GUÍA/NORMA DE SEGURIDAD DE LAS TIC (CCN-STIC-422)

DESARROLLO SEGURO DE APLICACIONES WEB

JULIO 2013

Edita:

Fecha de Edición: julio de 2013
S2 Grupo ha participado en la elaboración y modificación del presente documento y sus anexos.

LIMITACIÓN DE RESPONSABILIDAD

El presente documento se proporciona de acuerdo con los términos en él recogidos, rechazando expresamente cualquier tipo de garantía implícita que se pueda encontrar relacionada. En ningún caso, el Centro Criptológico Nacional puede ser considerado responsable del daño directo, indirecto, fortuito o extraordinario derivado de la utilización de la información y software que se indican incluso cuando se advierta de tal posibilidad.

AVISO LEGAL

Quedan rigurosamente prohibidas, sin la autorización escrita del Centro Criptológico Nacional, bajo las sanciones establecidas en las leyes, la reproducción parcial o total de este documento por cualquier medio o procedimiento, comprendidos la reprografía y el tratamiento informático, y la distribución de ejemplares del mismo mediante alquiler o préstamo públicos.

PRÓLOGO

El uso masivo de las tecnologías de la información y las telecomunicaciones (TIC), en todos los ámbitos de la sociedad, ha creado un nuevo espacio, el ciberespacio, donde se producirán conflictos y agresiones, y donde existen ciberamenazas que atentarán contra la seguridad nacional, el estado de derecho, la prosperidad económica, el estado de bienestar y el normal funcionamiento de la sociedad y de las administraciones públicas.

La Ley 11/2002, de 6 de mayo, reguladora del Centro Nacional de Inteligencia, encomienda al Centro Nacional de Inteligencia el ejercicio de las funciones relativas a la seguridad de las tecnologías de la información en su artículo 4.e), y de protección de la información clasificada en su artículo 4.f), a la vez que confiere a su Secretario de Estado Director la responsabilidad de dirigir el Centro Criptológico Nacional en su artículo 9.2.f).

Partiendo del conocimiento y la experiencia del CNI sobre amenazas y vulnerabilidades en materia de riesgos emergentes, el Centro realiza, a través de su Centro Criptológico Nacional, regulado por el Real Decreto 421/2004, de 12 de marzo, diversas actividades directamente relacionadas con la seguridad de las TIC, orientadas a la formación de personal experto, a la aplicación de políticas y procedimientos de seguridad, y al empleo de tecnologías de seguridad adecuadas.

Una de las funciones más destacables del Centro Criptológico Nacional es la de elaborar y difundir normas, instrucciones, guías y recomendaciones para garantizar la seguridad de los sistemas de las tecnologías de la información y las comunicaciones de la Administración, materializada en la existencia de la serie de documentos CCN-STIC.

Disponer de un marco de referencia que establezca las condiciones necesarias de confianza en el uso de los medios electrónicos es, además, uno de los principios que establece la ley 11/2007, de 22 de junio, de acceso electrónico de los ciudadanos a los servicios públicos, en su artículo 42.2 sobre el Esquema Nacional de Seguridad (ENS).

Precisamente el Real Decreto 3/2010 de 8 de Enero de desarrollo del Esquema Nacional de Seguridad fija los principios básicos y requisitos mínimos así como las medidas de protección a implantar en los sistemas de la Administración, y promueve la elaboración y difusión de guías de seguridad de las tecnologías de la información y las comunicaciones por parte de CCN para facilitar un mejor cumplimiento de dichos requisitos mínimos.

En definitiva, la serie de documentos CCN-STIC se elabora para dar cumplimiento a los cometidos del Centro Criptológico Nacional y a lo reflejado en el Esquema Nacional de Seguridad, conscientes de la importancia que tiene el establecimiento de un marco de referencia en esta materia que sirva de apoyo para que el personal de la Administración lleve a cabo su difícil, y en ocasiones, ingrata tarea de proporcionar seguridad a los sistemas de las TIC bajo su responsabilidad.

Julio de 2013

Félix Sanz Roldán
Secretario de Estado
Director del Centro Criptológico Nacional

ÍNDICE

ANEXOS

1. INTRODUCCIÓN

1. El presente documento es una guía que se enmarca en el conjunto de normas desarrolladas por el Centro Criptológico Nacional dentro de la serie CCN-STIC, formada por un conjunto de guías cuyo objeto es facilitar la aplicación de seguridad en distintas tecnologías que proporcionan servicios básicos en el entorno de la Administración Pública.

2. Estas guías son de aplicación para la Administración Pública y de obligado cumplimiento para aquellos sistemas que manejen información clasificada nacional.

2. ALCANCE

3. Esta guía establece las directrices de seguridad que las organizaciones deben cumplir o considerar en el ámbito del desarrollo de aplicaciones web, cubriendo todo el ciclo de vida del software, desde el análisis hasta la implantación y mantenimiento –no en exclusiva la programación en un lenguaje determinado-. Por tanto, es de aplicación a los equipos de desarrollo que abordan un proyecto específico (o la totalidad de éstos) en una organización, así como a las áreas de seguridad correspondientes bajo cuya responsabilidad se encuentra garantizar la integridad, confidencialidad y disponibilidad de la información.

3. OBJETO

4. El objetivo principal de esta guía CCN-STIC es, como se ha indicado, marcar las directrices de seguridad que deben ser aplicadas en el proceso de desarrollo de aplicaciones web (aunque parte de su contenido puede extenderse a otras aplicaciones). Se identifican y analizan controles en cada una de las fases del proceso, desde las más tempranas, en las que ya se debe incluir la seguridad de la información, hasta las correspondientes a la puesta en marcha en entornos de producción, así como al soporte y mantenimiento asociados a cualquier desarrollo de aplicaciones.

4. CONCEPTOS PREVIOS

5. La seguridad en el proceso de desarrollo puede ser un elemento complejo, en especial en el desarrollo o mantenimiento de grandes aplicaciones, en el que intervienen actores ligados tanto a equipos de desarrollo como ajenos a éstos (Seguridad, Dirección…). Por este motivo, se ha estructurado la presente guía CCN-STIC en apartados diferenciados, de forma que cada uno de ellos cubra un área de conocimiento específica del proceso de desarrollo; en términos generales, cada capítulo puede tener entidad propia y puede ser abordado de forma independiente, aunque en algunos casos las directrices expuestas referencian a puntos desarrollados en otros apartados.

6. A continuación se presenta la estructura de la guía con una breve descripción de los contenidos de cada capítulo, así como una orientación del perfil al que va dirigido en términos generales.

4.1. SEGURIDAD DEL PROCESO DE DESARROLLO

7. En este capítulo se abordan un conjunto de recomendaciones relacionadas con los objetivos a perseguir en el ámbito de la adquisición, desarrollo y mantenimiento de sistemas de información según lo establecido por el dominio de control correspondiente de la norma ISO 27002. Además se marcan algunas consideraciones adicionales de índole organizativa, necesarias para garantizar un adecuado nivel de comunicación, coordinación y soporte entre áreas para garantizar la seguridad del proceso de desarrollo.

8. Las directrices recogidas en este capítulo van dirigidas principalmente a personal involucrado en la gestión, coordinación o dirección de proyectos, así como al personal involucrado en la definición de políticas.

4.2. ANÁLISIS DE CASOS DE ABUSO

9. Este capítulo desarrolla la forma de elaborar la documentación de casos de abuso en el ámbito de un proyecto de desarrollo de aplicaciones web. La preparación de casos de abuso permite a las organizaciones modelar los escenarios en los que la aplicación pueda ser usada de forma indebida, siendo un medio para evaluar y definir cómo debe comportarse ante circunstancias especiales. La primera parte del capítulo introduce los conceptos básicos de la elaboración de casos de abuso, para pasar a continuación a proponer plantillas tipo que pueden ser utilizadas directamente o adaptadas en cada organización.

10. Las directrices recogidas en este capítulo van especialmente dirigidas a **personal involucrado en las tareas de análisis**.

4.3. ANÁLISIS DE RIESGOS

11. El objeto de este capítulo es la presentación de un análisis de modelado de amenazas (*ThreatModelling*) general para el desarrollo. El modelado de amenazas constituye esencialmente una técnica de análisis de riesgos orientada a aplicaciones y facilita a los desarrolladores de los entornos el análisis de las amenazas que afectan a sus sistemas, permitiendo optimizar la asignación de recursos a aquellas partes del sistema que más lo necesitan y centrar los esfuerzos en los puntos débiles de la aplicación.

12. Las directrices recogidas en este capítulo van especialmente dirigidas a **personal involucrado en las tareas de análisis**.

4.4. REQUISITOS LEGALES DE LAS APLICACIONES

13. En este apartado se presentan algunos de los aspectos legales que deben ser tenidos en cuenta a la hora de abordar el desarrollo de nuevos aplicativos o de mantener los ya existentes, con especial hincapié en el ámbito de la protección de datos de carácter personal.

14. Las directrices recogidas en este capítulo van dirigidas a **personal involucrado en las tareas de análisis y diseño** de las aplicaciones así como a aquellas **involucradas en el ámbito del *compliance***, especialmente legal.

4.5. DIRECTRICES DE DISEÑO

15. Este capítulo recoge un conjunto de directrices para el diseño seguro de las aplicaciones. Estas directrices van dirigidas a **personal involucrado en el diseño y la arquitectura de las aplicaciones**.

4.6. DIRECTRICES DE DESPLIEGUE

16. Este capítulo recoge un conjunto de directrices destinadas a reducir el nivel de riesgo de la plataforma tecnológica en la que se despliegan las aplicaciones desarrolladas en la organización, tanto desde el punto de vista técnico como desde el punto de vista organizativo.
17. Las directrices recogidas en este capítulo van especialmente dirigidas a personal involucrado en el diseño de la arquitectura de sistemas, en el diseño de las aplicaciones y en las operaciones de despliegue y puesta en producción.

4.7. DIRECTRICES DE PROGRAMACIÓN (JAVA)

18. Este capítulo recoge directrices técnicas de programación para la plataforma tecnológica Java, especialmente utilizada en el desarrollo de aplicaciones web; dichas directrices se acompañan de ejemplos de código fuente que ilustran el problema, la forma de explotarlo y las medidas a tomar para evitar su aparición.
19. Las directrices recogidas en este capítulo van dirigidas a **personal involucrado en las tareas de diseño y programación**.

4.8. DIRECTRICES DE AUDITORÍA

20. Una auditoría de seguridad de un producto o un sistema en una organización tiene como objetivo detectar vulnerabilidades o debilidades que puedan suponer un riesgo para dicho entorno o para la organización en su conjunto. Este capítulo recoge directrices para la realización de auditorías ligadas al producto y a la plataforma que garanticen el cumplimiento de las directrices realizadas en capítulos anteriores de esta guía.
21. Las directrices contempladas en este capítulo van dirigidas al personal que participa en tareas de auditoría de las aplicaciones y los sistemas.

5. SEGURIDAD DEL PROCESO DE DESARROLLO

22. En este apartado se analizan los objetivos a perseguir en el ámbito de la adquisición, desarrollo y mantenimiento de sistemas de información según lo establecido por el dominio de control correspondiente de la norma ISO 27002 junto con algunas consideraciones adicionales de índole organizativa necesarias para garantizar un adecuado nivel de comunicación, coordinación y soporte entre áreas para mantener la seguridad del proceso.

5.1. ALINEACIÓN DEL DESARROLLO CON LA POLÍTICA DE SEGURIDAD

23. Cualquier desarrollo realizado en la organización debe estar en todo momento alineado con las políticas de seguridad corporativas. Estas políticas deben ser revisadas al menos de forma anual, en el ámbito del desarrollo de software –aunque esta directriz debe ser extrapolada a cualquier otro ámbito- para garantizar que se ajustan a los requerimientos de seguridad corporativos, incluida la normativa vigente que pueda afectar al desarrollo de aplicaciones.
24. A la hora de abordar un proyecto de desarrollo, la organización debe identificar y analizar las políticas y procedimientos de seguridad corporativos que puedan aplicar al proyecto para extraer de ellos los requisitos correspondientes; en el caso de considerar indefinida dicha normativa, la organización debe analizar la conveniencia de refinar suficientemente dicha normativa para que pueda aplicarse en el proyecto de desarrollo, considerando en todo momento que cualquier instanciación o detalle de una política superior debe ser más restrictiva que ésta.

5.2. ESPECIFICACIÓN DE REQUISITOS DE SEGURIDAD

25. Los requisitos de seguridad de cualquier desarrollo realizado en la organización deben ser identificados y validados con anterioridad a la implementación, en las fases tempranas del proceso. Estos requisitos deben ser identificados durante la fase de análisis del proyecto y justificados, consensuados y documentados, formando parte del análisis de requisitos de servicio.
26. Los requisitos de seguridad deben ser identificados en la etapa de análisis de requisitos, tanto para nuevos desarrollos como para modificaciones o ampliaciones de los existentes, evaluándose y justificándose como parte de la fase de análisis del sistema. En los requerimientos debe indicarse qué funcionalidades debe cumplir el nuevo desarrollo –o la ampliación del existente- en al menos los siguientes aspectos:
27. Controles de acceso a la aplicación y a los datos.
28. Control y registro de acceso.
29. Control y registro de modificaciones.
30. Seguridad de los archivos informáticos y sistemas que los soportan.
31. Para realizar este análisis se debe tomar en consideración el valor de los activos desde el punto de vista del proceso/servicio, el nivel de confidencialidad y las implicaciones de su divulgación, modificación o pérdida, tanto desde el punto de vista legal como desde el punto de vista económico, reputacional, etc.
32. Los proyectos de desarrollo deben ajustarse a la política de **clasificación, marcado y tratamiento de la información** definida en la organización, contemplando al menos los siguientes aspectos:
33. Clasificación.
34. Marcado.
35. Restricciones de difusión.
36. Restricciones de tratamiento.
37. Tiempos de vida.
38. El esquema anterior debe aplicarse a toda la información de los proyectos de desarrollo, tanto en lo referente al desarrollo (código fuente, documentación, informes de auditoría...) como en lo referente a la explotación del proyecto.
39. La **documentación** asociada a un proyecto debe considerarse parte intrínseca del software, por lo que debe ser protegida de forma equivalente a como se debe proteger el código fuente; aparte de las restricciones de clasificación, tratamiento y marcado de la información anteriores, la organización debe evaluar la conveniencia de utilizar un repositorio centralizado de documentación en el que se permita el control de versiones, por ejemplo en un sistema de gestión documental. Las alternativas a estos entornos pasan en muchos casos por directorios de documentación centralizados en un servidor; si se considera esta aproximación (menos trazable que la anterior, pero válida en muchas ocasiones), debe restringirse adecuadamente, en lo relativo a permisos de usuarios y grupos, el acceso a la documentación del proyecto, tanto técnica como de usuario, garantizando que únicamente el personal autorizado en cada caso tiene acceso a la información correspondiente.

40. Todo nuevo desarrollo debe disponer de forma obligatoria de un apartado de Requisitos de Seguridad en el documento de Análisis de Requisitos, en el que se contemplen los requerimientos marcados con anterioridad.

5.3. CONTROL DEL DISEÑO Y PROGRAMACIÓN DE LAS APLICACIONES

5.3.1. CONTROL DE ACCESO AL CÓDIGO FUENTE

41. El intercambio de código fuente no debe realizarse en ningún caso mediante elementos fuera del dominio protegible de la organización; la organización debe definir e implantar un sistema de control de versiones acorde a sus requisitos de funcionalidad y seguridad (CVS o SVN son quizás los más habituales). La organización debe implantar estos sistemas para centralizar el código en sus diferentes releases, así como para garantizar un repositorio único de información, algo básico para efectuar copias de seguridad, descargar una versión concreta del código, evitar pérdidas de información, permitir una reacción segura ante regresiones o mantener la trazabilidad del acceso y los cambios sobre el código fuente.

42. El acceso al código fuente de los desarrollos debe estar limitado únicamente al personal del área correspondiente, así como a personal externo que colabore en el diseño, programación o implantación del software, siempre bajo acuerdos contractuales de confidencialidad. El otorgamiento de acceso se debe realizar siempre bajo la política de *needtoknow*, asignando el privilegio necesario únicamente a aquellas personas que lo necesiten de forma obligatoria para el correcto desempeño de sus funciones.

5.3.2. MODIFICACIONES NO AUTORIZADAS

43. Durante el ciclo de vida del desarrollo de software la organización debe planificar y ejecutar auditorías periódicas de la integridad del código fuente, para garantizar que no ha sido sometido a modificaciones no autorizadas, como la inserción de código malicioso como backdoors, troyanos o canales ocultos.

44. En caso de detectar alguna modificación se debe restaurar la última copia cuya integridad esté verificada y se debe notificar de la situación al área de Seguridad, con objeto de que analice el incidente para determinar el origen y el impacto del mismo, así como para identificar soluciones que lo eviten en el futuro.

5.3.3. DATOS DE PRUEBA

45. Los datos utilizados durante el desarrollo o las pruebas de los sistemas de información deben ser ficticios, aunque deban poseer las mismas características que los datos reales. Estos datos se obtendrán mediante procesos de despersonalización de los datos de producción o mediante el uso de herramientas automatizadas de generación de datos.

46. Cualquier excepción a esta directriz debe ser convenientemente aprobada en la organización, debiendo en este caso aplicar sobre éstos las mismas medidas de protección que se aplican en los sistemas en explotación y a los datos reales, en especial todas aquellas que afecten o puedan afectar a los datos de carácter personal, tal y como marca la Ley Orgánica de Protección de Datos de Carácter Personal (LOPD).

5.3.4. GESTIÓN DEL SERVICIO EXTERNALIZADO

47. Los contratos con terceros para la externalización del desarrollo de software siempre deben ir acompañados de un acuerdo ajustado a la normativa y que garantice la confidencialidad de la información tratada durante la prestación del servicio. Tales acuerdos deben cubrir la

propiedad del código fuente y las restricciones de no difusión y no reutilización del mismo fuera del ámbito que la organización determine en cada caso.

48. En el caso de externalizar total o parcialmente el desarrollo de software la organización debe garantizar que los terceros con acceso al código están sometidos a criterios de seguridad aceptables en la organización, en especial en lo referente a la confidencialidad de la información. En este caso es necesario reforzar especialmente los aspectos relacionados con el cese de una persona concreta o del servicio en su conjunto; debe implantarse un procedimiento de altas y bajas de usuarios que contemple todos los pasos a seguir -en este caso, para la baja de usuarios de terceros- siempre que finaliza el servicio prestado por un tercero y que marque la obligatoriedad de revocar las claves de acceso utilizadas por el tercero, así como de cambiar convenientemente las contraseñas a las que dicho tercero pudiera tener acceso.

49. La calidad de los servicios de desarrollo externalizados debe ser evaluada y monitorizada de forma periódica, tanto desde el punto de vista de la seguridad como desde el punto de vista funcional. La organización debe determinar en cada caso concreto la periodicidad de esta revisión, que se ejecutará habitualmente mediante informes o reuniones en las que quede constancia de las decisiones tomadas, las acciones a realizar y las fechas y responsables de la ejecución de dichas acciones, tanto por parte de la organización como por parte del tercero; obviamente, estos informes serán la entrada de la siguiente revisión a efectuar.

5.4. CONTROL DE CAMBIOS

5.4.1. SEPARACIÓN DE ENTORNOS

50. La organización debe separar de forma adecuada los entornos de trabajo que sirven de soporte para el ciclo de vida de las aplicaciones; el número de entornos debe ser el necesario para garantizar la separación de las distintas funciones durante el proceso de desarrollo y el adecuado control para el paso a producción de las aplicaciones. Como un número mínimo de entornos debería considerarse un entorno de desarrollo, un entorno de preproducción y un entorno de producción. A estos entornos deben sumarse los necesarios para poder realizar las pruebas necesarias de integración sobre los distintos entornos de producción.

51. Como se detalla más adelante en esta misma guía, la organización debe establecer una separación clara de funciones y responsabilidades asociadas a cada entorno

5.4.2. SEGREGACIÓN DE TAREAS Y GESTIÓN DE CAMBIOS

52. La segregación de tareas es un aspecto clave en la seguridad de cualquier proyecto de desarrollo de software; es necesario disgregar las responsabilidades de los desarrolladores de las responsabilidades de los administradores de sistemas y BBDD. En ningún caso un desarrollador debe tener potestad para subir un desarrollo nuevo a entornos de preproducción o producción ni tampoco para lanzar una marcha atrás ante un error en dichos pasos. La organización debe establecer al menos los siguientes controles:

53. Segregación formal de tareas y responsabilidades dentro de cada proyecto de desarrollo –o en la organización en su conjunto-.

54. Definición de un procedimiento aprobado que contemple el paso a preproducción y a producción de los nuevos desarrollos del proyecto, y que garantice la segregación de tareas. Dicho procedimiento debe ser claro en la identificación de responsabilidades, planificaciones y marcha atrás ante problemas, de forma que se minimicen los problemas en el servicio que puede ofrecer un desarrollo en producción.

55. Auditoría periódica o continua -por ejemplo mediante controles técnicos- por parte del área de Seguridad del cumplimiento estricto de los puntos anteriores.

5.4.3. PROCESOS DE CAMBIO DE ESTADO

56. Los procesos de cambio de estado de una aplicación entre los distintos entornos deben estar adecuadamente controlados y ejecutados por el personal responsable en cada caso. En el apartado dedicado al despliegue seguro de la presente guía CCN-STIC se aborda este tema en profundidad, marcando las directrices de aplicación en cada caso.

5.4.4. NUEVOS DESARROLLOS

57. Tal y como se indica en diferentes puntos de la presente guía, un desarrollo seguro se consigue dando el mismo nivel de importancia a la seguridad que se le da a la funcionalidad; de esta forma, es necesario que el personal de seguridad de la organización intervenga de forma directa en los nuevos desarrollos y en los procesos de generación de nuevas aplicaciones desde sus estados más tempranos (análisis de requisitos), y valide, junto al resto de personal involucrado en el proceso de desarrollo, las diferentes fases que se producen en este proceso.

58. Es especialmente interesante que el personal de Seguridad participe de forma activa en la **seguridad del código** generado, ya que éste es habitualmente uno de los puntos vulnerables de las aplicaciones e introduce unos niveles de riesgo que deben ser convenientemente mitigados; así, la organización debe establecer un esquema de auditoría de código como algo intrínseco a cualquier desarrollo, nunca como algo excepcional. Dicho esquema debe contemplar que la auditoría sea lanzada en diferentes fases de la codificación (un mínimo de dos previamente a su lanzamiento) y por supuesto que se ejecute por parte de personal independiente del equipo desarrollador, para garantizar así una correcta segregación de funciones en la organización.

5.4.5. CAMBIOS EN EL SOFTWARE

59. Todos los cambios ejecutados sobre el código fuente de un desarrollo deben estar soportados para su control por una herramienta de control de versiones; adicionalmente, en el caso de desarrollos complejos compuestos por varios módulos, la organización debe evaluar la conveniencia de complementar el sistema de control de versiones con el uso de un sistema de control de configuración que relacione adecuadamente los distintos módulos.

60. Durante el proceso de desarrollo, todos los cambios realizados en el código fuente de la aplicación–en sus ficheros- deben ser versionados e identificados mediante almacenamiento de la revisión en el sistema de control de versiones; los cambios registrados deben estar siempre asociados al usuario que los realiza y estar marcados con la fecha de registro.

61. En cada nueva liberación de una versión se debe registrar un identificador para la versión de la aplicación, acompañada de un registro de los cambios realizados desde la versión anterior. En el caso de aplicaciones complejas compuestas por varios módulos (o para controlar la relación entre aplicaciones diferentes pero dependientes), la organización debe evaluar la conveniencia de generar un paquete versionado que agrupe los distintos componentes, paquete que debe estar compuesto por los diferentes módulos versionados individualmente. La definición de los paquetes y los módulos de los que está compuesto debe ser almacenada en un sistema de control de la configuración o en un documento que a su vez se recoja en el sistema de control de versiones del desarrollo.

5.5. PROCESOS DE MANTENIMIENTO

62. El mantenimiento de aplicaciones (soporte en cuanto a nuevas funcionalidades, corrección de errores, incremento de versiones...) debe contemplar los mismos requisitos de seguridad que el proceso de desarrollo en sí. Cada desarrollo o integración de una nueva funcionalidad, solicitud de usuario, mantenimiento correctivo, etc., debería ser analizada por un equipo de seguridad independiente del equipo desarrollador, que validará que la nueva característica no introduce riesgos no asumibles en la organización. No obstante, este planteamiento es demasiado estricto en la mayor parte de ocasiones, por lo que debe restringirse a mantenimiento de entornos de muy alta criticidad y el modelo debe ser explícitamente aprobado en la organización.

63. En cualquier caso estándar, para el mantenimiento evolutivo y correctivo debe ejecutarse, por parte del propio equipo desarrollador, lo que denominamnos un FCRA (*First Cut Risk Analysis*), es decir, una estimación aproximada del riesgo que puede suponer cualquier tarea relacionada con el mantenimiento (una solicitud de mejoras en la aplicación, una corrección, etc.). El personal del equipo desarrollador debe realizar este análisis informal y, en aquellos casos en los que el FCRA determine que se trata de un nivel de riesgo potencialmente relevante, entrará en juego el área de seguridad para determinar el riesgo introducido con mayor nivel de exactitud y, en caso de ser necesario, establecer junto al resto de personal relevante en la aplicación las salvaguardas necesarias para mitigar este riesgo.

64. Para complementar los análisis FCRA realizados por el equipo de desarrollo, y controlar que dichos análisis se realizan correctamente en tiempo y forma, la organización debe planificar y ejecutar, al menos anualmente, una auditoría de los FCRA realizados durante el periodo; dicho análisis debe ser ejecutado desde el área de Seguridad, detectando así errores o desviaciones con respecto a los parámetros de riesgo establecidos en los análisis FCRA que realiza el equipo de desarrollo.

6. ANÁLISIS DE CASOS DE ABUSO

65. Para crear software seguro, los analistas y desarrolladores deben anticiparse a los comportamientos anómalos que puede tener un sistema, tanto los que derivan de acciones automáticas como los que son consecuencia de acciones de las personas; se deben establecer mecanismos que protejan la información incluso para el caso en que el sistema no se comporte como es habitual, es decir, cuando tenga un comportamiento inesperado. En este sentido, el objetivo de los casos de abuso es analizar, documentar y decidir cómo, el sistema o aplicación, debe reaccionar ante un uso ilegítimo (intencionado o accidental) que pueda desembocar en una degradación de calidad del servicio prestado.

6.1. DEFINICIÓN DE CASOS DE ABUSO

66. Se define caso de abuso como la interacción entre un actor y el sistema que provoca un daño a otro actor, otro activo de la organización o el sistema en sí mismo. La organización debe definir y desarrollar casos de abuso sobre los desarrollos de forma equivalente a la definición de casos de uso; para ello debe partir de la definición de estos últimos y buscar sobre ellos las posibles acciones que podría realizar un usuario malintencionado: recuperar datos que no le corresponde visualizar, saturar el sistema, acceder a los servidores… Para identificar dichas acciones es necesario analizar el sistema desde el punto de vista del atacante: a partir del modelo de amenazas definido se identifican los posibles ataques y en los casos de abuso se relacionan los casos de uso con estas amenazas, concretando los patrones de ataque relevantes y describiendo cómo reacciona el sistema ante los mismos.

67. El área de Seguridad debe evaluar la capacidad del analista del sistema para definir los casos de abuso de forma correcta; en función de dicha evaluación nos encontraremos ante dos situaciones:

68. El analista o analistas del sistema son capaces de definir los casos abuso. Esta es la situación ideal, en la que un único perfil se encarga del análisis y definición de casos de uso y abuso, ya que la seguridad debe ser intrínseca al proceso de desarrollo y por lo tanto debe estar incorporada a éste desde el principio.

69. El analista o analistas del sistema no son capaces de definir los casos de abuso. Esta es la situación más habitual: el equipo implicado en el desarrollo suele estar en muchas ocasiones más focalizado en aspectos funcionales que en aspectos de seguridad. En esta situación la organización debe definir los casos de abuso mediante la siguiente aproximación:

70. El analista de sistemas o aplicaciones define los casos de uso y se encarga de que se implementen.

71. El analista de seguridad conoce los patrones de ataque más comunes y define los casos de abuso o desuso a partir de los casos de uso y de la tecnología en la que se desarrolle el producto.

72. El equipo ideal será el que aproveche las sinergias y los puntos fuertes de cada grupo, teniendo en cuenta además, que los diseñadores de un sistema conocen el sistema a fondo y son un apoyo imprescindible para los analistas de seguridad.

73. Los **actores** de los casos de abuso se pueden dividir en dos grupos básicos, en función de las intenciones y motivaciones de los mismos:

74. Usuarios lícitos de la aplicación que cometen errores.

75. Atacantes malintencionados: espionaje industrial, modificación y robo de datos, control de sistemas para la creación de botnets, envío de SPAM, uso de recursos para cracking de contraseñas…

76. Para **definir los casos de abuso**, como primera aproximación se deben crear los casos de desuso mediante un proceso de *brainstorming* focalizado, es decir, centrándose en el caso de uso correspondiente, entorno de aplicación y entorno del sistema (otros métodos más exhaustivos presentan un elevado consumo de recursos, difícilmente justificable en la mayor parte de ocasiones). La definición de casos de abuso debe tratar de responder las siguientes preguntas:

77. ¿Qué asunciones implícitas hay en el sistema o aplicación?

78. ¿Qué situaciones harían nuestras asunciones falsas?

79. ¿Qué tipos de ataques (patrones) usará un atacante?

80. ¿Se han revisado cuidadosamente las interfaces de usuario?

81. ¿Se han tenido en cuenta todos los posibles eventos que pueden acontecer en el ámbito del sistema o aplicación?

82. ¿Cómo distingue el sistema entre las entradas buenas y malas?

83. ¿Cómo distingue el sistema los errores de usuario?

84. ¿Qué vería un atacante que capturase el tráfico entre los sistemas?

85. ¿Podría modificar este tráfico de forma que fuera aceptado como legítimo?

86. ¿Se leen o escriben ficheros de/en un equipo potencialmente controlado por un atacante?
87. La seguridad es un proceso continuo y el riesgo de la aplicación o el sistema puede aumentar si se produce cualquier cambio en el entorno de la misma. Por lo tanto, se deben **revisar** los casos de abuso al menos cada vez que:
88. Se solicite un nuevo requerimiento.
89. Se solicite una nueva funcionalidad y con ello un caso de uso.
90. Se produzca un cambio relevante en el sistema o su entorno.
91. Aparezca una nueva vulnerabilidad o técnica de ataque que pueda afectar al sistema.
92. La organización debe dedicar el tiempo necesario a analizar de forma correcta las implicaciones del cambio a realizar y como, en consecuencia, la aplicación puede ser mal utilizada, ya sea de forma accidental o intencionada.

6.2. PLANTILLA DE DEFINICIÓN DE CASOS

93. Cada caso de abuso identificado debe ir siempre acompañado de un documento que detalle el escenario asociado al caso; dicho documento debe recoger al menos la siguiente información:

Código	<Código identificativo del caso>	
Nombre	<Nombre del caso>	
Versión	<Código de revisión del caso>	
Fecha	<Fecha de revisión del caso>	
Autores	<Personal implicado en el desarrollo del caso>	
Descripción	<Funcionamiento del caso indicando el objetivo del actor en la interacción>	
Actores	<Rol/es que participan en el caso>	
Precondiciones	<Condiciones previas a la ejecución del caso>	
Secuencia Normal	**Paso**	**Acción**
	1	<Primer paso de la interacción>
	2	<Segundo paso de la interacción>
	…	
	N	<Enésimo paso de la interacción>
Postcondiciones	<Condiciones posteriores a la ejecución del caso>	
Excepciones	**Paso**	**Acción**
	2	Si el sistema comprueba que las credenciales del usuario no son válidas aborta el proceso y muestra un mensaje
	3	Si el sistema comprueba que los parámetros enviados no son válidos aborta el proceso y muestra un mensaje
Comentarios		

6.3. CASOS PRÁCTICOS

6.3.1. ACCESO A BASE DE DATOS DE PRODUCCIÓN

94. El conocimiento del usuario y contraseña de acceso en modo escritura a la base de datos de producción por parte de desarrolladores o, peor aún, por parte de terceros, es un hecho que debe ser convenientemente controlado. Se deben imponer restricciones de acceso siguiendo la política del mínimo privilegio, estableciendo una categorización jerarquizada de aquellos usuarios que tengan acceso e identificando a qué datos lo tienen. También se debe implantar

un sistema de auditoría que permita obtener trazabilidad completa sobre aquellas modificaciones y consultas que, realizadas por los usuarios, violen las restricciones de acceso definidas en la política.

95. La definición del caso de abuso es la siguiente:

Código	CA_S_01	
Nombre	Acceder a la base de datos de producción	
Versión	1.0	
Fecha	15-09-2012	
Autores	Juan Pérez	
Descripción	Un usuario puede intentar acceder a la base de datos de producción del sistema para consultar la información allí almacenada o para modificar su contenido	
Actores	Usuario interno o externo	
Precondiciones	Usuario validado	
Secuencia Normal	**Paso**	**Acción**
	1	El técnico, que conoce datos de usuario y contraseña accede a la base de datos
	2	El sistema registra el acceso
	3	El sistema chequea de forma automática la coherencia de los datos almacenados en sus tablas
Postcondiciones		
Excepciones	**Paso**	**Acción**
	2	El técnico obtiene las credenciales de acceso a la base de datos
	3	Si falla el chequeo se deben generar alertas, comunicándolas a los responsables
Comentarios	Se debe implantar un sistema de auditoría que permita obtener trazabilidad completa sobre aquellas modificaciones y consultas que, realizadas por los usuarios, violen las restricciones de acceso definidas en la política. Si se detectan accesos no autorizados se deberá proceder a actualizar los datos de conexión y al control de a quién se comunican.	

6.3.2. MANIPULACIÓN DE FICHEROS

96. A la hora de cargar un fichero que la aplicación tome como entrada es posible que el usuario trate de cargar un fichero modificado de forma malintencionada, por ejemplo un ejecutable, un troyano, malware, etc.

97. La definición del caso de abuso es la siguiente:

Código	CA_S_02	
Nombre	Cargar ficheros con formato inválido	
Versión	1.0	
Fecha	15-09-2012	
Autores	Juan Pérez	
Descripción	Un usuario intenta adjuntar un fichero con formato no esperado para el servidor	
Actores	Usuario	
Precondiciones	Usuario validado o no validado	
Secuencia Normal	**Paso**	**Acción**
	1	El usuario adjunta un fichero con formato inválido

	2	El servidor comprueba el formato del documento
	3	El sistema procesa el documento validando, además, su contenido
	4	Se muestra un mensaje al usuario con el resultado de la ejecución del fichero
Postcondiciones		
Excepciones	**Paso**	**Acción**
	1	El usuario puede adjuntar cualquier tipo de documento si no se filtra su tipo en el buscador de ficheros
	2	Aún filtrando el tipo, el usuario podría incluir cualquier fichero cambiando la extensión de éste
	3	Además del tipo, el contenido del fichero podría ser manipulado por el usuario para provocar fallos
Comentarios	Se debe validar el tipo de fichero y su formato. Por ejemplo si la aplicación espera un archivo XML se debe validar su formato contra una hoja de formato XSD	

7. ANÁLISIS DE RIESGOS

7.1. OBJETO

98. El objeto del presente apartado es presentar un análisis de modelado de amenazas (*ThreatModelling*) para el entorno de desarrollo de aplicaciones web. El modelado de amenazas constituye esencialmente una técnica de análisis de riesgos orientada a aplicaciones y permite a los desarrolladores de los entornos analizar las amenazas que afectan a sus sistemas, permitiendo optimizar la asignación de recursos a aquellas partes del sistema que más lo necesitan y centrar los esfuerzos en los puntos débiles de la aplicación.

99. Entre las ventajas, debe destacarse que este enfoque permite gran flexibilidad y adquisición de una visión práctica del sistema, al no estar condicionado por un conjunto definido de amenazas y escenarios de siniestro, como sucede en los análisis de riesgos clásicos. No obstante, al igual que éstos, de un buen modelado de amenazas no se desprende necesariamente un buen *software*, debido a que del modelado se derivan pautas, errores, y procedimientos que mejoran las prácticas, pero que no implican obligatoriedad de aplicación.

100. Por último, cabe destacar que, aunque el modelo sigue un enfoque iterativo, y puede aplicarse en cualquier estadio del desarrollo de software, para una mayor efectividad el proceso de modelado de amenazas debe llevarse a cabo en las fases tempranas del ciclo de vida de desarrollo de software. Esto permite reducir la probabilidad de que los errores técnicos, operativos y de gestión se trasladen al producto final, convirtiéndose entonces en vulnerabilidades susceptibles de ser explotadas.

101. El catálogo de amenazas identificado a continuación es el punto de partida para el desarrollo de las recomendaciones de programación; de esta forma, el catálogo de buenas prácticas en el ámbito de la programación segura se convierte en un conjunto de medidas para paliar el riesgo identificado en el catálogo de amenazas.

7.2. IDENTIFICACIÓN DE AMENAZAS

102. Se propone a continuación una taxonomía generalista para clasificar las amenazas más relevantes en el ámbito del desarrollo de software, así como potenciales salvaguardas de

aplicación habitual; la organización debe particularizar a sus análisis esta taxonomía, ampliándola convenientemente en caso de considerarse necesario.

7.2.1. AUTENTICACIÓN

103. Podemos definir autenticación como el proceso por el cual el sistema verifica las credenciales del usuario para determinar que éste es quien dice ser, generalmente a través de parámetros como un nombre de usuario y una contraseña, aunque cada vez es más habitual el uso de certificados digitales. Dentro de esta clasificación las amenazas más habituales son las siguientes:

7.2.1.1. Suplantación de identidad

104. Esta amenaza se materializa cuando un usuario, tanto interno como externo, accede al sistema con las credenciales de acceso de otro usuario (usuario y contraseña, certificado digital…). Esta situación debe considerarse siempre un problema de seguridad, tanto si se trata de un robo de credenciales como si el usuario legítimo ha sido colaborador necesario en la suplantación, prestando sus credenciales (aquí nos encontramos ante un problema de trazabilidad y responsabilidad).

105. Como salvaguardas potenciales, la organización debe concienciar a los usuarios en la aplicación de buenas prácticas en la elección de contraseñas, e implantar sistemas de robustez para las claves (caducidad, control de sintaxis…) que impidan que el usuario utilice claves demasiado sencillas o predecibles. Así mismo, de manera periódica deben realizarse revisiones muestrales de los accesos a los sistemas en producción, controlando que los accesos son lógicos y correctos en duración y franja horaria.

7.2.1.2. Obtención de credenciales de usuarios

106. La materialización de esta amenaza, que consiste en el acceso ilegítimo a las credenciales de acceso de usuarios del sistema, está relacionada con el uso de herramientas de escucha lógica (*sniffers*), la malas prácticas en la gestión y elección de claves de acceso o con ataques más sofisticados como *man-in-the-middle*.

107. La organización debe implantar esquemas de caducidad y robustez para las claves, que mejoren la elección de contraseñas por parte de los usuarios y eviten la posibilidad de ataques de fuerza bruta. Este aspecto debe ser acompañado de la distribución de información sobre buenas prácticas en materia de seguridad, para una mayor efectividad. Además, debe desplegarse una correcta arquitectura de red donde ubicar el desarrollo, evitando que un atacante escuche información de nodos de la red.

7.2.1.3. Robo y manipulación de la sesión de otro usuario

108. El robo de la sesión puede considerarse similar a la suplantación de la identidad del usuario, ya que a todos los efectos para el sistema no hay diferencia entre el atacante y el usuario legítimo. A través de este tipo de ataques es posible no sólo obtener sino también modificar datos del posible perfil del usuario legítimo en tiempo de ejecución.

109. Para evitar la materialización de esta amenaza la aplicación debe liberar de la memoria todos los objetos de la sesión del navegador una vez que el usuario cierre su sesión, al igual que los posibles datos almacenados temporalmente en *cookies*. Asímismo, el navegador deberá

implementar mecanismos para evitar que los tokens/cookies puedan ser extraídos, visualizados o capturados; por ejemplo, a través del cifrado de las comunicaciones.

7.2.2. AUTORIZACIÓN

110. Se define autorización como el proceso mediante el cual el sistema valida al usuario y le permite (o no) acceder a la aplicación o sistema, concediéndole los permisos que le han sido otorgados por el administrador. Encontramos las siguientes amenazas asociadas al proceso:

7.2.2.1. Elevación de privilegios

111. Esta amenaza consiste en obtener de una manera ilegítima un nivel de privilegios superior al concedido al perfil del usuario.

112. Para evitar este tipo de acciones, es necesario que el uso de privilegios se controle y limite a los usuarios y casos necesarios. De la misma forma, los desarrolladores deben aplicar las buenas prácticas habituales a la selección y uso de contraseñas de los usuarios con privilegios, más restrictivas si cabe que las del resto de usuarios, y revisar de manera periódica los accesos de éstos.

113. Por otra parte, dado que la mayor parte de vulnerabilidades de los sistemas y aplicativos tienen como principal objetivo la obtención de privilegios, bien sea para la ejecución de órdenes únicas y específicas o para la realización de ataques de intrusión más elaborados, es imprescindible mantener actualizados todos los sistemas y aplicativos en los que se apoya el desarrollo con los parches de los fabricantes, evitando con ello que vulnerabilidades que son públicas sean aprovechadas por atacantes externos.

7.2.2.2. Acceso a operaciones privilegiadas

114. Esta amenaza es muy similar a la anterior, estando relacionada con la utilización esporádica –no permanente- de cierta funcionalidad o acceso lógico cuyo uso está limitado a perfiles de acceso privilegiado; de esta forma, como salvaguardas de aplicación encontramos las indicadas previamente.

7.2.2.3. Acceso no autorizado a entornos de desarrollo y preproducción

115. El acceso por parte de terceros no autorizados a entornos fuera del de producción implica que éstos puedan introducir modificaciones no controladas en el producto final, accedan a los datos almacenados y accedidos por la aplicación (razón por la que es importante que los sistemas de producción y preproducción no compartan bases de datos) o incluso se robe el código fuente.

116. Para evitar esta situación, los entornos de desarrollo y preproducción deben encontrarse correctamente bastionados desde el punto de vista de seguridad, según las directrices expuestas en las guías CCN-STIC correspondientes en cada caso y su uso debe quedar estrictamente restringido al personal autorizado (administradores de sistemas y bases de datos en preproducción y desarrolladores/administradores en desarrollo). Esto reforzará la integridad del código y evitará razonablemente que programas maliciosos dañen el código fuente de las aplicaciones desarrolladas en la organización.

7.2.3. VALIDACIÓN DE DATOS DE ENTRADA

117. La validación de datos de entrada es un aspecto de seguridad crítico y obligatorio en el desarrollo de cualquier aplicación, incluidos por supuesto los desarrollos web; en cualquier desarrollo debe definirse un módulo –o equivalente- encargado de validar los datos provenientes de fuentes no controladas (usuarios, otras aplicaciones, etc.). La explotación de amenazas relacionadas con la validación de datos de entrada suele estar por tanto vinculada a dicho módulo, así como al interfaz web de la aplicación desarrollada.

7.2.3.1. Envío de parámetros de entrada inválidos

118. Esta amenaza está relacionada con la modificación de valores de parámetros de entrada en la URL de una aplicación web; a través de este ataque, podría ser posible acceder a información o áreas restringidas de la aplicación para las cuales el usuario no estaría autorizado.

119. Para evitar la materialización de esta amenaza, el módulo de validación de entradas de la aplicación debe tratar tanto los parámetros que se introducen a través de los campos de los formularios web como los que se introducen a través de los parámetros de la URL, y gestionarlos correctamente junto al resto de información recibida.

7.2.3.2. Ataque de inyección SQL

120. Esta amenaza se basa en explotar la interacción con el usuario ofrecida por aplicaciones web (formularios de petición de datos, por ejemplo) tratando de hacer llegar a la base de datos consultas SQL manipuladas, que al no ser filtradas correctamente pueden tener efectos indeseados, como proporcionar información sobre el sistema o la estructura de la base de datos, o incluso permitir el acceso o borrado de la información almacenada.

121. Para evitar su materialización las entradas de los formularios y de los parámetros de URL deben ser validadas y correctamente gestionadas antes de su procesamiento por otros módulos del sistema o su almacenamiento en la base de datos.

7.2.3.3. Ataque de Cross-Site Scripting

122. Un ataque de *Cross-Site Scripting* (o XSS) consiste en la inclusión de código malicioso (Javascript, HTML,VBScript, etc.) en formularios web o parámetros de URLs con el objetivo de que sea almacenado por el servidor y posteriormente ejecutado en el navegador de cualquier usuario que visualice el contenido incrustado. Este tipo de ataques es habitual en aplicaciones en las que existe un alto nivel de interacción con el usuario, como foros, blogs o medios digitales.

123. Como salvaguardas básicas de aplicación para mitigar el riesgo asociado a este ataque encontramos las mismas a las que hemos hecho referencia en el caso de la inyección SQL.

7.2.3.4. Ataque de Buffer Overflow

124. Un ataque de desbordamiento de búfer (o *buffer overflow*) se basa en exceder intencionadamente el espacio reservado en memoria para el almacenamiento de un campo de datos con código malicioso, con el objetivo de modificar el contenido de la memoria provocando de este modo la sobreescritura de otras zonas de la memoria, lo que puede conllevar la ejecución no autorizada de código por parte del sistema.

125. Aunque estos ataques son más habituales contra aplicaciones de escritorio, en el caso de ciertos desarrollos web pueden ser empleados para causar una denegación de servicio,

ralentización o mal comportamiento de la aplicación. La organización debe vigilar el uso de funciones que limiten la cantidad de memoria que se copia entre variables frente a aquellas funciones que no controlan dicho aspecto (por ejemplo, strncat_s() frente a strcat() en C).

7.2.3.5. Validación JavaScript

126. Las aplicaciones web que delegan la validación de los datos de entrada únicamente en la ejecución de código Javascript son susceptibles de ser explotadas mediante la simple desactivación de este tipo de código en el navegador del usuario, permitiendo que al integrador del sistema le puedan llegar datos malformados de forma intencionada.

127. La validación total o parcial de los datos de entrada debe en todo momento residir en un módulo externo a cualquier elemento potencialmente controlable por el usuario (modificación, activación o desactivación).

7.2.3.6. Ataque de denegación de servicio

128. Este tipo de ataque busca, mediante la combinación y utilización de los ataques anteriores asociados a los datos de entrada, causar que el sistema deje de funcionar correctamente al procesar un dato de entrada con valor inesperado y por tanto no tratado en el código, lo que puede dar lugar a una acusada ralentización, a un malfuncionamiento o incluso a la detención completa del sistema; es necesario hacer hincapié en que no se trata de un DoS por consumo excesivo de recursos, sino por un procesado incorrecto de los datos.

129. Para mitigar el riesgo asociado a este ataque el equipo de desarrollo debe aplicar las salvaguardas indicadas en los puntos anteriores, relacionadas con la validación de datos de entrada.

7.2.4. CONFIGURACIÓN

130. En este punto se detallan amenazas relacionadas con la configuración general de un entorno (sistema o aplicación) que pueden provocar que éste sea accedido, manipulado o que funcione de forma no controlada o correcta.

7.2.4.1. Acceso a datos de configuración

131. Esta amenaza hace uso de diversos ataques o explota la existencia de fallos en los permisos de acceso a la configuración del sistema para obtener acceso ilegítimo a valores de configuración que pueden contener información muy útil para un atacante: direcciones IP, claves de acceso, parámetros de conexión a base de datos… Asociados a esta exposición de datos, son errores habituales la existencia de directorios en servidores web que muestran todo el contenido, el excedente de información ofrecida al usuario tras un fallo de la aplicación o un exceso de visibilidad de la aplicación desde otra red, dejando al descubierto componentes que no deberían ser accesibles.

132. Para hacer frente a este tipo de amenazas la organización debe planificar y ejecutar auditorías de seguridad periódicas contra los sistemas en producción. Adicionalmente, deben revisarse los permisos de ejecución de los componentes del sistema, las restricciones de visibilidad sobre los directorios web, y el tratamiento adecuado de errores como el código 404: cuando una página web no se encuentra disponible, los servidores web devuelven al cliente un código 404, y el servidor web (o servidor de aplicaciones) devuelve una página que dependiendo de la versión y aplicativo puede mostrar un mayor o menor nivel de

información sobre el servidor (versión, fabricante, módulos instalados, etc.). Tanto por aspectos estéticos y de usabilidad, como por seguridad, se debe configurar el servidor para que, cuando se solicita una página no existente, devuelva un mensaje previamente formateado que informe al usuario de que la petición es errónea.

7.2.4.2. Permisos incorrectos: defecto

133. Esta amenaza es consecuencia de una mala parametrización o instalación de la aplicación en la que pueden no haberse tenido en cuenta la necesidad de permisos de lectura o escritura para el acceso a directorios, bases de datos o ficheros, lo que provoca que el sistema tenga un comportamiento anómalo o simplemente no funcione.

134. Para mitigar esta situación los permisos de la aplicación en producción deben ser idénticos a los permisos en preproducción, tanto en el servidor web o en el servidor de aplicaciones como en cualquier otro componente involucrado.

7.2.4.3. Permisos incorrectos: exceso

135. Al contrario que la amenaza anterior, un exceso de permisos en la instalación y/o configuración del sistema incrementa el impacto de un ataque exitoso, al permitir que el atacante disponga de una mayor visibilidad y un privilegio superior al que habría sido estrictamente necesario para el correcto funcionamiento de la aplicación, aumentando además las posibilidades de que el aprovechamiento de una vulnerabilidad en la aplicación comprometa sistemas adicionales.

136. La organización debe planificar y ejecutar auditorías de seguridad periódicas del entorno en producción, verificando que a través de la aplicación no es posible acceder a recursos del sistema o del entorno que no deberían ser accesibles.

7.2.4.4. Sincronización incorrecta

137. En entornos en los que varios sistemas interoperan entre ellos y el factor temporal es relevante (intercambio de datos, validación cruzada, etc.), es imprescindible que todos ellos se mantengan sincronizados mediante protocolos de sincronización horaria como NTP. Adicionalmente, este aspecto cobra especial importancia en el caso de la investigación de incidentes de seguridad, en la que conocer el orden temporal de los eventos y transacciones es imprescindible para un correcto análisis.

138. La organización debe desplegar un servidor de tiempos NTP e instalar clientes en todos los sistemas que formen parte del entorno tecnológico, de forma que sistemas y aplicaciones se sincronicen correctamente y por tanto mantengan una hora común.

7.2.4.5. Actualizaciones no controladas

139. La aplicación de actualizaciones en cualquiera de los componentes involucrados en el funcionamiento de una aplicación (es decir, versión de software base, sistema operativo, base de datos, etc.), tanto motivadas por problemas de seguridad o nuevas funcionalidades, debe ser analizada y convenientemente probada en sistemas de preproducción de forma previa a su despliegue en producción, evitando así que soluciones a problemas que pueden carecer de relevancia para el entorno desemboquen en problemas reales.

140. La aplicación de parches, cambios de versión o modificaciones de cualquier tipo en los sistemas que dan soporte a una aplicación debe realizarse previo análisis de sus

implicaciones y en horario de mínimo impacto; en caso de que el cambio represente un cambio significativo, deberá contar con la autorización del responsable correspondiente en cada caso. La aplicación debe definir un procedimiento de parches y vulnerabilidades acorde a su política de seguridad corporativa y que garantice, en la medida de lo posible, la estabilidad de los entornos de producción donde se hospedan las aplicaciones que dan servicios relevantes.

7.2.5. DATOS SENSIBLES

141. Debe limitarse la presencia de datos sensibles que proporcionen información sobre la aplicación, el sistema o su entorno a los usuarios o potenciales atacantes y gestionar y aprobar de manera correcta cualquier información de carácter público que maneje la aplicación.

7.2.5.1. Datos en código fuente

142. En ciertos casos, por comodidad y rapidez es habitual que los desarrolladores incrusten datos de configuración de la aplicación en el código, tales como direcciones IP de servidores o motores de bases de datos, rutas de acceso a recursos o incluso contraseñas. En tales casos, si no existe un estricto control del proceso de paso a producción, este tipo de prácticas son susceptibles de mantenerse en la aplicación desplegada en producción, lo que expone la seguridad de la información manejada y dificulta la modificación de dichos parámetros en un futuro.

143. Para evitar estas situaciones deben utilizarse ficheros de configuración generales cuyas rutas y acceso no sea público bajo el interfaz web, no incrustando ningún valor relevante (usuarios, rutas, IPs, etc.) en el código. De manera periódica, el área de seguridad debe auditar el código fuente de la aplicación para confirmar que este tipo de datos no están presentes en dicho código.

7.2.5.2. Datos en comentarios

144. Durante el desarrollo de una aplicación web los desarrolladores pueden incluir información sensible en los comentarios del aplicativo, como direcciones IP, nombres de máquinas o usuarios o anotaciones sobre la arquitectura del sistema, y tanto en un código fuente que será posteriormente compilado, en JavaScript o en el propio código HTML; es en estos dos últimos casos en los que es particularmente peligrosa la presencia de tales comentarios, ya que facilitan a un potencial atacante información técnica que puede ser muy útil para él.

145. De manera periódica, al igual que en la revisión anterior, debe auditarse el código fuente de la aplicación para confirmar que no existen comentarios que puedan ofrecer información relevante.

7.2.5.3. Escucha lógica

146. Esta amenaza representa la posibilidad de que un dispositivo conectado a los sistemas de transmisión electrónica de la organización escuche y copie la información emitida y transmitida por una aplicación (con bases de datos, usuarios, etc.).

147. Para evitar este tipo de riesgos, deben desplegarse arquitecturas de red que dificulten al máximo la escucha pasiva. De la misma forma, el intercambio de información relevante entre el navegador y la aplicación o el sistema debe realizarse SIEMPRE mediante SSL

(utilizando el protocolo HTTPS) y en los casos en los que sea necesario (a evaluar en cada desarrollo concreto), la comunicación entre diferentes componentes del sistema debe realizarse utilizando, de la misma forma, cifrado de datos.

7.2.5.4. Manipulación o acceso

148. Esta amenaza representa la posible manipulación, publicación o visualización por parte de personal no autorizado de datos sensibles y la posibilidad de que éstos sean accedidos por un tercero, ya sea a través de un error del interfaz web, una vulnerabilidad del sistema o una pérdida de datos.

149. Para evitar esta situación los sistemas de *backend* deben residir en un direccionamiento diferente al del servidor web y su acceso desde el frontend o DMZ (en el caso de aplicaciones publicadas a Internet) estar limitado a los puertos que el servidor web requiera para un correcto funcionamiento. De este modo se evita que el servidor pueda ser accedido desde otra zona de red, dejando visibles puertos que puedan ser aprovechados por usuarios malintencionados para llevar a cabo ataques de denegación de servicio, intentos de intrusión u obtención de información privilegiada. La separación de la capa web y la de aplicaciones en entornos diferentes y con el tráfico entre ellos controlado es imprescindible en cualquier arquitectura de seguridad que se plantee; en caso de no poder aplicar esta separación, la organización debe implantar controles compensatorios con el objetivo de reforzar al máximo la seguridad de los sistemas, aplicaciones y especialmente de los datos tratados. Ejemplos de controles compensatorios pueden ser los entornos de detección de intrusos en host para la propia máquina donde se ubica la aplicación, firewalls en el sistema, control del tráfico de salida, etc.

7.2.5.5. Robo de código fuente

150. En muchas ocasiones los desarrolladores poseen un entorno propio de desarrollo y preproducción en el que despliegan la aplicación, con independencia de los entornos habilitados formalmente por la organización a tal efecto. En este caso, un usuario malintencionado que atacara a estos entornos satélites podría obtener acceso a todo o parte del código fuente de la aplicación, además de identificar vulnerabilidades en la misma que podrían explotarse en entornos de producción.

151. Como se ha indicado previamente, el acceso a los sistemas de desarrollo y preproducción que contengan código fuente sólo debe ser permitido al personal debidamente autorizado en la organización y deben desplegarse controles para verificar que este extremo se cumple correctamente. En aquellos casos en los que el desarrollo es realizado por terceros, debe firmarse un contrato de confidencialidad y exclusividad que asegure que el código no se publicita ni comparte con personas ajenas a la colaboración con la organización.

7.2.5.6. Información no aprobada

152. Esta amenaza está relacionada con la posibilidad de que el contenido que una aplicación web presenta no haya sido aprobado por personal autorizado, dando lugar a posibles errores, confusiones o malentendidos. No se trata de una vulnerabilidad estrictamente asociada al desarrollo de aplicaciones, sí que lo es a la gestión de contenidos.

153. La organización debe garantizar que cualquier contenido que se muestre en el interfaz de una aplicación web haya sido previamente validado y aprobado en la organización, manteniendo registro de ello, para evitar la presencia de información sensible o no correcta.

7.2.6. GESTIÓN DE EXCEPCIONES

154. Las aplicaciones web deben gestionar todos los errores y avisos de manera adecuada, evitando que el usuario obtenga información excesiva o sensible y la aplicación sufra efectos inesperados. Dentro de esta categoría es necesario destacar las siguientes amenazas:

7.2.6.1. Parada de la aplicación

155. Un tratamiento incorrecto de los errores de la aplicación puede dar lugar a un malfuncionamiento del sistema, provocando comportamientos inesperados, ralentización del sistema e incluso la parada de éste.

156. Para evitar esta situación todos los errores de la aplicación deben ser correctamente tratados, enmascarando el error al usuario cuando sea procedente o generando una página "amigable". Dichos errores deben generar internamente informes que deben ser revisados y gestionados por los responsables de la aplicación.

7.2.6.2. Obtención de datos relevantes de las excepciones

157. Un tratamiento incorrecto de los errores de la aplicación puede mostrar a un atacante información valiosa del sistema o de la aplicación que éste puede utilizar para llevar a cabo otros ataques. Entre dicha información se encuentran versiones del servidor de aplicaciones, de la base de datos, direccionamiento IP, rutas absolutas de acceso, etc.

158. En aquellos casos en los que la aplicación muestre una pantalla de error al usuario, ésta debe haber sido previamente definida, delimitando claramente la información que se le muestra; este comportamiento es configurable en el servidor web o en el servidor de aplicaciones.

7.2.7. AUDITORÍA Y REGISTRO

159. Para la detección de problemas de seguridad y funcionamiento, todos los componentes de la aplicación deben contar con un sistema de registro que permita establecer la trazabilidad de los errores o incidencias detectadas, y llevar a cabo un análisis fiable de su origen. En este ámbito identificamos las siguientes amenazas:

7.2.7.1. Eliminación y enmascaramiento de trazas

160. Esta amenaza reside en la probabilidad de que los registros y trazas de los diferentes componentes que forman parte del sistema sean eliminados de manera accidental o intencionada, ocultando posibles ataques, errores de la aplicación o simples revisiones del funcionamiento.

161. Para evitar esta situación, los registros del sistema (bases de datos, portal web, servidor de aplicaciones, registros de acceso) no deben ser accesibles más que por usuarios privilegiados. En los casos en los que sea posible, la organización debe evaluar la conveniencia de implantar un sistema central de registro (*syslog*) ajeno al entorno que hospeda la aplicación, al que se mande toda la información relevante para su revisión y salvado posterior de manera centralizada.

7.2.7.2. Inexistencia de registros

162. La ausencia de registros de actividad de los diferentes componentes del sistema supone un gran inconveniente en numerosos aspectos: revisión de posibles errores de funcionamiento,

potenciales ataques o intentos de uso fraudulento, detección de sobrecarga o necesidad de dimensionamiento de la aplicación, análisis de problemas de uso del sistema, etc.

163. Para evitar estos inconvenientes deben configurarse los sistemas para que registren las acciones más relevantes del sistema, hasta un nivel en el que esto no interfiera de manera significativa en el rendimiento del entorno y el volumen de información generada sea manejable (por ejemplo, niveles de *debug* en la base de datos). Al igual que en el caso anterior, la organización debe evaluar la puesta en marcha de un sistema de registro centralizado de logs (*syslog*) y definir los mecanismos de análisis posterior de los registros.

7.2.8. CONTROL DEL DESPLIEGUE

164. Es imprescindible que el paso de la aplicación por los diferentes estadios (desarrollo, preproducción o espejo, y producción) esté sujeto a un procedimiento de gestión de cambios que determine autorizaciones formales y requisitos, tales como personal responsable, procedimientos de marcha atrás u horario de realización. La organización debe definir y aplicar dicho procedimiento, así como velar por su aplicación en cada uno de los entornos implicados en la gestión de cambios.

7.2.8.1. Validación incorrecta en el paso a preproducción

165. La ausencia de un proceso de validación formal por parte de un responsable de la organización para el paso de la aplicación de desarrollo a preproducción implica un riesgo potencial de presencia de errores en este entorno procedentes de desarrollo. Además de la propagación y persistencia de errores, esta situación también puede provocar la liberación de funcionalidad no aprobada formalmente.

166. Debe desarrollarse e implantarse un procedimiento de gestión de cambios, que contemple la aprobación formal por parte de un responsable en cualquier paso de desarrollo a preproducción y determine claramente las funciones y responsabilidades de los actores implicados. Las modificaciones directamente sobre el sistema de preproducción deben estar prohibidas y de manera periódica deben realizarse auditorías de seguridad y revisiones del código fuente modificado o introducido en los sistemas de preproducción.

7.2.8.2. Validación incorrecta en el paso a producción

167. La ausencia de un proceso de validación formal por parte de un responsable de la organización para el paso de la aplicación de desarrollo a preproducción implica un riesgo potencial de presencia de errores en el entorno final, accedido por el usuario de la aplicación. Además de la propagación y persistencia de errores, esta situación también puede provocar la liberación de funcionalidad no aprobada formalmente.

168. En la línea de la salvaguarda anterior, debe implantarse un procedimiento de gestión de cambios que requiera una aprobación formal en el paso de preproducción a producción, que contemple horarios, tiempos estimados, requisitos previos y posteriores, procedimientos de marcha atrás y determine claramente las funciones y responsabilidades de los actores implicados. Las modificaciones directamente sobre el sistema de producción deben estar prohibidas y de manera periódica deben realizarse auditorías de seguridad de este entorno.

7.2.8.3. Errores por discrepancia entre entornos

169. Las diferencias de versiones o configuración entre los distintos componentes del sistema utilizados en preproducción y producción, como por ejemplo motores de bases de datos, sistema operativo o servidores de aplicaciones, pueden provocar que la versión final de la aplicación exhiba un comportamiento diferente a la versión en preproducción, tanto desde el punto de vista estético y de funcionalidad como desde el punto de vista de seguridad.

170. Para evitar estos errores, los componentes de ambos entornos deben ser iguales en versiones, configuración y actualizaciones y cualquier modificación del entorno de preproducción debe ser trasladada, tras la verificación del correcto funcionamiento, al sistema en producción siguiendo procedimientos definidos en la organización.

7.2.8.4. Acceso de desarrolladores a entornos de producción

171. El acceso de personal ajeno a la administración del sistema a los sistemas de producción o preproducción puede condicionar la trazabilidad del entorno y permitir a los desarrolladores acceder a entornos, utilidades y herramientas restringido al personal de sistemas, facilitando entre otras situaciones los cambios no controlados en entornos de preproducción y producción, con las implicaciones de funcionalidad y seguridad asociadas.

172. Salvo razones debidamente identificadas, motivadas y aprobadas en la organización (habitualmente, problemas graves que requieren un cambio urgente de la aplicación), ningún desarrollador debe tener acceso a los sistemas en producción o preproducción, ni siquiera para el paso de los desarrollos a este último. Los pasos a preproducción y producción deben realizarse por parte de personal del área de sistemas o equivalente con la asistencia documental, presencial y/o telefónica del personal del equipo de desarrollo. Para aquellas excepciones puntuales, el procedimiento de gestión de cambios anteriormente indicado debe contemplar las circunstancias y necesidades que requieran que se solicite y autorice el acceso a preproducción y producción por parte de personal de desarrollo. La organización debe vigilar diariamente que no existen accesos no justificados a entornos de preproducción o producción.

7.2.8.5. Procedimientos de marcha atrás no definidos

173. La ausencia de un procedimiento de marcha atrás en el despliegue de una aplicación puede poner en riesgo el funcionamiento y seguridad de la misma y de los datos que ésta trata.

174. El paso a producción de una aplicación debe contar siempre con un mecanismo de vuelta al estado anterior en caso de existir problemas con el despliegue. Para todas aquellas modificaciones del sistema en producción que requieran de una parada del servicio o una modificación mayor de la aplicación, el procedimiento de gestión de cambios debe contemplar la obligatoriedad de que dicho mecanismo esté correctamente identificado, probado y autorizado en la organización.

7.2.8.6. Herramientas de desarrollo en entornos de producción

175. La existencia de herramientas de desarrollo en entornos de producción, como *debuggers*, compiladores o entornos completos de desarrollo puede provocar la utilización de éstas para la compilación de todo tipo de software malicioso por parte de un atacante, además de favorecer en ocasiones la modificación de la aplicación en producción por parte de los desarrolladores.

176. Los entornos finales de la aplicación deben contar con el mínimo número de herramientas necesario para el funcionamiento de la aplicación, evitando las amenazas indicadas y evitando además potenciales problemas de librerías o compatibilidad de la aplicación.

7.2.9. CONTROL DE VERSIONES

177. El código fuente de la aplicación, los componentes no compilables, las bases de datos y cualquier elemento significativo que forme parte de la aplicación debe ser salvado correctamente mediante copias de seguridad, asegurando su disponibilidad en caso de ser necesario.

7.2.9.1. Indisponibilidad de las copias de seguridad

178. La indisponibilidad de copias de seguridad, ya sea por robo, pérdida o mala conservación, puede poner en riesgo la continuidad del aplicativo en los casos más graves o dificultar la recuperación de información borrada o perdida.

179. Las copias de seguridad periódicas de los elementos del sistema deben planificarse convenientemente y sus medios asociados almacenarse en un lugar de acceso restringido. En aquellos casos en los que se haga uso de robots para la gestión integral de cintas como mínimo semanalmente deben sacarse copias completas del robot a una ubicación externa, preferentemente una caja fuerte ignífuga, para evitar que en caso de desastre en la ubicación donde se aloja el robot de cintas se pierdan todas ellas (este punto es además requerido en algunas situaciones por la normativa de Protección de Datos Personales). Por último, la organización debe planificar pruebas periódicas de recuperación para verificar que las copias son correctas y completas.

7.2.9.2. Imposibilidad de recuperar versiones anteriores del sistema

180. La ausencia de herramientas de control de versiones dificulta notablemente la recuperación de versiones previas de los desarrollos, impidiendo comparar cambios y regresar a estados consistentes de la aplicación si fuese necesario.

181. Debe implantarse una herramienta de control de versiones en la organización, que facilite la gestión del código fuente y permita manejar de manera eficiente versiones anteriores del código de la aplicación.

8. REQUISITOS LEGALES DE LAS APLICACIONES

182. Al abordar el desarrollo de cualquier aplicación, el equipo de trabajo debe identificar la legislación aplicable a los datos tratados, garantizando que el diseño, implementación y operación cumplirá dicha normativa, en especial en lo referente al tratamiento de datos de carácter personal. Adicionalmente, la organización debe establecer una revisión anual de la normativa aplicable con el objeto de identificar nuevos requisitos de seguridad o modificaciones de los existentes que puedan ser de aplicación en el ámbito del desarrollo o de los entornos ya operativos.

8.1. ASPECTOS DE ÍNDOLE ORGANIZATIVA Y DOCUMENTAL

8.1.1. MEDIDAS DE SEGURIDAD

183. En el ámbito de la protección de datos de carácter personal la organización, en el cumplimiento de la legislación vigente, debe garantizar que la aplicación a desarrollar cumple con las medidas de seguridad correspondientes en cada caso, en función del tipo de dato tratado. En concreto, si la aplicación a desarrollar gestiona datos relativos a ideología, afiliación sindical, religión, creencias, origen racial, salud o vida sexual, el art. 81.5 del RDLOPD determina que dichos datos requieren medidas de seguridad de nivel alto, especialmente restrictivas, salvo excepciones identificadas en la legislación vigente.

8.1.2. PRESTADORES DE SERVICIOS

184. Es práctica habitual en las Administraciones Públicas que existan empresas externas que prestan servicios de asistencia técnica tanto para el desarrollo de las aplicaciones corporativas, bajo la supervisión y control de los equipos de la organización, como para su posterior soporte. En esta situación la organización, como responsable del tratamiento de los datos, deberá velar por que los encargados del tratamiento –empresas que le prestan servicios que afectan a datos personales responsabilidad de la organización- reúnan las condiciones para garantizar el cumplimiento del Reglamento de Desarrollo de la LOPD en el ámbito de los servicios prestados (art. 20.3 del RDLOPD). Es decir, la organización debe requerir al prestador del servicio evidencias que acrediten que está al corriente de sus obligaciones en relación con el cumplimiento de la LOPD y el RDLOPD. En la práctica esto se traduce en que el prestador del servicio facilite su documento de seguridad como encargado del tratamiento -novedad introducida por el RDLOPD en sus art. 82.2 y 88.5 - o el informe de su última auditoría bienal de cumplimiento.

185. Por otro lado, estos prestadores de servicios de desarrollo no podrán subcontratar con un tercero la realización de trabajos que les haya encomendado la organización, salvo que medie el conocimiento previo y la autorización expresa de la organización como responsable del tratamiento.

8.1.3. OBLIGACIONES DEL PERSONAL

186. Las funciones y obligaciones de los usuarios con acceso a los datos personales gestionados por las aplicaciones corporativas, incluidas las de desarrollo propio, deben estar definidas y documentadas en el Documento de Seguridad de la organización, en cumplimiento del artículo 89 del RDLOPD. Adicionalmente, este artículo del RDLOPD traslada formalmente al responsable del fichero la obligación de **formar y concienciar a los usuarios** de sus aplicaciones en relación con las medidas de seguridad que deben seguir y cumplir en el desempeño de sus funciones

8.2. ASPECTOS TÉCNICOS RELACIONADOS CON LA PROTECCIÓN DE DATOS PERSONALES

8.2.1. IDENTIFICACIÓN Y AUTENTICACIÓN

187. Cualquier desarrollo debe garantizar el cumplimiento de los siguientes aspectos regulados por el RDLOPD en relación con el control de acceso lógico a los datos:

188. Los usuarios tendrán acceso únicamente a aquellos recursos que precisen para el desarrollo de sus funciones.

189. Debe existir una relación actualizada de usuarios con el perfil de acceso de cada uno de ellos.

190. Se deben establecer mecanismos para evitar que un usuario pueda acceder a recursos con derechos distintos de los autorizados para él.

191. Se debe garantizar la correcta identificación y autenticación de los usuarios; la aplicación no debe permitir la utilización de usuarios genéricos que sean compartidos por varias personas.

192. Si la autenticación se basa en un modelo de contraseñas, debe existir un procedimiento de asignación, distribución y almacenamiento que garantice su confidencialidad e integridad; las contraseñas deben estar almacenadas de forma cifrada.

193. La aplicación debe cumplir la política corporativa para criterios de robustez y tiempos de vida de las claves.

194. Debe existir un mecanismo que limite la posibilidad de intentar reiteradamente el acceso no autorizado a la aplicación.

8.2.2. GESTIÓN DE SOPORTES Y DOCUMENTOS

195. Los soportes que permitan el paso de información de datos de carácter personal o documentos relacionados con el proyecto (documentos de análisis, código fuente, etc.) deben ser inventariados y su salida debe ser registrada y únicamente permitida al personal autorizado. Cualquier soporte relacionado con cualquier fase del desarrollo de una nueva aplicación debe estar inventariado y bajo el control de los directores del proyecto.

196. Si no está previamente definida en la organización se debe elaborar y aplicar una normativa de uso de dispositivos móviles, que deberá ser trasladada formalmente tanto al personal interno como al personal externo de las asistencias técnicas que participen en el desarrollo de los nuevos proyectos.

197. Asimismo, la organización debe evaluar la conveniencia de implantar un esquema de clasificación de la información alineado con la política corporativa y asociado a la metodología de desarrollo de software de la organización, que clasifique los diferentes tipos de documentos y soportes que se generan en un proyecto-tipo de desarrollo con objeto de determinar los niveles de seguridad asociados a los mismos en cada caso.

8.2.3. COPIAS DE RESPALDO Y RECUPERACIÓN

198. Cualquier nuevo desarrollo que llega al entorno de producción debe estar incluido en la política de copias de seguridad de la organización; por este motivo, el equipo de desarrollo facilitará a los equipos de explotación la información necesaria relativa a las unidades de información a salvaguardar (configuraciones, bases de datos…) en el paso a preproducción de la aplicación.

199. El diseño de la política de copias de seguridad de la información tratada por cualquier nueva aplicación que se vaya a desarrollar en la organización debe contemplar el análisis de la conveniencia de que una copia de la base de datos en explotación resida en una ubicación diferente a la que residen los servidores de base de datos. Si la información tratada mediante la nueva aplicación va a contener datos personales de nivel alto, esta copia alojada en una ubicación alternativa es una exigencia del RDLOPD (artículo 102), que además exige que la misma esté cifrada (artículo 101.2).

200. Con el objeto de comprobar la corrección de los procedimientos de copia y restauración definidos sobre la nueva aplicación, con la periodicidad definida en la política corporativa se deben realizar pruebas de restauración de los datos salvaguardados.

9. DIRECTRICES DE DISEÑO

201. Para realizar el diseño de una aplicación segura se deben considerar las siguientes premisas:

202. Confidencialidad: permitir acceso únicamente a los datos a los que el usuario está autorizado.

203. Integridad: asegurar que los datos no se falsifican o alteran por usuarios no autorizados.

204. Disponibilidad: asegurar que los sistemas y datos están disponibles para los usuarios autorizados cuando lo necesiten.

205. El diseño de cualquier aplicación debe tener en cuenta todas y cada una de las premisas anteriores para garantizar la seguridad de esta aplicación desde todos sus puntos de vista.

9.1. ARQUITECTURAS DE SEGURIDAD

206. La arquitectura de seguridad se basa en un conjunto de características y requisitos concretos, controles, procesos seguros y una cultura de seguridad por defecto en la organización. Cuando se inicia el diseño de una aplicación o se rediseña una ya existente, se debe considerar cada característica funcional requerida analizando los siguientes aspectos de la misma:

207. Procesos que rodean a la característica funcional para determinar que sean lo más seguros posible.

208. Casos de abuso asociados a la característica funcional.

209. Activación condicional de la característica, por ejemplo a ciertos perfiles.

210. Identificación y establecimiento de límites u opciones que puedan reducir el riesgo asociado a la característica.

211. En el ámbito de la seguridad en el desarrollo de aplicaciones –o en cualquier otro ámbito- siempre debemos considerar que la seguridad es un proceso de larga vida y no algo implementado de manera puntual.

9.2. SUPERFICIE DE ATAQUE

212. La superficie atacable de una aplicación son todos los interfaces de la misma publicados al exterior para la interconexión de sistemas (RMI, WS...) o acceso de usuarios (web...). A priori todos los interfaces publicados pueden ser atacados, por lo que si minimizamos los interfaces publicando únicamente los necesarios para cumplir con la funcionalidad establecida en la aplicación reduciremos los posibles ataques desde el exterior.

213. La metodología de publicación de los interfaces accesibles desde el exterior debe tener en cuenta este punto evitando publicar funcionalidades similares y dependientes entre ellas y eliminar interfaces que queden obsoletos para no ir ampliando la superficie atacable a medida que crezca nuestra aplicación.

9.3. SEGURIDAD POR DEFECTO

214. Cualquier aplicación puede ser objeto de ataques de muy diversa índole, tanto tecnológicos como no tecnológicos; a la hora de diseñar una aplicación, el equipo de desarrollo debe adoptar un rol de atacante activo de la aplicación –para lo que habitualmente se apoyará en un equipo de seguridad- identificando posibles vulnerabilidades o debilidades en la gestión y configuración de seguridad. Este trabajo debe ayudar al equipo a establecer unos patrones

mínimos por defecto en la seguridad de las aplicaciones, aportando por defecto un nivel de securización de partida que luego se podrá reforzar.

9.4. PRINCIPIO DEL MÍNIMO PRIVILEGIO

215. Si un atacante consigue acceder a la aplicación utilizando los permisos de algún usuario o a través del aprovechamiento de vulnerabilidades en algún módulo concreto, podría obtener el control total si no se tienen establecidos permisos específicos para cada rol del entorno. Para minimizar el posible daño que un atacante puede causar es necesario que a la hora de establecer permisos a usuarios o roles de la aplicación, éstos se asignen en función de lo que es estrictamente acceder o ejecutar o, dicho de otra forma, que tales permisos tengan la mínima cantidad de privilegios necesarios para realizar la funcionalidad requerida.

216. El privilegio de acceso a bases de datos es habitualmente un elemento crítico que muestra esta situación; cuando la aplicación web accede a la base de datos, en muchas ocasiones lo hace mediante un único usuario para la aplicación con acceso de lectura y escritura a toda la base de datos. La organización debe evaluar la necesidad de que cada módulo de la aplicación acceda con usuarios diferentes y con distintos privilegios a dicha base de datos, dependiendo de la funcionalidad implementada.

9.5. PRINCIPIO DE DEFENSA EN PROFUNDIDAD

217. Implementar un único punto donde se valida la seguridad puede tener un rendimiento muy alto pero puede abrir el sistema a posibles ataques superando únicamente este punto. Una defensa en profundidad pasa por establecer distintos puntos de control de seguridad en las distintas capas de una aplicación, de forma que se dificulte a un potencial atacante el acceso a la aplicación en su conjunto.

9.6. GESTIÓN DE FALLOS

218. Habitualmente, ante un fallo en la aplicación, el módulo encargado en cada caso del tratamiento de errores transmite al usuario información detallada de la incidencia. Aunque esto es deseable de cara al diagnóstico del problema, para un equipo de mantenimiento de la aplicación, es muy perjudicial desde el punto de vista de la seguridad de la información tratada, ya que proporcionamos al usuario o a un potencial atacante datos útiles para causar un impacto en el entorno.

219. Desde el punto de vista de seguridad, la aplicación debe proporcionar la mínima información que pueda servir para localizar en el fichero interno de *log* (que es donde se debe volcar la información de depuración) de la aplicación las trazas del fallo que un usuario pueda reportar; **jamás** se debe mostrar esta información detallada fuera del registro interno (es decir, en el navegador del usuario), sino una referencia mínima para reportar el problema localizado –por ejemplo, mediante un código numérico-.

9.7. CONFIANZA EN TERCEROS

220. No debemos confiar en los interfaces publicados desde sistemas de cuya seguridad no tengamos ningún control y no se disponga de una adecuada visión, ya que el equipo desconocerá cómo se han desarrollado y qué riesgos introducen en la aplicación y en los datos que maneja. Todos los sistemas externos deben ser tratados de un modo similar que aplique los controles necesarios para verificar que la información enviada o recibida por dichos sistemas es tratada de forma conveniente, estableciendo en especial mecanismos de validación de los datos recibidos.

9.8. SEGURIDAD A TRAVÉS DE LA OSCURIDAD

221. En ocasiones se presupone que la ocultación del código de la aplicación lleva implícita la seguridad de ésta; dicha presunción es incorrecta: hoy en día existen multitud de metodologías para realizar ingeniería inversa de aplicaciones desplegadas y poder acceder a los detalles de las mismas. El equipo de desarrollo no debe confiar en que su código no vaya a ser accedido por un tercero y por lo tanto no debe considerar la ocultación, ofuscación o cualquier otra técnica para enmascarar datos como una salvaguarda de seguridad aceptable.

222. El equipo debe considerar que el código generado puede ser accesible por un tercero y actuar en consecuencia desde el punto de vista de seguridad, modularizando sistemas y delegando funciones a otras plataformas.

9.9. SIMPLICIDAD

223. Realizar complejos procesos que proporcionen controles de seguridad y validación implícitos puede llevar a cometer graves errores en la implementación, provocando incidencias de seguridad. Si por el contrario el mismo proceso se divide en varios módulos independientes, más simples que el global, se implementará de forma mucho más sencilla y se disminuirán considerablemente los posibles errores en la implementación. En el desarrollo de una aplicación, el equipo debe optar, siempre que sea posible, por aspectos de modularidad, escalabilidad y sencillez que refuercen la seguridad global de dicha aplicación.

9.10. CORRECCIÓN DE PROBLEMAS

224. Cuando se detecta un fallo de seguridad en la implementación de un sistema pueden producirse problemas de seguridad en los entornos dependientes de éste si el proceso de corrección y actualización no se realiza correctamente.

225. La organización debe establecer una metodología de implantación de mejoras de seguridad en los desarrollos, considerando las interdependencias funcionales y de seguridad entre entornos y analizando y probando previamente el impacto de cualquier mejora en las aplicaciones.

10. DIRECTRICES DE DESPLIEGUE

226. La organización debe definir, aprobar e implantar un procedimiento operativo para las tareas de despliegue de nuevas *releases* o funcionalidades de una aplicación; debe considerarse al menos la siguiente **estructura organizativa**:

227. Equipo de desarrollo. Responsable del desarrollo de la nueva funcionalidad (entendiendo desarrollo por análisis, programación, pruebas, documentación…) o *release* de la aplicación a desplegar.

228. Equipo de integración. En aquellos casos en los que se planifique una actualización especialmente crítica (por su envergadura, por los datos tratados, por su complejidad técnica…) la organización debe evaluar la conveniencia de definir un equipo de integración, un grupo de trabajo mixto formado por personal de desarrollo (quienes conocen la aplicación internamente) y por personal de explotación (quienes conocen el entorno en el que se va a desplegar), cuyo objetivo es la planificación detallada del paso a preproducción analizando la repercusión que el nuevo desarrollo puede tener en el entorno real y determinando las salvaguardas necesarias para minimizar los riesgos relacionados con errores en este entorno.

229. Equipo de explotación. Responsable de la explotación de la infraestructura TIC que da soporte a las aplicaciones, este equipo debe garantizar que tales elementos funcionan de forma adecuada a los requisitos de servicio de la organización. Generalmente, en entornos complejos, se definen al menos los siguientes grupos dentro del equipo de explotación:

230. Sistemas.

231. Comunicaciones.

232. Bases de datos.

233. Equipo de seguridad. Responsable de validar que los nuevos despliegues –así como la plataforma en su conjunto- se ajustan a los estándares, políticas, buenas prácticas… definidos en la organización, mediante metodología de auditoría y control.

234. Cada uno de los equipos anteriores debe tener identificado un **responsable** funcional del mismo, de forma que este perfil sea el responsable último de las acciones de su equipo en los cambios de estado dentro de la organización; obviamente, aunque es posible delegar la función en casos concretos (por ejemplo, nominando a un responsable concreto para un determinado cambio, diferente del responsable del equipo), no se debe permitir la delegación de responsabilidad en ningún caso.

235. Todos los despliegues deben **llevarse a cabo** por parte del personal del equipo de explotación (generalmente del grupo de Sistemas, aunque puede ser necesario que participe personal de otros grupos). En cada despliegue realizado este equipo de Sistemas debe disponer de soporte (presencial o remoto) de al menos una persona del equipo de desarrollo, una persona del equipo de comunicaciones y una persona del equipo de bases de datos; el primero de ellos (desarrollo) es obligatorio en cualquier caso, mientras que el soporte del resto está condicionado a la relevancia, complejidad o implicaciones del despliegue en cada caso, a evaluar por la organización. Adicionalmente, para aquellos despliegues en los que se haya definido un equipo de integración –es decir, aquellas especialmente relevantes-, se debe disponer de soporte de personal de este grupo.

236. Tal y como se ha indicado con anterioridad, la organización debe definir un **procedimiento** estandarizado y formal de cambios de estado para las aplicaciones (pasos a preproducción y pasos a producción), de forma que el conocimiento global no dependa del personal técnico en cada caso y que garantice la trazabilidad de los cambios realizados, con objeto de identificar problemas y permitir la marcha atrás de forma ágil en aquellos casos en los que se considere necesario. Este procedimiento debe cubrir al menos los siguientes puntos:

237. Clasificación de los cambios. Establecimiento de una clasificación de los posibles cambios a realizar en la organización, en función del riesgo que implican para ésta y definiendo en qué casos es necesaria aprobación, en qué casos es necesaria la definición de un equipo de integración, etc. Una primera aproximación puede ser –a evaluar en cada organización- la identificación de los cambios menores, mayores y de alto riesgo, para proceder convenientemente en cada uno de los casos.

238. Flujo del cambio de estado. Para los cambios de estado que así lo requieran (por defecto, todos salvo los cambios menores), es necesario definir de forma concisa el flujo que debe seguirse para realizar el cambio: peticiones de cambio –indicando los motivos, riesgos estimados, beneficios, prioridades…-, aprobaciones y aprobadores, identificación de personal involucrado, responsables, evidencias, marcha atrás…

239. Informe del cambio. Informe a realizar por el responsable del cambio en el que se indique cualquier circunstancia destacable en la ejecución del mismo.

240. Auditoría y control. Elementos que desde el equipo de Seguridad se deben establecer para garantizar que los cambios en las aplicaciones de la organización se realizan conforme al procedimiento, permitiendo así detectar desviaciones que puedan comprometer la seguridad de la organización.

241. En la fase de despliegue, es necesario considerar los siguientes aspectos en relación al **servidor de aplicaciones** sobre el que se despliega un desarrollo determinado, con el fin de no introducir vulnerabilidades o debilidades asociadas a este entorno:

242. Configuraciones por defecto. Muchos de los servidores de aplicaciones del mercado, como pueden ser Apache Tomcat u Oracle Application Server, presentan configuraciones que sin causar por sí mismas grandes problemas de seguridad, sí que pueden proporcionar a un potencial atacante información muy valiosa de cara a realizar una acción dirigida contra la organización; así, desde las típicas configuraciones que presentan un mensaje de bienvenida, hasta parámetros internos del servidor de aplicaciones, como el *DirectoryIndexing*, deben ser revisadas para garantizar que su configuración es la correcta y que cualquier información que pueda ser útil a un atacante es eliminada

243. Ejemplos y páginas de prueba. De la misma forma que en el caso anterior, muchos servidores de aplicaciones presentan al ser instalados diferentes páginas de ejemplo de uso, *webapps* de prueba, etc. que, además de los peligros anteriores –fuga de información potencialmente útil para un atacante- presentan otros peligros mucho mayores: los *bugs* o problemas de seguridad que estas aplicaciones de prueba o ejemplos de funcionamiento puedan presentar, desde la obtención de variables del sistema hasta la posibilidad de enviar ficheros internos a través de un interfaz web, pasando por problemas de inyección SQL. Es especialmente relevante considerar aquellas aplicaciones de prueba que permiten la introducción de datos por parte del usuario, ya que en la mayor parte de casos no son de utilidad real y permiten ser utilizadas por un atacante en beneficio propio.

244. Mensajes de error. Otro aspecto a considerar a la hora de realizar un despliegue son los mensajes de error que, bien por parte del servidor de aplicaciones, bien por parte de la propia aplicación, se presentan al usuario. Típicamente, estos mensajes proporcionan demasiada información que, como en el primer caso, por sí misma no representa un peligro directo pero que puede ser utilizada para reforzar un ataque a través de otros mecanismos. Desde el error habitual HTTP 404 del servidor web hasta los errores de las aplicaciones que se ejecutan sobre ese mismo servidor, estos mensajes deben ser analizados para garantizar que no proporcionan más información de la estrictamente necesaria.

245. Para poder minimizar estos problemas y muchos otros, la organización debe aplicar en primera instancia las guías CCN-STIC de bastionado de servidores web o de aplicaciones necesarias en cada caso, en función de la tecnología concreta que está utilizando. Adicionalmente, se debe ejecutar un análisis de seguridad exhaustivo sobre las plataformas de preproducción y producción, tanto hacia el servidor como hacia la aplicación web. En el caso del análisis del servidor, los analizadores automáticos habituales (Nikto, Nessus, WMap…) son de gran ayuda para detectar este tipo de errores, así como la presencia de directorios que puedan contener información sensible, fallos de configuración, aplicaciones vulnerables, etc., por lo que la organización debe utilizar este tipo de herramientas antes de publicar de forma directa sus aplicaciones, aplicando las directrices de mitigación correspondientes en caso de presentar vulnerabilidades o debilidades; estos análisis no se deben limitar al despliegue de un nuevo desarrollo o una nueva release, sino que deben ser mantenidos en el tiempo mediante la ejecución periódica de los mismos.

246. Para encontrar vulnerabilidades o debilidades en los desarrollos propios, es necesario un análisis más exhaustivo que el anterior, en el que si bien diferentes herramientas automáticas pueden resultar de utilidad, sus resultados deben completarse obligatoriamente con pruebas manuales para garantizar la seguridad global de la aplicación y de la infraestructura que la soporta. En función de la criticidad de los datos manejados y de los riesgos a los que se expone una aplicación, la organización debe evaluar la conveniencia de realizar sobre la misma auditorías de código, con objeto de determinar que éste es correcto desde el punto de vista de su seguridad.

11. DIRECTRICES DE PROGRAMACIÓN (JAVA)

247. Los agujeros de seguridad o *bugs* son un problema demasiado común en las aplicaciones software de todo tipo, incluyendo por supuesto a las aplicaciones web; en este caso particular, a las vulnerabilidades que introduce una mala programación se une la superficie de ataque de la aplicación, que generalmente será accesible de forma remota y gestionará datos críticos para la organización. Ambos factores determinan un riesgo muy alto asociado a los errores de programación de aplicaciones web de cualquier tamaño: a pesar de considerar que los proyectos de gran envergadura son los únicos susceptibles de contener errores, incluso aplicaciones con un conjunto reducido de código y bien testeadas pueden incorporar vulnerabilidades de seguridad significativas.

248. Es casi imposible garantizar que un código generado en un lenguaje imperativo esté libre de fallos; en esta guía se aportan algunas directrices de programación que deben considerarse obligatorias en cualquier caso, aplicadas al lenguaje Java, cada vez más popular en el desarrollo de aplicaciones web. En el caso de que la organización esté trabajando en otro lenguaje, debe identificar los requisitos correspondientes de programación segura en el mismo y aplicarlos en su desarrollo, de forma que se minimicen los riesgos asociados a bugs en las aplicaciones.

11.1. INTRODUCCIÓN

249. La selección de un lenguaje de programación tiene impacto en la robustez de la aplicación. El lenguaje Java y la su máquina virtual proveen herramientas para evitar determinados errores de programación: entre otras, el lenguaje es tipado, es decir, tiene la capacidad de los lenguajes y clases de intercambiar información a través de modelos de definiciones, el entorno de ejecución proporciona gestión de memoria automática y comprobación de longitud en vectores, etc. Estas características y otras adicionales garantizan que la tecnología Java sea inmune a los ataques de desbordamiento de cola (*stack-smashing* y *buffer overflow*)

250. Sin embargo, la plataforma Java tiene sus propios problemas. Uno de los axiomas utilizados en su diseño es el de ofrecer un entorno seguro para ejecutar aplicaciones móviles: su arquitectura de seguridad protege a usuarios y sistemas de programas maliciosos descargados, pero no puede protegerlos de errores de implementación que suceden en programas o módulos en los que se confía. Estos errores pueden abrir agujeros de seguridad en la arquitectura diseñada, incluyendo la pérdida de información privada, el abuso de privilegios, el acceso a recursos sensibles por parte de usuarios no autorizados, por citar unos cuantos.

11.1.1. CALIDAD DE CÓDIGO JAVA

251. Para disponer de un software seguro la organización debe garantizar que la codificación de la aplicación sea de calidad; este criterio de calidad está caracterizado por los siguientes aspectos.

252. Eficiencia. El código debe hacer un uso proporcional de los recursos del entorno, sin exigir la última tecnología o presentando un consumo excesivo. A pesar de que Java, frente a otros lenguajes de programación, consume un número considerable de recursos por el uso de la máquina virtual, estos no suelen ser excesivos en comparación con los recursos disponibles actualmente en los sistemas que ejecutan las aplicaciones.

253. Portabilidad. Debe ser sencillo el cambio de entorno en el que se ejecuta la aplicación. El uso de Java garantiza un elevado nivel de portabilidad, por lo que es un lenguaje que directamente facilita este aspecto a la hora de desarrollar una aplicación de calidad.

254. Facilidad de uso. La comunicación con las diferentes partes integrantes de la aplicación debe ser lo más simple posible. De nuevo, Java está preparado para garantizar este aspecto puesto que la programación orientada a objetos proporciona mecanismos para poder realizar la comunicación entre diferentes entidades de forma clara.

255. Compatibilidad. La aplicación debe ser combinable con otras aplicaciones y no incorporar posibles incompatibilidades con ellas. Dado que cada aplicación se ejecuta en su propio *thread* de la máquina virtual de Java, las aplicaciones Java no son intrusivas con el resto de aplicaciones.

256. Corrección. Debe ser sencillo corregir los problemas que puedan aparecer en el desarrollo. Para ello el diseño de las clases y sus métodos deben realizar exactamente lo que necesitan, de forma que sea posible corregir una incidencia modificando una determinada clase identificada fácilmente y un método de tamaño reducido que realiza dicha funcionalidad.

257. Extensibilidad. Debe ser fácil realizar modificaciones en la aplicación para adaptarse a posibles cambios en sus especificaciones. Dada la facilidad de encapsular la funcionalidad y características necesarias en cada clase, en general suele ser sencillo extender éstas para cumplir con esta característica.

258. Robustez. Debe asegurarse que la aplicación se comporta correctamente ante situaciones anómalas. Una correcta validación de datos de entrada, así como el uso de bloques *try catch*, puede garantizar la estabilidad de la aplicación en situaciones inesperadas.

259. Verificabilidad. Debe ser sencillo comprobar si la aplicación funciona correctamente. El hecho de desarrollar las clases con una funcionalidad y un propósito determinado posibilitan el testeo y verificabilidad de ellas de forma sencilla.

260. Reutilización. Debe ser posible reutilizar ciertas partes de la aplicación en nuevas aplicaciones. El desarrollo de las clases de la aplicación debe estar desacoplado de la problemática propia de la aplicación en la medida de lo posible, de forma que los desarrollos puedan aprovecharse en otras aplicaciones que tengan similares necesidades.

261. Integridad. Los componentes de la aplicación deben estar protegidos contra los procesos que no dispongan de privilegios para acceder a ellos. Se deben emplear correctamente los modificadores de visibilidad de las diferentes clases, con el fin de garantizar la integridad de los componentes.

262. Los desarrollos abordados en la organización deben cumplir, en la medida de lo posible, los criterios de calidad anteriores.

11.1.2. SEGURIDAD DE CÓDIGO JAVA

263. Para que un código Java sea seguro debe cumplir al menos las siguientes directrices generales:

264. Acceso y herencia. Se debe limitar el acceso a los diferentes atributos y métodos de las clases para garantizar que no es posible comprometer el comportamiento de las mismas por una excesiva visibilidad de sus atributos o métodos.

265. Parámetros de entrada y salida. Se deben validar los valores de los parámetros en todos los casos, puesto que un mal uso de éstos puede provocar que se modifiquen en situaciones no esperadas, provocando un posible mal funcionamiento de la aplicación.

266. Construcción de objetos. Si el constructor de un objeto necesita invocar a un método para su correcta inicialización debe asegurarse que este método sea final, para que no pueda sobrescribirse por una clase hija que pueda comprometer el comportamiento de la clase.

267. Manejo de excepciones y logado. Se debe asegurar un correcto tratamiento de las excepciones de forma que no se ofrezca un exceso de información en las situaciones en las que no sea estrictamente necesario, así como también debe asegurarse el volcado de toda la información sensible para su posterior auditoría en el caso de que el sistema haya sido comprometido.

268. Cifrado y autenticación. Debe asegurarse un uso adecuado de los sistemas de cifrado y deben implantarse procedimientos de bloqueo de usuarios que intenten de forma reiterada el acceso al sistema de forma incorrecta.

269. Relación con sistemas externos. Debe garantizarse que los sistemas externos a los cuales acceden las diferentes aplicaciones se mantienen en un estado conocido y esperado en todas las circunstancias o que los posibles errores o desviaciones motivados por éstos son tratadas adecuadamente, con el fin de evitar posibles situaciones anómalas que puedan comprometer el funcionamiento del sistema.

270. Adicionalmente a estas directrices generales, se presentan a continuación particularidades y casos específicos que son igualmente necesarios en el código para considerarlo seguro.

11.2. CASOS DE USO GENERAL

11.2.1. USO DE SYSTEM.OUT.PRINTLN

271. Se debe evitar el uso de la salida estándar para devolver información de registro o error; se debe utilizar una clase específica de logado que redirija a un fichero o base de datos para su posterior auditoría, ya que la trazabilidad de la aplicación no está garantizada si los datos se vuelcan a la consola del sistema. Esta situación puede ser especialmente delicada, puesto que la trazabilidad de la aplicación queda comprometida a la visualización de datos en el momento exacto de la consola del servidor en el que se está ejecutando la aplicación.

11.2.2. INDENTACIÓN Y USO DE CARACTERES DE APERTURA Y CIERRE

272. Se debe realizar una correcta indentación del código siguiendo las convenciones de codificación de Java, así como el uso de bucles condicionales sin usar etiquetas de apertura y cierre. En caso contrario, es posible que el código no sea fácilmente legible, lo que puede provocar la ejecución de código en condiciones no deseadas ocasionando incluso vulnerabilidades dentro de la lógica de la aplicación.

273. Sirva de ejemplo la definición de este código:

```
If (iAccess == 10) checkUserAccess();
```

```
        enableAccess();
```

274. En el caso de que ocurra una incidencia que necesite ser revisada por parte de un técnico que no haya desarrollado la aplicación es posible que el código sea confuso, pudiendo provocar un nuevo error en la lógica de la aplicación que ocasione una brecha de seguridad. Para minimizar la posibilidad de esta situación, se debe realizar una correcta indentación del código así como un uso de las etiquetas adecuadas de apertura y cierre. Por tanto, una implementación considerada más segura del código anterior sería la siguiente, donde es fácilmente identificable el código que está siendo afectado por la condición considerada

```
if (iAccess == 10){
        checkUserAccess();
}
enableAccess();
```

11.3. IDENTIFICACIÓN, AUTENTICACIÓN Y AUTORIZACIÓN

11.3.1. LIMITACIÓN DE ACCESOS

275. Se debe incorporar al código fuente un mecanismo de bloqueo de cuentas de usuario para mitigar el impacto de ataques de fuerza bruta o, en su defecto, un mecanismo de control que notifique automáticamente a un equipo de seguridad ante intentos reiterados de acceso; también debe establecerse un mecanismo de bloqueo de orígenes (habitualmente, en función de la dirección IP desde la que se accede a la aplicación web), de forma que se garantice que sólo puede realizarse un número limitado de intentos de conexión desde un origen concreto, que serán procesados por la aplicación durante un período de tiempo dado.

11.4. SESIONES

276. Tras la autenticación correcta del usuario, el sistema debe mantener la validez de la misma, en la medida de lo posible, en las peticiones que se vayan realizando al servidor, evitando así la solicitud de credenciales en cada petición.

277. Para mantener activa esa identificación, se utilizan sesiones. Se trata de un identificador que el sistema asigna tras la validación y que se envía en cada petición para que el servidor pueda identificar quién la está realizando.

278. Este identificador se suele almacenar en una cookie en el propio equipo del usuario y para aportar seguridad a este mecanismo, se debe especificar un tiempo de validez de la sesión (de forma que si caduca vuelva a pedir las credenciales al usuario) y eliminar la cookie al finalizar la sesión para que si otro usuario intenta acceder al mismo sitio web le pida las credenciales y no sean utilizadas las de la sesión anterior.

279. En el caso de que se utilice una cookie para los usuarios anónimos ésta debe ser sustituida en el momento en el que un usuario se autentica con unas credenciales válidas en el sistema.

280. Siempre debe estar disponible un enlace para el cierre de sesión que invalide en el servidor la sesión del usuario con el fin de evitar posibles suplantaciones.

11.5. VALIDACIÓN DE DATOS

11.5.1. CLASES

11.5.1.1. Longitud de clases

281. Se debe evitar el desarrollo de clases demasiado extensas; esta directriz es básica en la programación orientada a objetos, siendo una de las principales recomendaciones de las convenciones de codificación y diseño de clases Java. El desarrollo de clases extensas provoca un grado de acoplamiento no necesario, que puede ocasionar que determinadas clases que no están preparadas para ofrecer una funcionalidad concreta sean reutilizadas en contextos de aplicaciones que puedan comprometer la seguridad de los datos almacenados por la clase.

11.5.1.2. Longitud de métodos

282. Se debe evitar el uso de métodos demasiado extensos. Estos métodos provocan una revisión y seguimiento complicados de la lógica asociada, introduciendo puntos de posibles vulnerabilidades y acoplamiento entre las variables definidas en el método, además de no ceñirse a las convenciones de codificación y diseño en Java.

11.5.1.3. Imports

283. Se debe evitar el uso de la importación de todo el paquete mediante el uso de la directiva:

```
import java.util.*
```

284. Únicamente deben aparecer en la lista de imports las clases realmente utilizadas, ya que de lo contrario se puede provocar un error a la hora de codificar la aplicación que podría ocasionar problemas durante la ejecución de código. Por ejemplo, la clase Date se encuentra disponible tanto en el paquete java.util como en el paquete java.sql.

```
Import java.sql.*;
publicClassNonFInal {
        private Date dValue = null;
        public NonFinal() {
                prepareData();
        }
        public void prepareData() {
                dValue = new Date();
        }
}
```

285. Esta situación provoca que el objeto dValue pueda ser declarado, sin que el compilador muestre ningún mensaje y sea el programador el que revise la declaración del objeto si necesita acceder a algún método que no está disponible en la implementación de la clase. De otro modo, el error en la definición de los datos no es detectable, pudiendo variar significativamente el comportamiento de la aplicación por la diferente implementación de la funcionalidad de cada clase. Así, una implementación más segura de la clase anterior sería:

```
import java.sql.Connection;
import java.util.Date;
public Class NonFInal {
        private Date dValue = null;
        public NonFinal() {
                prepareData();
        }

        public void prepareData() {
```

```
                dValue = new Date();
        }
}
```

11.5.1.4. Atributos estáticos públicos

286. Las diferentes clases pueden acceder y modificar los atributos que son públicos, estáticos y no finales. El *manager* de seguridad se encarga de determinar la política de seguridad activa y realizar las verificaciones de control de acceso a través de los diferentes métodos de las clases.

287. Si se definen los atributos como públicos, cualquier clase que tuviese acceso al contenedor de clases de la aplicación podría acceder a estos valores sin acceder a través del control y supervisión del *manager* de seguridad, pudiendo provocar un comportamiento no deseado. Por tanto, se deben tratar los atributos públicos estáticos como constantes, es decir, se debe añadir el cualificador final a estos atributos y almacenar en ellos únicamente objetos no mutables.

288. Se muestra a continuación un ejemplo de clase y definición correcta e incorrecta de atributos:

```
public final class Directions {
        //uso incorrecto de atributo
        public static int NORTH = 1;
        //uso incorrecto de atributo, el objeto es mutable
        public static List lValues = new ArrayList();
        //uso correcto de atributo
        public static final int SOUTH = -1;
}
```

11.5.1.5. Métodos de envoltura

289. Si algún atributo interno de una clase debe ser accesible y modificable, se debe declarar el atributo como privado y permitir el acceso a través de métodos, que deberán ser públicos o protegidos dependiendo de la visibilidad que se quiera ofrecer sobre dicho atributo.

290. Sirva de ejemplo la definición de esta clase:

```
public class RootLevel {
        //atributo que puede ser accedido globalmente
        public int iLevel = 0;
        //atributo que puede ser accedido globalmente
        public List lValues = new ArrayList();
}
```

291. Los atributos declarados son públicos, por lo que cualquier clase puede acceder a ellos, lo que puede provocar una situación no controlada en el código que puede llegar a comprometer la seguridad del sistema. Esta clase puede ser utilizada en la lógica de la aplicación de la siguiente forma:

```
int iSize = RootLevel.lValues.size();
for (int i=0;i<iSize;i++) {
        String sData = (String) RootLevel.lValues.get(i);
        …
```

```
        }
```

292. Sin embargo, una clase que tuviese acceso a esta funcionalidad y quisiera comprometer la funcionalidad del sistema podría borrar los valores de la lista, añadir nuevos elementos o modificar los valores existentes:

```
//Limpia la lista de valores
RootLevel.lValues.clear();

//Añade en la primera posición del vector el valor -1
RootLevel.lValues.add(0, "-1");
```

293. Esta situación es especialmente delicada en instancias de clases que son utilizadas para acceder a base de datos confiando en los valores de los atributos definidos en las clase, puesto que, por ejemplo, podría provocar una vulnerabilidad de inyección SQL. Para solventar esta situación, se deben implementar los métodos necesarios para ofrecer la visibilidad adecuada a los atributos de las clases y garantizar que en todo momento sea trazable y controlable la modificación de sus valores. Así, una implementación más segura de la clase descrita con anterioridad sería la siguiente:

```
public class RootLevel {
        //atributo que puede ser accedido globalmente
        private int iLevel = 0;
        //atributo que puede ser accedido globalmente
        private List lValues = new ArrayList();
        public int getLevel() {
                return iLevel;
        }
        public List getValues() {
                return new ArrayList(lValues);
        }
}
```

11.5.1.6. Envolturas de métodos nativos

294. El código Java está sujeto a la supervisión del *manager* de seguridad; sin embargo, el código nativo que se puede invocar no lo está; además, mientras que el código Java puro es inmune a ataques de desbordamiento de buffer, los métodos nativos pueden no serlo. Para ofrecer una protección extra durante la invocación de código nativo, se debe declarar el método nativo como privado. De esta forma, es posible implementar un método público que realice todas las validaciones necesarias para asegurar la correcta ejecución del código nativo y posteriormente realizar la llamada necesaria para la ejecución de éste.

295. Como ejemplo, la siguiente clase que muestra los métodos nativos de una determinada clase:

```
public final class NativeMethodPublisher {
        public native void nativeOperation(byte[] data, int offset, int len);
}
```

296. Desde cualquier punto de la aplicación es posible invocar al método nativo sin realizar ningún tipo de validación sobre el contenido que se está pasando, lo que puede provocar un comportamiento inseguro de la lógica de la aplicación. Para solventar esta situación se deben añadir las comprobaciones necesarias a los datos que se deben pasar al método nativo, por lo que una implementación más segura podría ser la que se muestra a continuación:

```
public final class NativeMethodWrapper {
   // método nativo privado
private native void nativeOperation(byte[] data, int offset, intlen);

   // envoltorio que realiza la validación de seguridad
public void doOperation(byte[] data, int offset, intlen) {
      // se comprueban los permisos
securityManagerCheck();
if (data == null) {
throw new NullPointerException();
}
      // se clonan los datos de entrada
      data = data.clone();
      // se realizan las validaciones necesarias
if (offset < 0 || len< 0 || offset >data.length -□“ len) {
throw new IllegalArgumentException();
}
//se invoca el método nativo
nativeOperation(data, offset, len);
   }
}
```

11.5.1.7. Literales

297. Se debe evitar el uso de literales en la codificación de las clases; es preferible disponer de una clase de utilidades con etiquetas para su utilización por parte del resto de clases y disponer de un único punto de acceso a éstos. De esta forma, si se decide modificar en cualquier momento el valor de estos literales, sólo existirá un punto en el que se deba realizar la modificación, evitando la incorporación de posibles errores por no realizar las modificaciones en todos los puntos en los que se utiliza y minimizando las implicaciones de seguridad relacionadas con el acceso, por ejemplo, a las bases de datos.

11.5.1.8. Comprobación de cadenas

298. Se debe evitar el uso de comparaciones de *strings* que puedan llevar a excepciones del tipo NullPointerException cuando se compara un *string* con una constante. Por ejemplo, el código siguiente provoca una excepción que interrumpe el flujo de la aplicación y que puede no ser tratado de forma correcta por el resto de lógica:

```
String sVal = null;
sVal.equalsIgnoreCase(“null”);
```

299. Sin embargo, la instrucción siguiente devuelve *false* y se puede continuar con el flujo del programa sin comprometer el funcionamiento deseado, sin necesidad de comprobar si la variable se encuentra inicializada.

```
“null”.equalsIgnoreCase(sVal)
```

11.5.1.9. Salida de métodos

300. Se debe evitar la posibilidad de tener varios puntos de salida para un mismo método; por ejemplo, las instrucciones siguientes provocan un problema de legibilidad de código y un

posible error en el valor de retorno si no se tratan adecuadamente todas las condiciones del método.

```
…
if (comparacion) {
        return false;
} else {
        return true;
}
```

301. Frente al anterior, un código equivalente como el siguiente asegura un retorno controlado del valor del método en el que todas las condiciones disponen de un único punto de salida.

```
boolean bRet = false;
…
if (comparacion) {
        bRet = false;
} else {
        bRet = true;
}
return bRet;
}
```

11.5.2. OBJETOS

11.5.2.1. Construcción

302. No se debe invocar a métodos que pueden ser sobrescritos desde el código que implementa el constructor, ya que un código malicioso podría acceder a la referencia directa del objeto construido antes de que éste haya sido completamente inicializado. Del mismo modo, no se debe invocar a métodos que pueden ser sobrescritos desde métodos que devuelvan instancias de objeto, como puede ser el método clone.

303. En el siguiente ejemplo, en caso de que una clase hija sobrescriba el método initLevel, el comportamiento de la clase puede verse comprometido, pudiendo mostrar una información no contemplada por la aplicación:

```
public Class RootLevel {
        private int iLevel = 0;
        public RootLevel() {
                initLevel();
        }
        public void initLevel() {
                this.iLevel = -1;
        }
        public void increaseLevel(final intiValue)  {
                this.iLevel += iValue;
        }
}
```

304. Una implementación más segura de la clase anterior podría ser la siguiente, ya que, independientemente del comportamiento cualquier clase que extienda de ésta, en el momento que finalice la construcción de la instancia de la clase, la variable iLevel tendrá el valor -1 tal y como indica el método initLevel:

```
public Class RootLevel {
```

```
        private int iLevel = 0;
        public RootLevel() {
                initLevel();
        }
        final private initLevel() {
                this.iLevel = -1;
        }
}
```

11.5.3. ACCESO Y HERENCIA

11.5.3.1. Acceso a clases, interfaces, métodos y campos

305. Un paquete Java agrupa un conjunto de clases e interfaces. Se deben declarar todas las clases e interfaces como públicos si se desea que sean considerados como parte de la interfaz de programación de la aplicación, conocida como API, y que puedan ser accedidos desde otros puntos de la aplicación. En caso contrario, debe limitarse su visibilidad al paquete en el que se encuentran definidos. Del mismo modo, se deben definir todos los miembros de las clases como públicos o protegidos, si también deben formar parte del API y, en caso contrario, debe limitarse su visibilidad al paquete, si forman parte de la funcionalidad de éste o limitar su visibilidad a la implementación de la clase que las contiene.

306. Además, se debe prestar especial atención a la hora de incrementar el nivel de acceso de métodos heredados, ya que esto puede provocar errores por asunciones realizadas por la superclase. Sirva como ejemplo la siguiente definición de clases:

```
public class NonFinal {
        public NonFinal() {
                //llamada a método para inicializar valores
                initValues();
         }
        protected void initValues() {
        }
}

public class ExtendNonFinal extends NonFinal {
        public ExtendNonFinal() {
                //llamada al constructor de la clase padre
                super();
        }

        //Modificación de la visibilidad del método.
        public void initValues()  {
        …
        }
}
```

307. De este modo la clase ExtendNonFinal puede sobrescribir el método initValues y declararlo como público, lo cual provoca que un código malicioso pueda invocar este método en cualquier instancia de una clase y modificar el comportamiento esperado por la clase padre. Si la implementación de la superclase no está preparada para manejar este conjunto de llamadas podría provocar una excepción en ejecución, que puede llegar a ocasionar una pérdida de información o dejar a los objetos en un estado indeterminado que comprometa la

seguridad del desarrollo. Así, una implementación más segura del conjunto de clases anterior no debería modificar la visibilidad del método en la clase hija:

```
public class ExtendNonFinal extends NonFinal {
        public ExtendNonFinal() {
                //llamada al constructor de la clase padre
                super();
        }

        protected void initValues()  {
        …
        }
}
```

308. Una excepción a esta directriz se debe realizar en las clases que implementan la interfaz *cloneable*. En estos casos, el acceso al método *clone* debe ser incrementado de protegido a público.

11.5.3.2. Extensión de clases y métodos

309. Se deben diseñar las clases y los métodos de forma adecuada para garantizar su correcta funcionalidad en caso de ser utilizadas como superclases o se deben declarar como finales; en caso contrario, una clase o un método puede ser sobrescrito por parte de un código malicioso.

310. Si una clase es pública y no está declarada como final y quiere limitar la herencia únicamente para implementaciones en las que confíe, se debe confirmar el tipo de la clase de la instancia creada; esto debe realizarse en todos los puntos donde una instancia de una clase no final puede ser creada. En caso de detectar el uso de una subclase, se debe forzar la comprobación del SecurityManager para bloquear implementaciones maliciosas.

311. Sirva como ejemplo la siguiente definición de clase:

```
public class NonFinal {
        public NonFinal() {
                initValues();
        }

        public void initValues() {
        …
        }
}
```

312. De este modo una clase puede sobrescribir el método initValues y modificar su comportamiento, lo cual provoca que un código malicioso pueda comprometer el comportamiento esperado por la clase padre ya que la inicialización de sus valores no puede ser garantizada en el constructor. Por tanto, una implementación más segura de la clase anterior debe realizar las comprobaciones pertinentes en la clase que está siendo invocada, así como definir como final el método invocado en el constructor del objeto de la clase:

```
public class NonFinal {
        // constructor
        public NonFinal() {
           // Obtenemos una instancia de la clase
        ClasscItem = getClass();
```

```
        // Confirmamos el tipo de la clase
        if (cClass != NonFinal.class) {
                // comprobamos la seguridad
                securityManagerCheck();
                }
        // continuamos
        initValues();
        }

        private void securityManagerCheck() {
        SecurityManager sm = System.getSecurityManager();
                if (sm != null) {
                sm.checkPermission(...);
                }
        }

        final public void initValues() {
                …
        }
}
```

11.5.3.3. Herencia de una clase

313. Las subclases no tienen la capacidad de mantener control absoluto sobre su propio comportamiento ya que una superclase puede modificar la implementación de un método que las subclases no sobrescriban; incluso si una subclase sobrescribe todos los métodos todavía puede ver comprometida su funcionalidad si se introducen nuevos métodos. Estos cambios en la superclase pueden modificar la lógica que las clases herederas asumen para realizar su implementación y ocasionar así las consecuentes vulnerabilidades de seguridad.

314. Si por ejemplo se tiene la clase siguiente:

```
Public class RootLevel {
        private int iValue = 0;
        public RootLevel() {
                ...
        }

        public void setLevel(final intiValue) {
                this.iValue = iValue*0,8 + 0,5;
        }
}
```

315. Y a continuación se define una clase que hereda de ésta y sobrescribe el método setLevel, de la forma siguiente:

```
public class CompanyRootLevel {
        //método sobrescrito que hace otra funcionalidad
        public void setLevel(final intiValue) {
                this.iValue = iValue*0,5 + 0,8;
        }
}
```

316. La lógica de la aplicación confiará que el valor introducido en el atributo iValue de la clase tendrá el valor calculado por ella y no por su clase padre e implementará su lógica asociada considerando que el valor de esta variable no puede ser modificado de otra forma. Sin

embargo, si en una futura implementación de la clase RootLevel se añade el siguiente método:

```
public void resetLevel() {
        //algún tipo de código que modifique el valor del atributo
                ...
}
```

317. Será posible acceder a este método por parte de las instancias de la clase CompanyRootLevel, por no haber sobrescrito éste, con el consecuente desconocimiento del comportamiento de la aplicación. Por tanto, se debe prestar especial atención a los métodos no sobreescritos de las clases abstractas o de las clases de las que se hereda funcionalidad, para que modificaciones futuras no incorporen nuevos riesgos a los desarrollos realizados.

11.5.4. ENTRADA Y SALIDA

11.5.4.1. Entradas y salidas modificables

318. Si un método no opera directamente sobre un parámetro de entrada que puede ser modificado, en general, debe generar una copia de éste y únicamente realizar la lógica del método en dicha copia; de otro modo, un código malicioso podría modificar el parámetro para provocar condiciones de carrera en dicho método.

319. Por ejemplo, en el siguiente método, independientemente del valor de retorno, siempre que la validación sea correcta, es decir, que el atributo que contiene el valor por defecto sea igual a cero, el valor del atributo será modificado y pasará a contener el valor -1, que será el valor que se visualice en toda la lógica de la aplicación por el resto de objetos independientemente del valor de retorno:

```
public class Copy {
        public boolean checkMutableInput(CompanyRootLevel level) throws NullPointerException {
                if (level == null) {
                        throw new NullPointerException();
                }
                if (level.getDefaultValue() == 0) {
                        level.setDefaultValue(-1);
                }
                return level.hasDefaultValue();
        }
}
```

320. Esta situación da lugar a que determinadas validaciones no se realicen correctamente, provocando posibles errores en la aplicación que pueden comprometer la lógica funcional deseada. Por ejemplo, el código siguiente se ejecutará de forma correcta y mostrará un mensaje de log con el valor del atributo esperado, “Valor del nivel 3”:

```
CompanyRootLevel level = new CompanyRootLevel();
//Se calcula el valor actual y se fija en el objeto
level.setDefaultValue(3);
if (checkMutableInput(level)) {
        //Acciones a realizar
        }
logger.info(“Valor del nivel “+level.getDefaultValue());
...
}
```

321. Por el contrario, el código siguiente no se ejecutará de forma correcta y mostrará un mensaje de log con el valor del atributo no esperado, "Valor del nivel 0":

```
CompanyRootLevel level = new CompanyRootLevel();
//Se calcula el valor actual y se fija en el objeto
level.setDefaultValue(0);
if (checkMutableInput(level)) {
        //Acciones a realizar
        }
logger.info("Valor del nivel "+level.getDefaultValue());
...
}
```

322. Para solventar esta situación, se deben realizar copias de los objetos sobre los que se van a realizar las operaciones que no deben ser reflejadas en el resto de la aplicación. La copia de las entradas puede realizarse de dos formas, para clases finales que implementen el método *clone*, y para clases no finales.

```
public class Copy {
        public void copyMutableInput(CompanyRootLevellevel) throwsNullPointerException {
                if (level == null) {
                        throw new NullPointerException();
                }
                //a pesar de usar el mismo nombre de variable, al ser creado el objeto dentro del método este valor
                //no será modificado en el contenido de la variable original
                level = level.clone();
                //realizar las operaciones
                ...
        }
}
```

323. Se debe prestar especial atención a las copias de los *arrays* de objetos, ya que, en este caso, si se utiliza el método *clone*, no se realiza una copia de los valores, de forma que los valores referenciados por las diferentes variables son los mismos. En este caso, se deben crear las copias manualmente:

```
public void doCopy(Integer[] iValues) {
        if (iValues == null) {
                throw new NullPointerException();
        }
        //mismos elementos
        Integer[] iSameValues = iValues.clone();
        //mismos valores, pero diferentes elementos
        Integer[]  iCopyValues = new Integer[iValues.length];
        for (int i=0; i<iValues.lenght; i++) {
                //copiamos cada elemento
                //sí se puede usar clone en este método puesto que  ya es un objeto
                iCopyValues[i] = iValues[i].clone();
        }
}
```

324. Estos conceptos se aplican del mismo modo a los valores de retorno; se debe devolver una copia de cualquier objeto que puede ser modificado, salvo que el método explícitamente deba devolver la referencia a su propio objeto. En caso contrario, un código malicioso podría modificar los valores de las variables privadas de un objeto con el valor de retorno.

11.5.4.2. Funcionalidad de copia para clases modificables

325. Si se diseñan clases cuyos atributos pueden ser modificados se debe implementar la funcionalidad de copia de sus instancias, lo que permite que las diferentes instancias de la clase sean pasadas de forma segura cuando son utilizadas por otros objetos. Esta funcionalidad puede ser implementada mediante un constructor que realice la copia o implementando la interfaz *java.lang.Clonable* y declarando el método *clone* como público.

326. Cualquier implementación del método *clone* debe invocar en primer lugar el método *clone* de la clase superior; a partir de ese momento, debe modificar cualquier objeto modificable en la nueva instancia clonada con copias de dicho objeto. Esto garantiza que la instancia clonada es independiente del objeto original.

327. Si la clase es final y no provee la funcionalidad de copiado, puede ser posible realizar una copia manual de los datos; sin embargo, esto únicamente hará que se copien los atributos que sean declarados como públicos o los que dispongan de un método para recuperar su valor, por lo que cualquier valor que no sea público, no será copiado y podrá provocar un estado inconsistente del objeto que se desea clonar.

328. Veamos por ejemplo el siguiente código:

```
public class NonFinal {
        private int iState = 0;
        private String sLabel = “”;

        public NonFinal() {
                iState = Constants.INIT;
        }

        public String getLabel() {
                returnthis.sLabel;
        }

        public void setLabel(StringsValue) {
                this.sLabel = sValue;
        }

        public void setState(intiValue) {
                this.iState = iValue;
        }
}
```

329. Si se debe realizar una copia de un objeto de la clase NonFinal, al no implementar el método *clone* debería realizarse una copia de los atributos que han sido declarados como públicos, por lo que el atributo iState no podría ser copiado. Esto podría provocar que el objeto copiado no estuviera en el mismo estado que el objeto original clonado, ocasionando un posible comportamiento inesperado de la aplicación:

```
NonFinal nfItem1 =  NonFinal();
nfItem1.setState(3);
//Creamos una copia del objeto
NonFinal nfItem2 = NonFinal();
nfItem2.setValue(nfItem1.getValue());
```

330. Como puede observarse, el valor del estado en ambos objetos es diferente, por lo que el comportamiento del código que se ejecute a continuación dependerá del objeto que sea tratado por la aplicación, pudiendo ocasionar comportamientos no deseados que causen fuga

de información sensible por el uso inadecuado de la copia de los objetos. Por tanto, una implementación más segura debería implementar el método *clone* o definir como públicos todos los métodos que permiten acceder a todos los atributos del objeto:

```
public class NonFinal {
        private int iState = 0;
        private String sLabel = “”;

        public NonFinal() {
                iState = Constants.INIT;
        }

        public String getLabel() {
                returnthis.sLabel;
        }

        public int getState() {
                returnthis.iState;
        }

        public void setLabel(StringsValue) {
                this.sLabel = sValue;
        }
}
```

11.5.4.3. Validación de tipos

331. Se deben validar los datos esperados desde la entrada y garantizar que el tipo enviado es el esperado, con el fin de evitar un uso inadecuado de los valores que se insertan en los formularios de envío. Debe prestarse especial atención a los campos ocultos y valores de desplegables, puesto que la validación de estos campos no debe asumirse como correcta puesto que es posible modificarlos en la interfaz del cliente.

332. Por ejemplo, en la siguiente definición de método, aunque el campo esperado por la lógica de la aplicación es de tipo entero, el envío se realiza encapsulado en un *string* que puede ser utilizado para componer las diferentes sentencias SQL que se ejecutan contra la base de datos, lo que puede provocar la ejecución de sentencias SQL con código no esperado, puesto que no se comprueba el contenido de la variable enviada:

```
public String getParam(Vector vParams) {
        return (String) vParams.get(0);
}
```

333. Una implementación más segura, asumiendo que el valor debe ser de tipo *integer* aunque enviado como *string*, como se ha comentado con anterioridad, sería la siguiente:

```
public int getParam(final Vector vParams) throws InvalidArgumentException, NumberFormatException {
        intiRet = 0;
        //Comprobamos que el vector existe y tiene elementos
        If (vParams != null&& !vParams.isEmpty()) {
                if (vParams.get(0)  instanceof String) {
                        StringValue = (String) vParams.get(0);
                } else {
                        throws new InvalidArgumentException();
                }
        }
```

```
        return Integer.parseInt(sValue);
}
```

334. De esta forma se valida tanto el tipo como los valores enviados, provocando la propagación de la excepción adecuada en cada caso para que la aplicación sea capaz de tratar la información de forma adecuada sin comprometer en ningún caso su lógica.

11.5.4.4. Validación de datos mediante listas blancas

335. Los datos recibidos desde las clases de control deben validarse en la medida de lo posible, mediante lista blanca, con el fin de garantizar que éstos son correctos; por tanto, una implementación mediante listas blancas involucraría la validación en la parte del servidor del tipo y del valor de la variable:

```
Object objData = request.getParameter("sData");
String sData = null;
//Se comprueba el tipo
If (objData instanceOf String) {
        sData = (String) objData;
        }
//Se comprueba el valor suponiendo que los valores de la lista blanca
//esperados están en una lista
If (sData != null) {
        //En caso de que el valor no esté en la lista blanca lo reseteamos
        If (!lValues.contains(sData)) {
                sData = null;
                }
        }
```

11.5.4.5. Validación de campos ocultos

336. Los datos recibidos desde los formularios de la aplicación se deben validar, en la medida de lo posible, mediante el uso de las listas blancas explicado con anterioridad, con el fin de garantizar que éstos son correctos.

337. Por ejemplo, en la definición del formulario siguiente, los campos de tipo *hidden* declarados no son visibles en los navegadores; sin embargo, existen servicios que permiten mostrar los valores de los campos ocultos, así como una posible modificación de éstos:

```
<form action="" method="GET"  name="formData" target="">
<input type="hidden" name="sData" value="<%=sData%>">
</form>
```

338. Si se confía en el valor del parámetro enviado y su tipología sin comprobarlo, es posible utilizar el valor enviado de la siguiente forma, asumiendo que el valor introducido es uno de los valores conocidos de la aplicación:

```
String sDataType = (String) request.getParameter("sData");
…
//uso de la variable sDataType
```

339. Sin embargo, un usuario que tuviese acceso a este formulario y quisiera comprometer la funcionalidad del sistema podría modificar los valores del atributo oculto alterando su valor, por ejemplo mediante la definición de este formulario:

```
<form action="" method="GET"  name="formData" target="">
<input type="hidden" name="sData" value="<script>alert(document)</script>">
</form>
```

340. Esta situación es especialmente delicada en formularios que realizan la validación en el lado del servidor y devuelven los valores para que vuelvan a ser representados en el navegador, puesto que podría provocar, entre otras, una vulnerabilidad de *Cross Site Scripting*. Para solventar esta situación, se deben implementar los mecanismos necesarios para realizar las validaciones de tipos y de los datos recibidos en el servidor, tal y como se ha indicado previamente.

11.6. MANEJO DE ERRORES

11.6.1. PROPAGACIÓN DE INFORMACIÓN SENSIBLE

341. Los objetos que contienen la información de las excepciones contienen datos relevantes para tratar la excepción de forma adecuada, pero pueden dar visibilidad de información relevante a usuarios no autorizados si no se tratan correctamente. Por ejemplo, una excepción relacionada con la ruta, que no se pasa como parámetro, de un fichero no encontrado que es propagada hasta la interfaz de usuario puede exponer la distribución del sistema de ficheros.

342. Con el fin de evitar este exceso de información se deben capturar las excepciones para eliminar toda la información que pueda revelar datos del sistema y posteriormente relanzar la misma excepción, o una excepción de otro tipo con otro mensaje.

11.6.2. GESTIÓN DE EXCEPCIONES

343. Se debe evitar la captura de las excepciones con la clase *Exception*, ya que esto provoca el enmascaramiento de posibles errores que deban ser considerados; puesto la aplicación se comportaría del mismo modo independientemente de la excepción.

344. Por ejemplo, la definición del siguiente método, provocará que cualquier excepción que se produzca en el cuerpo del método sea propagada a la aplicación, sin realizar una captura previa de la excepción en concreto. Además, esto conlleva que el resto de la lógica de la aplicación no capture excepciones concretas, sino que se capture la excepción general, comprometiendo el tratamiento necesario de las excepciones generadas:

```
public void loadProperties()  throws Exception {
        ….
        File f = new File(Constants.CONF_FILE);
…
}
```

345. Para solventar esta situación, se deben implementar los mecanismos necesarios para escalar las excepciones del tipo adecuado, así como la captura de las excepciones adecuadas. En el ejemplo anterior, un tratamiento correcto sería el siguiente:

```
public void loadProperties() throws FileNotFoundException {
        …
        File f = new File(Constants.CONF_FILE);
        …
}
```

11.7. SISTEMA DE ARCHIVOS

11.7.1. USO DE RUTAS PRIVADAS

346. No se deben utilizar rutas públicas para almacenar la información, puesto que un usuario podría acceder a esta información sin disponer de la autorización necesaria. Por ejemplo, suponiendo que la ruta /usr/share/ fuese de uso compartido, el código siguiente estaría almacenando un fichero en una ruta que podría ser accesible desde fuera de la aplicación:

```
FileOutputStream foItem = new FileOutputStream(new File("/usr/share/doc.xml");
foItem.write(bData);
foItem.close();
```

347. Esta situación es especialmente delicada desde el punto de vista de la seguridad de la información, puesto que podría provocar una lectura no autorizada de los datos que están siendo almacenados o una modificación de su contenido. Para solventarla se debe garantizar que la ruta en la que se almacenen los ficheros únicamente sea accesible por la aplicación.

11.7.2. USO DE RUTAS ABSOLUTAS

348. No se deben utilizar rutas fijas para almacenar los datos que estén escritas directamente sobre el código fuente. Por ejemplo, veamos el siguiente código:

```
FileOutputStream foItem = new FileOutputStream(new File("/usr/app/data/doc.xml");
foItem.write(bData);
foItem.close();
```

349. El hecho de incorporar una ruta absoluta en el almacenamiento del fichero obliga a instalar la aplicación en un entorno determinado y a depender en exceso de la distribución de directorios del sistema operativo, situación que puede provocar que la instalación de la aplicación en un entorno diferente al actual provoque anomalías difícilmente detectables hasta que se produzca un error, con la consecuente pérdida de información de los ficheros asociada.

350. Para corregir esta situación se deben utilizar rutas relativas, que garanticen que la aplicación puede desplegarse en diferentes entornos y que una modificación en la organización del sistema de ficheros no provoca ninguna anomalía en el funcionamiento de la aplicación. Para el ejemplo anterior, una implementación más segura sería la siguiente:

```
FileOutputStream foItem = new FileOutputStream(new File("data/doc.xml");
foItem.write(bData);
foItem.close();
```

351. Esta situación es especialmente crítica en aspectos asociados a la continuidad del servicio que proporciona la aplicación, puesto que si por algún motivo fuese necesario aplicar los procedimientos definidos en el plan de contingencia, es posible que la puesta en producción en el nuevo sistema llegara a requerir algún cambio en la definición de estas rutas.

11.7.3. USO DE NOMBRES FIJOS

352. No se debe usar un nombre fijo para almacenar en archivos la información enviada por los usuarios puesto que, en caso de existir concurrencia, es posible que el fichero no se almacene correctamente o se sobrescriba por otro, haciendo imposible su lectura posterior.

353. Por ejemplo, en el siguiente código, el hecho de incorporar un nombre fijo puede provocar que la ejecución sucesiva del mismo por parte de dos hilos llegue a sobrescribir el contenido del fichero, generando condiciones de carrera en las que la información leída no sea la

esperada (ya que el archivo contendría la información de otra petición), excepciones de acceso al fichero o información corrupta, con imposibilidad de acceder a su contenido.

```
FileOutputStream foItem = new FileOutputStream(new File("/usr/app/data/doc.xml");
foItem.write(bData);
foItem.close();
```

354. Una implementación más segura del código anterior sería la siguiente:

```
String sName = "doc"+new Date().getTimeInMilis()+".xml";
FileOutputStream foItem = new FileOutputStream(new File(sName));
foItem.write(bData);
foItem.close();
```

355. De esta forma, cada vez que se invoque al código indicado, el nombre del fichero almacenado será diferente y cada ejecución podrá acceder al fichero que está procesando, garantizando de esta forma el contenido del fichero y su legibilidad.

11.7.4. ALMACENAMIENTO EN MEMORIA

356. No se deben utilizar los parámetros que se han enviado a los métodos para modificar los valores de éstos; en el caso de ficheros enviados como *byte arrays* la situación es incluso peor, puesto que esta práctica elimina la posibilidad de recuperar los datos del fichero original enviado. Por ejemplo, en el código siguiente, los datos enviados originalmente al método no se encuentran disponibles en el resto del código del método, lo que provoca problemas de trazabilidad a la hora de volcar la información de lo que ha podido suceder en el código para un posterior análisis en caso de anomalías:

```
public void setFile(byte[] bData) {
        bData = FileUtils.decodeFile(bData);
        …
        //uso de la variable bData en la lógica de la aplicación
}
```

357. Aunque esta funcionalidad es difícilmente explotable por parte de un usuario malintencionado, es una buena práctica para garantizar que en todo momento se dispone de la información necesaria para poder reproducir cualquier situación que pudo llevar al sistema a un estado inconsistente y con posibles pérdidas de información. Una implementación más segura podría ser la expuesta a continuación, de forma que en cualquier punto de la aplicación se disponga de los datos originales y del contenido de los datos modificados:

```
public void setFile(byte[] bData) {
        byte[] bFile = FileUtils.decodeFile(bData);
        ..
        //uso de la variable bFile en la lógica de la aplicación,
        //con los datos originales disponibles en la variable bData
}
```

11.7.5. CREACIÓN DE CADENAS PARTIR DE BYTE ARRAYS

358. En el caso de que sea necesario pasar el fichero como *String* a diferentes métodos, esta acción debe ser realizada en un único punto, con el fin de asegurar que la creación del objeto se realiza una sola vez y que este valor es el mismo que utilizan todos los métodos. Por ejemplo, veamos el siguiente método:

```
public void getData(byte[] bData) {
String sCompany = getCompany(new String(bData, “ISO-8859-1”));
…
String sPerson = getPerson(new String(bData, “ISO-8859-1”));
…
String sId = getPersonId(new String(bData, “ISO-8859-1”));
…
}
```

359. El hecho de crear un nuevo *String* cada vez que se desea utilizar la variable puede provocar que, si se realizan modificaciones, controladas o no, de la variable que almacena los datos, la generación del *String* utilizado por parte de la lógica de la aplicación sea diferente en cada uno de los casos:

```
public void getData(byte[] bData) {
String sCompany = getCompany(new String(bData, “ISO-8859-1”));
…

//se modifica la variable bData dentro de un método de forma no controlada
checkDataValues(bData);  //Modifica el valor del byte array
String sPerson = getPerson(new String(bData, “ISO-8859-1”));
//El string utilizado para obtener la persona puede no tener ninguna relación con el esperado por la aplicación
```

360. Una implementación más segura del método anterior podría ser la siguiente, en la que cualquier punto de la aplicación utilice el mismo valor para proceder a ejecutar la lógica de la aplicación necesaria:

```
public void getData(byte[] bData) {
//Este valor sera siempre el mismo durante la ejecución del código
String sValue = new String(bData, “ISO-8859-1”);
String sCompany = getCompany(sValue);
…
String sPerson = getPerson(sValue);
…
String sId = getPersonId(sValue);
..
}
```

11.8. AUDITORÍA Y REGISTRO

11.8.1. USO DE LOGGER NO ESTÁTICO

361. Se debe evitar el uso del objeto de logado como no estático, que provoca que cada vez que se crea una nueva instancia de la clase se deba crear un nuevo objeto de tipo logado:

```
private final Logger logger = Logger.getLogger(“clase”);
```

362. Debemos asegurar que únicamente se crea un objeto de tipo logger para todas las instancias de la clase, minimizando la carga de la máquina virtual utilizada para cargar la información:

```
prívate static final Logger logger = Logger.getLogger(“clase”);
```

11.8.2. REGISTRO INCORRECTO DE EXCEPCIONES

363. El logado completo de la información de las excepciones garantiza la correcta trazabilidad en caso de que ocurran situaciones que requieran un análisis posterior; esta información es especialmente relevante a la hora de garantizar la trazabilidad y auditoría de la aplicación, tanto en la detección de errores como en el caso de que la seguridad del sistema desarrollado se vea comprometida.

364. Por ejemplo, en el siguiente código la información que se almacena de la excepción contiene únicamente el tipo de excepción producida y su descripción, en caso de que disponga de ella, haciendo imposible la reproducción de las condiciones de la excepción:

```
catch (IOException e) {
        logger.info("Excepcion de lectura de fichero "+e).
}
```

365. Para solventar esta situación, se debe modificar la forma en la que se almacena la información de la excepción para que se guarde toda la traza, pudiendo así reproducir de forma exacta dónde se está produciendo la excepción para ser capaces de tratarla adecuadamente.

```
catch (IOException e) {
        logger.info("Excepcion de lectura de fichero",e).
}
```

11.9. CIFRADO

366. Toda la información sensible debe ser manejada con la debida seguridad en todas las aplicaciones, pero especialmente en las aplicaciones web, tanto en el modo en el que se transmite la información como en la manera en la que se almacena. Para ello, en el caso de los sistemas de trasmisión de información en una aplicación web, está deberá utilizar un canal seguro de comunicaciones, implementado mediante el despliegue y uso del protocolo HTTPS (*Hypertext Transfer Protocol Secure*), basado en HTTP; este protocolo utiliza un sistema de cifrado basado en SSL (*Secure Sockets Layer*) y TLS (*Transport Layer Security*). La transferencia de información entre servidor web y el navegador del cliente se realiza por un canal cifrado que evita que los datos puedan ser interpretados si se realiza una captura del tráfico de red entre los dos puntos.

367. De esta manera, el uso de HTTPS no garantiza que la lógica de la aplicación sea segura, sino que el medio por el que se envía la información es seguro y no puede ser capturado mediante técnicas de interceptación de información.

368. En el caso de necesitar almacenar información sensible, no es suficiente con incorporar un mecanismo seguro de transmisión de información sino que se debe incluir un sistema que permita tratar estos datos con las medidas suficientes que garanticen su integridad, disponibilidad y confidencialidad. Esta situación se consigue mediante el uso del cifrado. Existen múltiples métodos de cifrado de la información, y en función de las necesidades de la aplicación desarrollada y de la información tratada, se deberán emplear mecanismos de cifrado simétrico, en el que la clave de cifrado es la misma que la de descifrado, mecanismos de cifrado asimétrico, en el que se usa una clave para cifrar y una diferente para descifrar o mecanismos de resumen, en los que no es posible recuperar la información original que ha sido cifrada.

11.10. ACCESO A BASES DE DATOS

11.10.1. OBTENCIÓN DE CONEXIONES

369. Se debe evitar la obtención de conexiones en bucles que puedan provocar una carga excesiva o incluso una saturación de los recursos computacionales disponibles, tanto de la base de datos de *backend* como de máquina en la que se está ejecutando la aplicación. Además, la obtención de conexiones debe gestionarse desde un único lugar de la aplicación con el fin de facilitar el mantenimiento y evitar posibles errores.

370. Supongamos una aplicación que hace un uso intensivo de conexiones a base de datos para implementar su lógica de negocio y que tiene repartida por el código la obtención de conexiones, del modo siguiente:

```
Connection con = DriverManager.getConnection(url, user, pass);
```

371. Tanto por cuestiones de mantenimiento evolutivo como correctivo, incluyendo aspectos de corrección de problemas de seguridad (por ejemplo imaginemos una nueva vulnerabilidad en la obtención de conexiones a través de la clase *DriverManager*), se puede provocar una revisión completa del código de la aplicación para localizar los puntos en los que la aplicación es vulnerable, pudiendo generar pérdidas de información si no se realiza una correcta modificación en todos los puntos en los que se están obteniendo las conexiones. Así, una implementación más segura y con un mejor mantenimiento sería la expuesta a continuación:

```
public Connection getConnection (final String sUrl, final String sUser, final String sPass) throwsDBException {
        try {
                Class.forName(System.getProperty("driver.classname"));
        } catch (Exception ex) {
                throw new DBException("No se pudo cargar el driver jdbc");
        }
        return DriverManager.getConnection(sUrl, sUser, sPass);
}
```

11.10.2. CIERRE DE CONEXIONES

372. Debe garantizarse en todo momento que las conexiones abiertas contra la base de datos son convenientemente cerradas, por ejemplo incluyendo código dentro de bloques *try/catch/finally* que garanticen esta situación. La gestión incorrecta del cierre de las conexiones puede llegar a desbordar el máximo reservado de conexiones para la aplicación y poner en riesgo así la disponibilidad del sistema, e incluso en casos extremos la integridad de los datos.

373. Por ejemplo, en el código expuesto a continuación, en caso de que la lógica de la aplicación lance alguna excepción la conexión no se cierra de forma adecuada, lo que puede provocar que el número de conexiones realizadas a la base de datos aumente sin control causando problemas de rendimiento en la aplicación, situación que finalmente puede ocasionar un problema de disponibilidad:

```
public void saveFile(String sFile, String sData) throwsException {
        Connection conn = ConnectionUtils.getConnection();
        File f = new File(sFile);
        FileUtils.saveFile(f, sData);
        conn.closeConnection();
}
```

374. Una implementación más segura del código anterior sería la siguiente, en la que se garantiza que la conexión se liberará de forma controlada independientemente del resultado de la lógica de la aplicación:

```
public void saveFile(String sFile, String sData) throwsException {
        Connection con = null;
        try {
                conn = ConnectionUtils.getConnection();
                File f = new File(sFile);
                FileUtils.saveFile(f, sData);
        } catch(Exception e) {
                throw new Exception("Error en el almacenamiento del fichero "+e);
        } finally {
                If (conn != null) {
                        try {
                                conn.closeConnection();
                        } catch(Exception ex) {
                                logger.error("Error liberando conexión", e);
                        }
                }
        }
}
```

11.10.3. CONSULTAS SQL

375. Se debe evitar el uso de literales en las consultas de base de datos del mismo modo que se debe evitar el uso de literales en el código, puesto que cualquier cambio en la estructura de datos exigiría la revisión completa del código fuente, lo que aparte de problemas en el mantenimiento puede ocasionar problemas de seguridad por errores en el código. Para corregir esta situación se debe disponer de un único punto en el que estén disponibles las sentencias y literales que se deben emplear para acceder a la base de datos, de forma que exista únicamente un punto de revisión y posible error, minimizando el impacto que puede ocasionar una modificación en la estructura de datos.

11.10.4. CONSULTAS PARAMETRIZADAS

376. Se debe evitar el uso de literales en las consultas medianteprocedimientos de base de datos. Por ejemplo, el siguiente código puede provocar un problema de seguridad, así como de legibilidad, puesto que una modificación en la consulta que se desea ejecutar puede provocar que no se ejecute la sentencia SQL deseada y que la aplicación sea víctima de un ataque de inyección SQL:

```
st.setString(7, "valor");
```

377. Una implementación más segura sería la expuesta a continuación, en la que el parámetro PARAM_VALUE es utilizado para indicar la posición en la que debe asignarse el valor de la variable, resultando más sencilla la gestión de modificaciones:

```
st.setString(PARAM_VALUE, "valor");
```

378. Adicionalmente esta aproximación permite la recuperación del valor sin necesidad de recordar la posición exacta del parámetro o la revisión de todo el código en caso de que se produzca un cambio en la consulta que se desea ejecutar.

379. Del mismo modo, cualquier parámetro que provenga de una fuente no confiable, como puede ser el navegador del usuario o un servicio web externo, debe ser incorporado a la sentencia SQL mediante el uso de prepared Statements, así como validar el tipo de dato esperado.

380. De esta forma no deberían existir situaciones como la que puede observarse en el siguiente fragmento de código, puesto que emplean un valor proveniente de la interfaz de usuario que puede contener código malicioso

```
String query = "SELECT * FROM usuario WHERE idusuario = "+request.getParameter("userId")
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

381. Una implementación más segura del anterior fragmento de código, utilizaría las consultas parametrizadas y validaría el tipo de dato esperado; en este caso, se asume que el campo idusuario es de tipo entero malicioso

```
String query = "SELECT * FROM usuario WHERE idusuario = ?"+request.getParameter("userId")
PreparedStatement stmt = con.prepareStatement(query);
stmt.setInteger(1, Integer.parseInt(request.getParameter("userId")));
ResultSet rs = stmt.executeQuery();
```

11.11. CATÁLOGO DE VULNERABILIDADES HABITUALES

382. Todos los entornos de desarrollo web y todos los tipos de aplicaciones web pueden presentar a priori vulnerabilidades (*bugs*), desde validaciones insuficientes hasta errores en la lógica de la aplicación, problemas de seguridad que repercuten en la integridad, confidencialidad o disponibilidad de la información tratada. Existen diferentes análisis, como los asociados a OWASP (Open Web Application Security Project) o al instituto SANS, para identificar el conjunto de vulnerabilidades más explotadas y habituales en los desarrollos web, entre las que destacan las expuestas en el presente apartado de la guía.

11.11.1. SECUENCIA DE COMANDOS EN SITIOS CRUZADOS (XSS)

383. Los errores de XSS se materializan cuando una aplicación toma la información provista por el usuario y la envía a un navegador sin validarla primero o codificar el contenido. El XSS permite a los atacantes ejecutar *scripts* en el navegador de la víctima que pueden provocar la captura de la sesión del usuario, la modificación de sitios web, la posible introducción de gusanos, etc.

384. Para mitigar este tipo de ataques se deben seguir las directrices de programación expuestas en los siguientes apartados de la presente guía:

385. 8.5.4 Entrada y salida

11.11.2. ERRORES DE INYECCIÓN

386. Las inyecciones, particularmente las inyecciones SQL, son comunes en las aplicaciones web debido a la utilización de los datos suministrados por los usuarios en consultas dinámicas o a una mala construcción de los procedimientos de acceso a datos en la base de datos. Estos ataques de inyección SQL permiten a los atacantes crear, leer, actualizar o borrar cualquier dato disponible por la aplicación y, en el peor escenario, comprometer completamente el sistema gestor de base de datos y todos los sistemas que lo utilizan.

387. Para mitigar este tipo de ataques se deben seguir las directrices de programación expuestas en los siguientes apartados de la presente guía:

388.8.5.4.3 Validación de tipos

389.8.5.4.4 Validación de datos con listas blancas

390.8.5.4.5 Validación de campos ocultos

391.8.11.3 Consultas SQL

392.8.11.4 Consultas parametrizadas

11.11.3. EJECUCIÓN MALICIOSA DE ARCHIVOS

393. El código vulnerable a inclusión remota de archivos permite a un atacante incluir código y datos hostiles, resultando en ataques que pueden incluso comprometer la seguridad global del entorno donde se ejecuta la aplicación. Los ataques de ejecución maliciosa de archivos afectan a ficheros con contenido XML y cualquier *framework* que acepte nombres de archivos o archivos desde los usuarios.

394. Para mitigar este tipo de ataques se deben seguir las directrices de programación expuestas en los siguientes apartados de la presente guía:

395.8.5.4.3 Validación de tipos

396.8.5.4.4 Validación de datos con listas blancas

397.8.5.4.5 Validación de campos ocultos

398.8.8 Sistema de archivos

11.11.4. REFERENCIA DIRECTA E INSEGURA A OBJETOS

399. Una referencia directa e insegura a objetos se produce cuando un desarrollador expone una referencia a una implementación interna de un objeto como un archivo, un directorio, un registro de BD o una clave, en forma de parámetro de URL o equivalente: los atacantes pueden manipular esas referencias para acceder a otros objetos sin tener autorización

400. Para mitigar este tipo de ataques se deben seguir las directrices de programación expuestas en los siguientes apartados de la presente guía:

401.8.5.3 Acceso y Herencia

402.8.5.4 Entrada y salida

11.11.5. FALSIFICACIÓN DE PETICIÓN EN SITIOS CRUZADOS (CSRF)

403. Un ataque CSRF fuerza al navegador web validado de una víctima a enviar una petición a una aplicación web vulnerable, la cual realiza la acción elegida a través de la víctima; al contrario que en los ataques XSS, los cuales explotan la confianza que un usuario tiene en un sitio en particular, el CSRF explota la confianza que un sitio tiene en un usuario en particular.

404. Para mitigar este tipo de ataques se deben seguir las directrices de programación expuestas en los siguientes apartados de la presente guía:

405.8.5.4 Entrada y salida

11.11.6. REVELACIÓN DE INFORMACIÓN Y GESTIÓN INCORRECTA DE ERRORES

406. Las aplicaciones pueden revelar información sobre su configuración, funcionalidad interna o violar la privacidad a través de una variedad de problemas asociados al código, información que es usada por atacantes para robar datos delicados o como apoyo a ataques más serios.

407. Para mitigar este tipo de ataques se deben seguir las directrices de programación expuestas en los siguientes apartados de la presente guía:

408. 8.9 Auditoría y registro

11.11.7. AUTENTICACIÓN Y GESTIÓN DE SESIONES INCORRECTOS

409. Este ataque se materializa debido a que las credenciales de las cuentas e identificadores de sesión no se protegen adecuadamente y los atacantes roban estos datos para suplantar la identidad de otros usuarios.

410. Para mitigar este tipo de ataques se deben seguir las directrices de programación expuestas en los siguientes apartados de la presente guía:

411. 8.3 Identificación, autenticación y autorización.

11.11.8. ALMACENAMIENTO CRIPTOGRÁFICO INSEGURO

412. No es habitual que las aplicaciones web usen funciones criptográficas adecuadamente, con objeto de proteger información o credenciales; así, si un atacante consigue acceso a esta información mal protegida, puede usarla para cometer robos de identidad o fraudes, entre otros.

413. Para mitigar este tipo de ataques se deben seguir las directrices de programación expuestas en los siguientes apartados de la presente guía:

414. 8.10 Cifrado

11.12. HERRAMIENTAS DE UTILIDAD

415. Para ayudar y controlar el buen trabajo del equipo de desarrollo de aplicaciones, es necesario integrar, en el entorno y procedimientos operativos correspondientes, el uso de herramientas como las expuestas a continuación u otras con funcionalidad equivalente, en función del entorno de desarrollo Java utilizado (estos ejemplos corresponden a JDeveloper):

11.12.1. FORMATEADORES DE CÓDIGO

416. Entornos de desarrollo como JDeveloper o Eclipse disponen de formateadores de código propios que corrigen la indentación de las clases seleccionadas, con el fin de disponer de un código más legible y trazable que asegure la correcta ejecución de las instrucciones deseadas. Existen otras alternativas que realizan esta misma funcionalidad, como Jalopy (http://sourceforge.net/projects/jalopy/).

11.12.2. PMD

417. Esta herramienta comprueba el código fuente Java y busca problemas potenciales, entre los que podemos destacar los siguientes:

418. Posibles bugs y sentencias try/catch/finally/switch vacíos.

419. Código muerto: variables, parámetros y métodos privados no usados.

420. Código no óptimo: uso inadecuado de nuevos objetos o de concatenaciones de Strings.

421. Código duplicado: posibles errores por código sospechosamente parecido.

422. Podemos obtener más información de la herramienta en la siguiente URL:

423. http://pmd.sourceforge.net/

11.12.3. FINDBUGS

424. Esta herramienta emplea un análisis estático de código con el fin de encontrar errores en el código Java, entre los que cabe destacar:

425. Error corregible. Detección de código que aparentemente puede conllevar a errores.

426. Malas prácticas. Detección del mal uso de determinadas expresiones relevantes.

427. Estilo. Detección de código especialmente confuso.

428. Podemos obtener más información en la URL siguiente:

429. http://sourceforge.net/projects/findbugs-jdev/

12. DIRECTRICES DE AUDITORÍA

12.1. INTRODUCCIÓN Y OBJETIVOS

430. Una auditoría de seguridad de un producto o una infraestructura de una organización tiene como objetivo detectar errores o debilidades que puedan suponer un riesgo para la seguridad del entorno analizado. Así, el equipo auditor valora, de forma objetiva y ciñéndose estrictamente a las evidencias encontradas, los riesgos de seguridad hallados, sin entrar en el debate de suposiciones o comentarios.

431. En todo momento, el equipo auditor debe compartir objetivos con todo el equipo de desarrollo y esos objetivos pasan, obviamente, por que el producto final ofrezca las mejores condiciones de calidad y seguridad posibles. Es muy importante que el personal auditor sea considerado una pieza más del proceso de desarrollo, una herramienta que ayuda a que el producto final sea de la mayor calidad posible, de la misma forma que el uso de un sistema de control de versiones ayuda a conseguir un mejor producto final evitando errores en el versionado del código.

12.2. DISEÑO DE AUDITORÍAS

432. La organización debe contemplar al menos la realización de auditorías de seguridad en dos fases diferenciadas pero que comparten el objetivo común indicado con anterioridad. Estas fases son las siguientes:

433. Auditoría ligada al desarrollo. Es aquella que se realiza embebida en el proceso de desarrollo de software, en el paso de cada una de las fases. Se realiza de forma asíncrona, cuando una de las etapas del desarrollo de una aplicación ha llegado a su fin y debe comenzar la siguiente.

434. Auditoría periódica de la plataforma. Es aquella que se realiza al finalizar la implantación del producto en producción y que deberá ejecutarse de forma periódica para detectar las vulnerabilidades publicadas durante el tiempo de vida del sistema, errores en el proceso de mantenimiento de la aplicación, etc.

435. Se detallan a continuación ambas aproximaciones a la auditoría de aplicaciones y desarrollos web:

12.2.1. AUDITORÍA LIGADA ALDESARROLLO

436. El proceso de desarrollo comprende etapas como la fase de análisis, diseño, implementación, etc. Para garantizar la máxima seguridad posible, como se ha indicado previamente en esta guía, la organización debe establecer mecanismos de control en el paso de una etapa a la siguiente, que requerirán que los datos de salida de una fase y entrada de la siguiente sean aprobados tanto por el responsable de la etapa saliente como por el responsable de la etapa entrante, así como por un equipo auditor de seguridad que revise que no se aprecian en la información aportada evidencias de incumplimiento del procedimiento de desarrollo seguro descrito en los apartados anteriores de este documento. Tanto el receptor del documento como el auditor de seguridad pueden rechazar el documento por una falta de información o por detectar potenciales riesgos de seguridad provenientes de la etapa anterior; dicha justificación es enviada al emisor del documento para que sea corregido, volviéndose a repetir el proceso. A efectos de registro las correcciones del documento deben quedar trazadas por medio de un control de versiones, ya sea en el propio documento o mediante herramientas que garanticen la trazabilidad. De esta manera, se consigue introducir el proceso de auditoría en todas las fases del ciclo de desarrollo del *software*.

437. La organización debe definir al menos las siguientes fases de auditoría ligadas al desarrollo de la aplicación:

438. Paso de **análisis a diseño**. El equipo auditor es el encargado de revisar que la información del análisis no incluye casos de uso que puedan poner en riesgo la seguridad de la aplicación; así mismo, debe comprobar que el análisis no incluye requerimientos no compatibles con la legislación vigente, para lo que puede requerir soporte del departamento legal o equivalente de la organización.

439. Paso de **diseño a desarrollo**. El equipo auditor es el encargado de revisar el diseño de la aplicación con el fin de localizar elementos potencialmente peligrosos para la seguridad previamente a que comiencen a ser implementados. Dicha revisión comprende las siguientes etapas:

440. Revisión de completitud de los casos de abuso detectados durante la fase de diseño.

441. Revisión del modelo de amenazas empleado y del posterior análisis de riesgos y selección de contramedidas, en función de la criticidad de la información manejada o de las exigencias regulatorias aplicables a la aplicación o a la organización en su conjunto.

442. Revisión de mecanismos de *backup* que garanticen un correcto y seguro almacenamiento de la información necesaria para restablecer el servicio en caso de contingencia. El equipo auditor debe comprobar que los elementos necesarios para el servicio que presta o va a prestar la aplicación están integrados de forma correcta en el sistema de copias corporativo.

443. Revisión de mecanismos de transmisión de la información y cifrado de comunicaciones.

444. Revisión de mecanismos de almacenamiento de la información y cifrado de ficheros o sistemas de archivos.

445. Revisión de la estrategia de autenticación de los usuarios y la gestión de sus respectivos roles y privilegios.

446. Revisión de la gestión de sesiones desde el punto de vista de su fortaleza contra el robo y la persistencia de sesiones abiertas.

447. Revisión de especificaciones en el manejo de errores y mecanismos de registro del sistema, elementos que deben proporcionar la suficiente información para identificar y analizar ataques realizados sobre la aplicación o, en general, incidentes de seguridad.

448. Revisión del diseño de roles para los diferentes perfiles de usuario de la aplicación, siguiendo en todo momento el principio de mínimo privilegio y máxima separación de tareas y granularidad.

449. Paso de **desarrollo a pruebas**. El equipo auditor es el encargado de realizar una verificación de la eficacia de los controles diseñados en las fases anteriores, por medio de una auditoría de caja negra y caja blanca de la aplicación (para más información técnica en este sentido, se puede consultar el punto relativo a ejecución de auditorías de la presente guía CCN-STIC). Esta etapa también sirve para verificar el correcto funcionamiento del sistema de registro de anomalías de seguridad registradas en los logs de la aplicación, que deben ser suficientemente significativos como para trazar las actividades sospechosas y determinar tanto su éxito o fracaso como su impacto.

450. Paso de **pruebas a producción**. El equipo auditor es el encargado de analizar el entorno de producción para garantizar que cumple los requisitos necesarios para ofrecer el servicio de forma segura. Esta etapa también sirve para verificar el correcto funcionamiento de los controles de detección de intrusos, así como la calidad de la información registrada por los logs de seguridad de los sistemas del entorno (servidores de aplicación, de bases de datos…) en el que se ejecuta la aplicación desarrollada. En gran medida, esta fase de auditoría se corresponderá con la auditoría periódica de la plataforma.

12.2.2. AUDITORÍA PERIÓDICA DE LA PLATAFORMA

451. Una vez implantada la aplicación en el entorno de producción o actualizado éste a una nueva *release*, su análisis de seguridad entra dentro de la auditoría periódica de sistemas que debe estar implantada en la organización y que tiene como objetivo verificar que la visibilidad, nivel de parcheo, configuración de los servicios de los sistemas que albergan el servicio o aplicación, etc. son los adecuados.

452. Aquellas vulnerabilidades producidas por errores en el mantenimiento de la aplicación o por fallos en el entorno detectados por los fabricantes con posterioridad al momento de la implantación de la aplicación deben ser detectadas y corregidas en el menor plazo de tiempo posible; para ello la organización debe no únicamente establecer un modelo de auditoría periódica de vulnerabilidades contra la aplicación y los entornos que la soportan –y por extensión, contra todos los sistemas de la organización-, sino además mantenerse informada de nuevos problemas y vulnerabilidades en los elementos del entorno tecnológico corporativo, por ejemplo mediante la suscripción del personal de seguridad a listas de correo, foros, alertas de fabricantes, etc. en los que se traten temas que puedan afectar a la seguridad de la organización en su conjunto. Obviamente, el detalle de estos aspectos queda fuera del ámbito de la presente guía.

453. Independientemente de la auditoría de los sistemas y de la frecuencia de liberación de nuevas *releases* de la aplicación, es necesario realizar de forma periódica, al menos una vez al año, una auditoría completa del sistema, incluyendo la aplicación, con el fin de detectar vulnerabilidades desconocidas en el momento de realizar la correspondiente auditoría durante el ciclo de desarrollo del producto.

12.3. EJECUCIÓN DE AUDITORÍAS

454. Aunque la planificación, desarrollo y proceso de resultados asociados a las auditorías de seguridad queda fuera del ámbito de la presente guía, es necesario marcar unas pautas mínimas que el equipo auditor debe cumplir en todos los casos y modelos de análisis, tanto los realizados en el desarrollo de la aplicación como los ejecutados contra la plataforma que le da soporte. La organización debe definir e implantar un procedimiento operativo de auditoría técnica que abarque al menos los aspectos que aquí se indican, sin menoscabo de otros cualesquiera de aplicación concreta en una organización o desarrollo determinados.

455. Tal y como se ha indicado previamente, existen dos fases de auditoría que abarcan el ciclo de vida completo del desarrollo del producto. En cada una de ellas, en especial en la auditoría contra la plataforma, el equipo auditor debe analizar al menos los siguientes aspectos de la seguridad de la aplicación, derivados del framework de auditoría web OWASSP y la guía de desarrollo seguro en su versión 2.0 de la misma entidad:

456. Recopilación de información. El equipo de auditoría debe recopilar tanta información como sea posible sobre la aplicación objeto del análisis, de forma independiente a la ofrecida por la organización y haciendo uso de herramientas de acceso público: motores de búsqueda, analizadores de vulnerabilidades, peticiones HTTP simples, peticiones especialmente diseñadas… El objetivo de esta tarea es obtener la mayor cantidad de información sobre la aplicación auditada por medios ajenos a la organización, de forma que se determine a qué datos relevantes puede tener acceso un potencial atacante.

457. Pruebas de gestión de configuración de la infraestructura. Con frecuencia los análisis sobre la infraestructura o la topología de la arquitectura pueden revelar datos importantes sobre una aplicación web: código fuente, métodos HTTP permitidos, funcionalidades administrativas, métodos de autenticación, configuraciones de la infraestructura… El equipo de auditoría debe identificar las posibles vulnerabilidades de la aplicación, en especial puntos de entrada del sistema, mediante esta información.

458. Comprobación del sistema de autenticación. Dado el carácter crítico de la información gestionada por muchas aplicaciones web, es necesario comprobar que el proceso de verificación de identidades se realiza de forma segura y no caben ataques de usurpación o falsificación de identidad. Para ello se debe establecer en primera instancia si se están utilizando canales seguros de comunicación, para a continuación determinar la posibilidad de enumerar usuarios del sistema y, si existen sistemas de recordatorio o recuperación de credenciales, revisarlos convenientemente.

459. Gestión de las sesiones. El núcleo de toda aplicación web se encuentra en el sistema en que la aplicación mantiene los estados, y por lo tanto controla la interacción del usuario con el *site*. La gestión de sesiones cubre ampliamente todos los controles que se realizan sobre el usuario, desde la autenticación hasta la salida de la aplicación. El equipo de auditoría debe comprobar todos aquellos estados vulnerables que podrían ser aprovechados por un usuario ilícito para adquirir información o ganar privilegios en el sistema.

460. Gestión de las autorizaciones. Las pruebas de autorización significan entender cómo funciona el proceso de autorización y usar esa información para evitar el mecanismo establecido. La autorización es un proceso que llega después de una autenticación correcta, por lo que el equipo analista debe verificar este punto después de obtener credenciales válidas, asociadas a un conjunto de perfiles y privilegios bien definidos. Durante este tipo de evaluaciones, debe verificarse al menos si es posible evitar el sistema de autorización, encontrar una vulnerabilidad de traspaso de rutas o encontrar maneras de escalar los privilegios asignados al evaluador.

461. Comprobación de la lógica de negocio. Los ataques sobre la lógica de negocio de una aplicación son peligrosos, difíciles de detectar y específicos a la aplicación; por este motivo el equipo de auditoría debe analizar de forma particularizada el funcionamiento lógico de la aplicación y del entorno que la soporta, determinando todos aquellos estados inconsistentes o potencialmente peligrosos para el sistema, los usuarios o la organización.

462. Validación de los datos de entrada. Una de las debilidades más comunes en la seguridad de aplicaciones web es la falta de una validación adecuada de las entradas procedentes del cliente o del entorno de la aplicación. Esta debilidad conduce a casi todas las principales vulnerabilidades en aplicaciones, como inyecciones sobre el intérprete, ataques locale/Unicode, ataques contra el sistema de archivos o desbordamientos de búfer, por citar unas cuantas. De esta manera el equipo de auditoría, tras hacer uso de la información recopilada en fases anteriores, debe tratar de explotar las vulnerabilidades encontradas, cruzando en todo momento la información con el equipo de operación y la plataforma para garantizar el mínimo impacto de las pruebas en la organización, en caso de que la aplicación esté en producción.

463. La periodicidad de la auditoría contra la plataforma que da soporte a la aplicación desarrollada no debe ser superior a un mes, mientras que la auditoría ligada al desarrollo debe planificarse con una periodicidad no superior a un año. El proceso de auditoría debe ser realizado por un equipo ajeno al proyecto de desarrollo, ya sea propio de la organización o externo a ésta, para garantizar de esta forma tanto una correcta segregación de funciones como la independencia del equipo auditor.

464. Para realizar cualesquiera análisis de seguridad, el equipo auditor debe mantener la última versión de las herramientas utilizadas en cada caso, con el fin de poder abordar las auditorías garantizando que el análisis se ejecutará verificando los últimos tipos de ataques y *exploits*, obteniendo así la máxima exactitud del análisis realizado. Además, y de cara a garantizar el mínimo impacto de la auditoría, la organización debe definir un horario de mínimo impacto que permita ejecutar las tareas necesarias en cada análisis.

ANEXO A. GLOSARIO

API. ApplicationProgramming Interface.

CSRF. Cross-SiteRequestForgery.

DoS. Denial of Service.

DDoS.DistributedDenial of Service.

HTML. HyperTextMarkupLanguage.

HTTP. HyperText Transfer Protocol.

HTTPS. SecureHyperText Transfer Protocol.

IP. Internet Protocol.

JS. Javascript.

NTP. Network Time Protocol.

OWASP. Open Web Application Security Project.

SQL-i. SQL injection.

SSL. Secure Sockets Layer.

URL. UniformResourceLocator.

VBS. Visual Basic Script.

XSS. Cross Site Scripting.

ANEXO B. LISTAS DE COMPROBACIÓN

COMPROBACIÓN	ESTADO	OBSERVACIONES
Seguridad del proceso de desarrollo		
Política de Seguridad		
Se encuentran identificadas las políticas y procedimientos de seguridad corporativos.		
Especificación de requisitos de Seguridad		
Se encuentran identificados y validados los requisitos de seguridad de la aplicación con anterioridad a la implentación (fase de análisis).		
Están ajustados a la política de clasificación, marcado y tratamiento de la información en la organización.		
Existen repositorios de documentación que controlan el versionado de los documentos.		
Existe un apartado de Requisitos de Seguridad en el documento de Análisis de Requisitos.		
Control de Diseño y Programación		
El intercambio de código se realiza mediante un sistema de control de versiones.		
Existe control de acceso al código fuente de los desarrollos, así como control de privilegios de acceso.		
Se realizan auditorias periódicas de la integridad del código fuente.		
Los datos de prueba son ficticios pero disponen de las mismas características que los datos reales.		
Existen acuerdos de confidencialidad con terceros en caso de servicios externalizados.		
Existe un procedimiento de alta y baja de empleados.		
Existe entorno de trabajo diferenciados para desarrollo, preproducción y producción.		
El equipo de desarrollo no dispone de privilegios para subir un nuevo desarrollo fuera del entorno de desarrollo.		
Existen una separación formal de tareas y responsabilidades en el equipo del proyecto.		
Existe un procedimiento de paso a preproducción y producción.		
Se encuentra establecido un esquema de auditoría de código que se lanza al menos dos veces con anterioridad a la puesta en producción.		
Cada liberación de versión es registrada con un identificador que permite identificar la versión, así como un registro de cambios realizados.		
El proceso de mantenimiento contempla los mismos requisitos de seguridad que el proceso de desarrollo.		
Se realizan estimaciones aproximadas del riesgo de las tareas de mantenimiento (FCRA) y son analizadas periódicamente.		
Análisis de Casos de Abuso		
Se encuentran definidos y desarrollados casos de abuso que puedan desembocar en una degradación de calidad de servicio		

COMPROBACIÓN	ESTADO	OBSERVACIONES
Se revisan los casos de abuso ante cambios en los requisitos		
Se documentan los casos de abuso en un documento con el detalle del escenario asociado al caso.		
Análisis de Riesgos		
Existe una identificación de las amenazas, así como de sus correspondientes salvaguardas, en el ámbito del desarrollo.		
Requisitos legales de las aplicaciones		
La legislación aplicable a los datos tratados, en particular la relacionada con datos de carácter personal, ha sido identificada por el equipo de trabajo.		
Se garantiza que la aplicación desarrollada cumple los requisitos de seguridad en función del tipo de dato tratado.		
En caso de que existan empresas externas, éstas garantizan, con evidencias, que están al corriente de sus obligaciones de tratamiento de datos.		
El documento de seguridad de la organización indica las funciones y obligaciones de los usuarios con acceso a datos personales.		
Existe una relación actualizada de usuarios con el perfil de acceso de cada uno de ellos.		
Existen mecanismos de control de acceso a recursos no autorizados.		
Existen mecanismos que garantizan la correcta identificación y autenticación inequívoca de los usuarios		
Existen mecanismos que cifran las credenciales de usuario mediante un procedimiento de asignación, distribución y almacenamiento.		
Existen mecanismos que limitan la posibilidad de intentar reiteradamente el acceso no autorizado a la aplicación.		
Los soportes que permiten el paso de información se encuentran inventariados y su salida es registrada.		
Existe una normativa de uso de dispositivos móviles.		
Se incluyen los nuevos desarrollos en la política de copias de seguridad de la organización.		
Se realizan comprobaciones periódicas de los procedimientos de copia y restauración.		
Directrices de diseño		
Se incorporan controles para permitir únicamente acceso a los datos que el usuario está autorizado.		
Se incorporan controles para garantizar que los datos no son alterados por usuarios no autorizados.		
Se incorporan controles para garantizar que los sistemas y datos se encuentran disponibles cuando se necesitan.		
Directrices de despliegue		
Existe un procedimiento operativo para las tareas de despliegue de nuevas releases o funcionalidades de una aplicación.		
Existe un responsable funcional de cada uno los elementos de la estructura organizativa definida para los despliegues.		
Los despliegues los realiza el personal del equipo de explotación.		

COMPROBACIÓN	ESTADO	OBSERVACIONES
Existe un equipo de soporte (presencial o remoto) para el personal de explotación.		
Existe un procedimiento estandarizado y formal de control de cambios de estado.		
Han sido revisadas las configuraciones por defecto de los servidores de aplicaciones sobre los que se despliega la aplicación.		
Se han eliminado los ejemplos y páginas de prueba existentes por defecto en los servidores de aplicaciones.		
Se han aplicado las medidas adecuadas de bastionado de los servidores.		
Se realizan análisis de seguridad periódicos tanto de los nuevos desarrollos como de los existentes.		
Se realizan verificaciones manuales que garanticen la seguridad global de la aplicación y de la infraestructura que la soporta.		
Directrices de programación (JAVA)		
La codificación de la aplicación cumple los estándares de calidad existentes.		
Se limita el acceso a los diferentes atributos y métodos de las clases.		
Se validan los valores de los parámetros, tanto en tamaño como en tipo.		
Se vuelca la información de las excepciones a un sistema de logado para su posterior auditoría y análisis en caso de necesidad.		
Se evita el uso de la salida estándar para devolver información de registro.		
Se indentan correctamente el código de forma que se facilite la lectura siguiendo las convenciones de codificación JAVA.		
Se incorpora un mecanismo de bloqueo de cuentas de usuario para mitigar ataques de fuerza bruta o un mecanismo de notificación.		
Existe un mecanismo de bloqueo de orígenes.La sesión de un usuario autenticado se debe mantener durante la validez de ésta evitando reiteradas solicitudes de autenticación.		
La sesión dispone de un tiempo de caducidad asociado.		
Se elimina la cookie de sesión ante el cierre de sesión por parte del usuario o por una caducidad de sesión.		
El proceso de autenticación reemplaza la cookie del usuario anónimo, en caso de que ésta existiese.		
El tamaño de las clases desarrolladas es reducido y su funcionalidad se encuentra acotada.		
El tamaño de los métodos de las clases es limitado.		
Se evita el uso de wildcards en la sección de imports de las clases, así como clases no utilizadas.		
Los atributos de las clases que puedan ser modificados son definidos como privados y tienen métodos get y set preparados para su acceso y modificación.		
Los atributos de las clases que deben ser tratados como constantes, están definidos como estáticos y finales.		

COMPROBACIÓN	ESTADO	OBSERVACIONES
Las variables cuyo contenido es mutable (por ejemplo, mapas o listas) devuelven copias de estos objetos, salvo que su contenido pueda ser modificado por una entidad externa.		
La envoltura de métodos nativos (JNI) están definidos como privados y existe (al menos) un método público que se encarga de realizar las validaciones oportunas.		
Existe una clase de utilidades en las que se encuentran definidos los literales de la aplicación.		
Las comparaciones con strings comprueban que el contenido del primer elemento de la comparación no puede ser nulo, especialmente cuando se trata de constantes.		
Únicamente existe un punto de retorno para asegurar un retorno controlado del valor.		
Los métodos invocados desde el constructor se encuentran definidos como privados o como finales.		
Se limita la visibilidad de las clases, así como sus métodos para que únicamente sea público el API necesario.		
Aquellas clases cuyo propósito no puede ser sobrescrito se encuentran definidas como finales.		
Las operaciones sobre parámetros de entrada se realizan con copias de estas variables, salvo que sea necesaria su modificación.		
Las copias sobre los elementos de un array se realizan de forma manual.		
Las operaciones sobre parámetros de salida que son mutables devuelven una copia del objeto, salvo que deba devolver directamente la instancia del objeto.		
Las implementaciones del método clone invocan a super.clone()		
Los datos de entrada se validan tanto en contenido como en tipo esperado, revisando especialmente los campos ocultos y valores desplegables.		
Se utilizan listas blancas para la validación del contenido de los valores de entrada, así como de las constantes que pueden ser enviadas.		
En la lógica que representa la información se escapan y validan los datos a representar.		
Las excepciones son capturadas para eliminar la información sensible que es mostrada.		
Existen clases de excepción distintas a Exception que encapsulan la excepción esperada.		
No se utilizan rutas públicas que puedan ser accedidas por otras aplicaciones para almacenar información en el sistema de archivos.		
No se utilizan rutas absolutas para el almacenamiento de los datos en el sistema de archivos.		
No se utilizan nombres fijos para almacenar en archivos información enviada por los usuarios.		
No se sobrescribe el contenido de las variables utilizadas en el cuerpo del método para garantizar la trazabilidad.		

COMPROBACIÓN	ESTADO	OBSERVACIONES
El objeto de logado debe ser estático en cada clase e incorporar los atributos final y private.		
Se almacena toda la traza de la excepción y no únicamente su descripción.		
Se utiliza un canal seguro de comunicaciones.		
La información sensible almacenada ha sido cifrada para garantizar su confidencialidad.		
La conexión a la base de datos se gestiona desde un único punto de la aplicación que encapsula la obtención de una conexión.		
Se incorporan los mecanismos para garantizar el cierre de las sesiones a la base de datos, incluso en situaciones no controladas.		
Las consultas SQL se encuentran centralizadas y disponibles en un único fichero siendo accesibles en los puntos de la aplicación que lo necesitan.		
Las consultas que se generan con datos de entrada de fuentes no confiables verifican el tipo de datos introducidos, así como su contenido.		
Directrices de auditoría		
Se encuentran diseñados los procesos de auditoría en el paso de fases del ciclo de vida del desarrollo: análisis, diseño, implementación, pruebas.		
Se encuentran diseñadas las auditorías periódicas de la aplicación para detectar vulnerabilidades durante el tiempo de vida del sistema.		

ANEXO C. REFERENCIAS

Software Security: Building Security in. Gary McGraw. Addison Wesley Software Security Series. 2006.

Using Abuse Case Models for Security Requirements Analysis. John McDermott and Chris Fox. Computer Security Applications Conference (ACSAC 99). 1999.

Abuse-Case-BasedAssuranceArguments. John McDermott. Computer Security Applications Conference (ACSAC 2001). 2001.

CWE/SANS TOP 25 Most Dangerous Software Errors. http://cwe.mitre.org/

Exploiting Software. Greg Hoglund and Gary McGraw. Addison-Wesley. 2004.

Security Requirements Engineering: When Anti-requirements Hit theFun. Robert Crook, DarrelInce, LunchengLin, BasharNuseibeh. In Proc. of RE'02, IEEE Press. 2002.

19 Deadlysins of Software Security: Programming flaws and howtofixthem. Michael Howard, David LeBlanc and John Viega. McGraw Hill. 2005.

Java SE Security. Oracle. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html

OWASP. https://www.owasp.org/