define la secuencia de acciones que le dan un valor agregado al usuario, en uno de mal uso se define la secuencia de acciones que se traducen en pérdida para la organización o los usuarios interesados. Un mal actor es lo contrario a un actor. Es un usuario que no se quiere que el sistema soporte y que podría ser un agresor. En la Figura 8 se ilustra un mal uso como un caso de uso invertido.

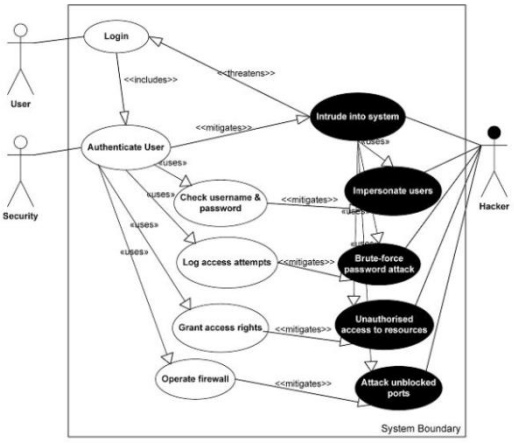

Fig. 8. Caso de uso invertido [8]

5. COMPARACIÓN DE METODOLOGÍAS Y HERRAMIENTAS

Las metodologías expuestas tienen un patrón común y es que todas son formales, pero todas presenta la misma falencia: ninguna provee una herramienta automática de verificación interna; esto amerita la búsqueda de una metodología que lo resuelva.

Estas metodologías aseguran al software en todas las etapas del CVSD, pero en hacen mayor énfasis en alguna de esas etapas. Por ejemplo, SREP se enfoca en los requisitos mientras que UMLSec lo hace en la etapa de diseño y desarrollo. Esto hace difícil seleccionar una sola metodología y sería mucho mejor y más completo seleccionar lo mejor de algunas de ellas para aplicarlo y cubrir la mayor parte de las brechas que cada una deja. Se recomienda SREP para la fase de requisitos, porque contiene un conjunto de buenas prácticas y permite hacer un análisis de riesgos profundo en cada una de las fases del CVDS. Por otro lado, se basa en estándares internacionales, a diferencia de propuestas como SQUARE. Además, muchos de los artefactos generados en SREP se pueden basar en casos de Mal Uso apoyados en UML. En el diseño y codificación sería bueno utilizar UMLsec, porque UML es un lenguaje bien definido, ampliamente aceptado y aplicado en la mayoría de productos software, por lo que sería muy útil y fácil de implementar en cualquier aplicación; además, permitiría reutilizar gran parte de la documentación existente en las aplicaciones actuales. Esta metodología tiene en cuenta la perspectiva del atacante, lo que permite tener un mejor control en el diseño de las vulnerabilidades de la aplicación.

Para la etapa de desarrollo y pruebas no se puede definir una metodología estricta, debido a que estas etapas dependen más de las habilidades y competencias del equipo de desarrollo y de los probadores; por esto es posible afirmar que en estas etapas existen muchas pautas que en la mayoría de los casos están apoyadas en diferentes herramientas automáticas que favorecen la seguridad de las aplicaciones. A continuación se presenta una lista de pautas que pueden ser útiles para prevenir algunas de las vulnerabilidades más comunes.

- En el desarrollo.
  - Herramientas de chequeo estático de código.
  - Herramientas de ofuscamiento de código.
  - Estándares de codificación para código seguro.
  - Validación de entradas/salidas de datos.
  - Correcto manejo de excepciones.
  - Trazabilidad en todo el flujo de la aplicación a través de *logs*.
  - Auditoria de clases o tablas sensibles para evitar el no repudio.

- En Pruebas
  - Pruebas de penetración.
  - Pruebas de carga.
  - Monitoreo de la ejecución de la aplicación.

Para llevar a cabo estas pautas, generalmente se utilizan ciertas herramientas automáticas que permiten evaluar posibles riesgos o vulnerabilidades de seguridad y que no se ven a simple vista o no son identificadas por una persona, tal vez por posibles sesgos mentales o por desconocimiento de temas de seguridad de la información, por el *Copy/Paste*, o en algunos casos por los mapas o estructuras mentales que les impide realizar las cosas forma diferente o buscar nuevas formas de actualizarse. Además, permiten identificar la causa raíz del problema de manera temprana. Algunas de las herramientas para análisis de código estático se observan en la Figura 8.

Nombre	Licencia	Descripción
BOON	académico	Verificador de modelos orientado a deterctar buffer overflow en C.
BugFind	open source	Defector de defectos (y escáner de seguridad muy básico) para Java.
Bugscam	open source	Analizador sobre IDA. Busca llamadas peligrosas en código binario ejecutable.
Compuware DevPartner SecurityChecker	comercial	Escáner de seguridad de código para .NET (C# y VB.NET)
CodeScan	comercial	Análisis estático de seguridad para ASP y PHP
Coverity Prevent	comercial	Detector de defectos y escáner de seguridad para C/C++.
Cqual	académico	Analizador de flujo de datos para C, basado en análisis de tipos.
Flawfinder	open source	Escáner de seguridad para lenguaje C.
Fortify SCA	comercial	Escáner de seguridad para diversos lenguajes.
Grammatech CodeSonar	comercial	Vulnerabilidades y otros defectos en C/C++ y ADA.
HP DevInspect	comercial	C#, Java, HTML/XML, JavaScript, SOAP
ITS4	freeware	Busca llamadas a funciones potencialmente peligrosas en código C.
Klockwork Insight	comercial	Escáner de seguridad para lenguajes C/C++ y Java.
McCabe IQ	comercial	Analizador de flujo de control general y cobertura. Soporta C, C++, C#, Java, Fortran, VB, COBOL, y otros.
Microsoft FxCop, prefast, CAT.NET/XSSDetect	freeware	Analizadores de código para lenguajes .NET, y C/C++ (flag /analyze)
MOPS	académico	Verifica vulnerabilidades en secuencias de llamadas a funciones C.
OunceLabs Ounce5	comercial	Escáner de seguridad para C/C++, Java/JSP, .NET (C#/VB.NET) y VB6/ASP
OWASP LAPSE	open source	Escáner simple de seguridad para Java (plugin Eclipse).
Parasoft *Test	comercial	Análisis estático general, con algunas reglas de seguridad, para Java, C/C++, .NET y HTML
Pixy	open source	Inyección SQL y XSS (para PHP)
PHP-SAT	open source	Analizador para PHP
Pscan	open source	Busca llamadas a funciones potencialmente peligrosas en código C.
RATS	open source	Busca llamadas a funciones potencialmente peligrosas en código C.
smatch	open source	Defector de defectos (y escáner de seguridad) para C/C++.
splint	open source	Busca vulnerabilidades potenciales y malas prácticas de codificación en el código C.

Fig. 8: Herramientas de análisis de código estático [12]

Como se observa en esta Figura, muchas herramientas son OpenSource, por lo que no incluyen costos adicionales a los proyectos, algo que preocupa mucho a las empresas que aún no implementan procedimientos de seguridad y que se excusan en este tipo de prejuicios para no llevar a cabo la implementación de estas medidas preventivas.

Estar herramientas se utilizan en diferentes plataformas y tipos de aplicaciones. Muchas analizan librerías inseguras, tanto propias como externas —incluyendo librerías de servidores de aplicaciones y *frameworks* en los que fue desarrollado el aplicativo—, o patrones de duplicidad de código —*copy/paste*— que se pueden convertir en código inseguro replegado por toda la aplicación, o validando patrones de rutinas inseguras en las llamadas, las entradas y los flujos de datos; otras, por el contrario, verifican que se cumpla un estándar de codificación que evite la inyección de vulnerabilidades por cuenta de errores humanos, como errores de sintaxis, mal uso de punteros, variables no inicializadas, procedimientos, variables y funciones no utilizadas, presentación de salida de datos al usuario con información sensible de base de datos, o la aplicación y el manejo correcto de excepciones para evitar ataques de denegación de servicios o de *overflow* , entre otros.

Además de las herramientas de análisis de código estático existe otro tipo de herramientas que permiten ofuscar el código; esto consiste en hacer ilegible el código fuente sin afectar la funcionalidad del mismo,

evitando de esta forma que se tenga acceso indebido al código fuente o que puedan entenderlo o incluso hasta copiarlo. Algunas de esas herramientas son: Proguard [13], un ofuscador de código Java, Jabson [14], ofuscador de código Javascript y Skater.NET [15], ofuscador de código .NET

Como se mencionó antes, estas herramientas se utilizan en la etapa de desarrollo y su gran ventaja, además de identificar vulnerabilidades en una etapa muy temprana del CVDS, es que son baratas y no requieren que la aplicación este desplegada o instalada. Aunque su mayor desventaja puede ser la detección de falsos positivos dentro del código, debido a se basan en patrones de reglas pre-establecidas para que los analizadores semánticos las puedan reconocer. Esto hace que su margen de error en detección de vulnerabilidades se incremente directamente de forma proporcional al número de líneas de código de la aplicación, por lo que es necesario que los resultados se evalúen, analicen y refinen continuamente por el equipo de desarrollo, con el objetivo de evitar inconvenientes innecesarios.

En la etapa de pruebas existen varias herramientas que son de gran utilidad para lograr buenos resultados en pruebas de penetración, de carga y de monitoreo del tráfico y el flujo de información de la aplicación. Entre estas se encuentran:

- Webgoat. Una guía que sugiere los pasos para las pruebas de vulnerabilidad.
- Webscarab. Con la que se prueban diferentes tipos de ataque como inyección, *hijacking*, *cookies tampering*, *man in the middle, fuzzing.*
- Wireshark. Un *sniffer* que permite revisar el flujo de paquetes en la red con el fin de identificar contraseñas o sentencias SQL que viajen sin cifrar.
- jMeter [9]. Permite similar concurrencia de usuarios y de esta forma similar ataques de DoS; soporta el uso de *cookies*.

6. CONCLUSIONES

El diseño de aplicaciones seguras se ha convertido en un tema de alta prioridad. Debido al auge que la seguridad de la información ha tenido en los últimos años y a la dependencia que las empresas tienen de sus aplicaciones, se hace necesario garantizar su correcto funcionamiento mediante protección a los ataques internos y externos.

El análisis de riesgos se convierte en un factor común en cualquier metodología de desarrollo seguro, porque es imperativo conocer cuáles son los activos críticos de la organización y cuáles son las amenazas asociadas con ellos. Es recomendable, para cualquier aplicación, utilizar metodologías basadas en UML porque son aceptadas de forma general y no requieren mayor esfuerzo en entrenamiento del equipo de desarrollo para su implementación.

Por otro lado, también se recomiendan los modelos iterativos, debido a que permiten refinar y sostener las metodologías en el tiempo, ya que no basta sólo con implementar una buena metodología sino también que debe ser mejorada y refinada continuamente; porque en cada iteración se pueden encontrar nuevas falencias o aspectos a mejorar que enriquecen el producto final.

El uso de herramientas automáticas para análisis de vulnerabilidades en etapas de desarrollo y de pruebas es muy recomendado, siempre y cuando se monitoreen y refinen continuamente los resultados por uno o varios integrantes del equipo de trabajo.

7. REFERENCIAS

[1] J. H. Allen et al. "Software Security Engineering: A guide for Project Managers". Addison-Wesley. 2008.

[2] M. Brown & A. Paller. "Secure software development: Why the development world awoke to the challenge". *Information Security Technical Report*, Vol. 13, No. 1, pp. 40-43, 2008.

[3] J. Jürjens. "Foundations for Designing Secure Architectures". *Electronic Notes in Theoretical Computer Science*, Vol.142, pp. 31-46, 2006.

[4] N. R. Mead & T. Stehney. "Security Quality Requirements Engineering (SQUARE) Methodology". *ACM SIGSOFT Software Engineering Notes*, Vol. 30, No. 4, pp. 1-7, 2005.

[5] D. Mellado D.; E. Fernandez-Medina & M. Piattini. "A common criteria based security requirements engineering process for the development of secure information systems". Computer Standards & Interfaces, Vol. 29, No. 2, pp. 244-253, 2007.

[6] R. Villaroel; E. Fernandez-Medina & M. Piattini. "Secure information systems development: A survey and comparison". Computers & Security Vol. 24, No. 4, pp. 308-321, 2005.

[7] CMMI Product Team. "CMMI® for Development. Version 1.22". CMU/SEI-2006-TR-008. Carnegie Mellon, 2006.

[8] W. Brooks & M. Warren. "A Methodology of Health Information Security Evaluation". In J. Westbrook et al (Eds.), HIC 2006 and HINZ 2006: Proceedings. Brunswick East, Vic.: Health Informatics Society of Australia, pp. 464-470, 2006.

[9] D. Nevedrov. "Using JMeter to Performance Test Web Services", 2006. Online [Oct. 2011].

[10] Fundación OWASP. https://www.owasp.org.

[11] Fundación OWASP. https://www.owasp.org/index.php/Top_10_2010-Main. [Oct. 2011].

[12] OWASP Spain Charter Meeting III. "Herramientas para análisis estático de seguridad: estado del arte". Online [Oct. 2011].

[13] J. Sanz A. "Ofuscadores de Código Java". *Sólo Programadores*, No. 83, pp. 6-11, 2001.

[14] JavaScript Obfuscation Fascination. www.jason.com. [Oct. 2011].

[15] Rustemsof. http://www.rustemsoft.com/obfuscator_net.asp. [Oct. 2011].