Guía

CÓMO CONSTRUIR *SOFTWARE* SEGURO PARA MI *START-UP*

Seguridad en la construcción de *software* desde la perspectiva del ahorro de costes y tiempo

Financiado por la Unión Europea
NextGenerationEU

GOBIERNO DE ESPAÑA
MINISTERIO PARA LA TRANSFORMACIÓN DIGITAL Y DE LA FUNCIÓN PÚBLICA
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL

Plan de Recuperación, Transformación y Resiliencia

incibe_
INSTITUTO NACIONAL DE CIBERSEGURIDAD

incibe emprende

Índice

1. Introducción ¿Por qué esta guía? 06

2. La filosofía "*pushing left*" en el mundo de las start-up 08

3. Requisitos de seguridad desde la perspectiva de la agilidad y la economía de recursos 09

3.1. RS1 Requisitos de validación de entradas..... 12

3.1.1. RS1.1 No confiar en ningún dato que nos llegue 12

3.1.2. RS1.2 Hacer toda la validación en el servidor 19

3.1.3. RS1.3 Codificar todas las salidas de la aplicación 22

3.2. RS2 Requisitos de cifrado de la información manejada 23

3.2.1. RS2.1 Cifrar todos los datos sensibles guardados 24

3.2.2. RS2.2 Cifrar todos los datos "en tránsito" 24

3.2.3. RS2.3 Guardar adecuadamente las contraseñas (si es necesario guardarlas) 24

3.2.4. RS2.4 Hacer obligatorio acceder vía HTTPs a todos los sitios web 26

3.2.5. RS2.5 Asegurarse de que se usa la última versión de TLS para HTTPs 27

3.3. RS3 Requisitos de gestión de dependencias de *software* de terceros que necesitemos para hacer una operación 28

3.3.1. RS3.1 Establecer una política de comprobación de código de terceros 28

3.3.2. RS3.2 Minimizar la "superficie de ataque" 31

3.4. RS4 Requisitos de gestión de datos 33

3.4.1. RS4.1 Clasificar y etiquetar todos los datos de la aplicación 33

3.4.2. RS4.2 Guardar todos los secretos de la aplicación en una *secret store* 36

3.4.3. RS4.3 Prohibir terminantemente el "*hardcoding*" de secretos 37

3.4.4. RS4.4 Asegurarse de que no hay ningún dato sensible en los comentarios de la aplicación 37

3.4.5. RS4.5 Usar solo *prepared statements* (con *parametrized queries*) para el acceso a datos 38

3.4.6. RS4.6 No pasar datos importantes en los parámetros de una URL 39

3.5. RS5 Requisitos tecnológicos y de revisión del *software* 39

3.5.1. RS5.1 Usar todas las cabeceras de seguridad aplicables si se trata de una aplicación que use el protocolo HTTP 39

3.5.2. RS5.2 Usar una configuración de cookies segura 42

3.5.3. RS5.3 Usar las características de seguridad de los *frameworks* elegidos 44

3.5.4. RS5.4 Usar la última versión estable, soportada y actualizada de los *frameworks* 44

3.5.5. RS5.5 Mantener las dependencias actualizadas 45

3.5.6. RS5.6 Asegurarse de que se usan herramientas de seguridad en la aplicación antes de su puesta en producción 45

3.5.7. RS5.7 Hacer *code review* de la aplicación antes de que sea puesta en producción 46

3.5.8. RS5.8 Asegurarse de que la aplicación captura todos los errores 47

3.5.9. RS5.9 Deshabilitar el cacheo en páginas que contienen información privada 47

3.6. RS6 Requisitos de cuentas de usuario, autenticación y autorización 49

3.6.1. RS6.1 Asegurarse de que todas las cuentas usadas en la aplicación son cuentas de servicio 49

3.6.2. RS6.2 Fomentar que todos los usuarios de la aplicación usen *password managers* y prohibir la reutilización de claves 49

3.6.3. RS6.3 Forzar el uso de claves largas (*pass phrases*) y con un mínimo de complejidad 50

3.6.4. RS6.4 Verificar que la clave de un usuario no ha sido ya filtrada usando un servicio externo que se encarga de recopilar filtraciones 50

3.6.5. RS6.5 Activar MFA en todas las cuentas consideradas importantes 51

3.6.6. RS6.6 Requisitos de cambio de claves 54

3.6.7. RS6.7 Definir roles de usuario, asignárselos a los mismos y definir los permisos asociados a esos roles 54

3.6.8. RS6.8 Fijar un único método de autenticación y gestión de identidades para toda la aplicación . 57

3.6.9. RS6.9 Forzar siempre el mínimo privilegio para todas las cuentas de usuario o servicio 58

3.6.10. RS6.10 Habilitar copiar / pegar en los elementos de interfaz que usan *passwords* 59

3.7. RS7 Requisitos de funcionalidades particulares 60

3.7.1. RS7.1 Hacer *threat modeling* de la aplicación 60

3.7.2. RS7.2 Evitar si es posible la funcionalidad de subida de archivos 60

3.7.3. RS7.3 Hacer *backups* periódicos de todo el código y recursos de desarrollo 61

3.8. RS8 Requisitos de log o de registro de acciones que han ocurrido en la aplicación 63

3.8.1. RS8.1 Registrar todos los intentos de acceso 63

3.8.2. RS8.2 Si se detectan accesos indebidos, lanzar alertas de seguridad 63

4. Elementos de diseño seguro 64
4.1. Modelado general de las operaciones de la aplicación 64
4.1.1. Diseño de la protección de datos personales 64
4.1.2. Protección de filtraciones 66
4.1.3. Filosofía de diseño general 67
4.1.4. Diseño de APIs 67
4.1.5. Aislamiento por diseño 68
4.2. Estructura segura del código de la aplicación 68
4.2.1. Sobre la posible arquitectura de la aplicación 68
4.2.2. Prevención de posibles ataques en tiempo de diseño 71
4.3. La importancia del *threat modeling* 72

5. Aspectos clave para la creación de código seguro 76
5.1. Selección de los *framework* a usar o su versión concreta 76
5.2. Gestión de PII y secretos de la aplicación 77
5.3. Algoritmo general para implementar la validación de datos 78
5.3.1. Listas de elementos permitidos vs Listas de elementos bloqueados 80
5.3.2. Formas de implementar la validación 81
5.3.3. Sobre la codificación de datos 82
5.3.4. "Saneamiento" de la entrada HTML (*HTML sanitizer*) 85
5.3.5. Plantillas de codificación automática para contextos web 86
5.4. Autenticación con servicios de terceros 87
5.4.1. Implementación de MFA 89
5.5. Autenticación sin servicios de terceros 90
5.6. Manejo de sesiones si la aplicación es web 91
5.7. Implementación adecuada de los *logs* de la aplicación 92
5.8. *Checklist* de validación de prácticas de creación de código seguro 94
5.9. Aspectos básicos de usabilidad 99
5.10. Codificando centrados en seguridad: Atacando los errores más comunes 100

6. Integración rápida de herramientas de AST (*Application Security Testing*) en tu proceso de producción de *software* 102
6.1. Usar herramientas SCA 102
6.2. Herramientas SAST 103
6.3. Herramientas DAST 106

7. Más allá: Salir a producción con unas mínimas garantías de seguridad 109

7.1. Escaneo periódico de vulnerabilidades 109

7.1.1. *Nessus* 111

7.1.2. *OpenVAS* 111

7.2. Protección en nube 113

7.2.1. El efecto positivo de contar con una CDN 113

7.2.2. *API Gateway* 115

7.3. Protección *on-premises* 116

7.3.1. WAFs 116

7.3.2. *Proxies* inversos 117

7.3.3. Vigilancia con productos gratuitos: XDR/SIEM *Wazuh* 118

8. Referencias 121

1. Introducción
¿Por qué esta guía?

Si estás leyendo esta guía significa que, muy probablemente, está en tus planes **crear una *start-up*** para intentar cubrir con éxito una necesidad en el mercado, crecer y expandir sus posibilidades a medida que vas captando nuevos clientes. Y que, como parte de ese proceso, **vas a desarrollar un producto *software***. Lo primero de todo es felicitarte y desearte toda la suerte posible en esta tarea.

Ya sea ese *software* el centro de tu negocio o una parte de este, si te planteas crecer e ir consiguiendo un "nombre" y una reputación en el mercado que te permita seguir adelante con tu *start-up*, **tienes que distinguirte de la competencia**, especialmente al principio.

> ***No pienses que la distinción de la competencia es solo por la funcionalidad de tu producto, también tiene mucho que ver la calidad de los servicios que ofreces. Y la seguridad es calidad.***

En este contexto, una de las cosas que no te puedes permitir es una **pérdida de reputación o económica debida a un fallo de seguridad** que comprometa tu servicio o los datos de tus clientes. Tratar de evitar fallos de seguridad requiere una acción multidisciplinar, que conlleva mejorar la seguridad no solo tus **máquinas**, sino también concienciar a tus **clientes y empleados**. Pero hay una tercera "pata" que es en la que nos vamos a centrar en esta guía: **el desarrollo de tu *software* debe ser seguro**.

Por tanto, el desarrollo de un *software* con seguridad integrada desde sus primeras etapas es una forma de **mitigar riesgos**, pero también de lograr una **distinción de la competencia** qué puede ayudar a tus ventas.

> ***No desprecies este aspecto. Bien por los rigores de un calendario demasiado apretado (lanzar el producto lo antes posible para poder tener un efecto disruptivo en el mercado), o por falta de formación específica, es posible que tus competidores no puedan ofrecer un software con un nivel de seguridad adecuado o equiparable al tuyo.***

En definitiva, lo que vas a encontrar en esta guía es una descripción introductoria al proceso de construcción segura de *software*, que planteará distintas opciones de actuación en las distintas fases de construcción teniendo en cuenta que está enfocada en el contexto de una *start-up*.

> ***Ten en cuenta además que caer en la contradicción de ofrecer un producto de ciberseguridad sin que el producto sea en si seguro puede acabar fácilmente con tu reputación y tu negocio...***

El propósito de esta guía es precisamente ese: Evitar riesgos que puedan comprometer la idea de tu *start-up* y que puedas presentar tu producto diciendo que sigues unos determinados estándares y normas de seguridad. Para ello te guiará en el **proceso de desarrollar un *software* de forma segura**, pero introduciendo elementos que tengan en cuenta el caso particular de una *start-up*.

> ***No "sacrifiques" la seguridad por la falta de recursos. Sabemos que en el contexto de una start-up los tiempos y la inversión en ciberseguridad disponibles son limitados. Esta guía trata de conseguir el máximo efecto con los mínimos recursos (económicos y de tiempo) posibles.***

En esta guía no se van a tratar temas de gestión de la privacidad, identidad digital y reputación online de los miembros de la *start-up* por limitar el alcance de esta. No obstante, es un tema muy importante que en algún momento del proceso de creación de la *start-up* tendremos que abordar. Por suerte, para facilitar la tarea el INCIBE tiene una formación llamada ***Privacidad, identidad digital y reputación online*** [1] que te puede ayudar con este proceso.

Tampoco hemos contemplado la respuesta a un incidente que se haya producido, por ser algo posterior a la puesta en marcha del *software* creado y también por limitar el alcance de este documento. No obstante, el INCIBE tiene una guía llamada ***Guía nacional de notificación y gestión de ciber incidentes*** [2], que te puede orientar en cómo hacer este proceso en el territorio español.

2. La filosofía "*pushing left*" en el mundo de las *start-up*

Independientemente de la metodología que uses para el desarrollo de tu aplicación, la **filosofía *pushing left*** es algo que encaja muy bien con el contexto de una *start-up*. Esta filosofía se basa en **detectar cuanto antes errores de seguridad** en la típica cadena de pasos que implica el desarrollo de un *software*, representada en la Figura 1:

Figura 1. Cadena típica de pasos de construcción de un software

Una **detección temprana** de un posible error o problema de seguridad es lo más adecuado, porque se ha determinado que cuanto más "a la izquierda" del diagrama anterior se haga (de ahí el nombre), **más barato será arreglar dicho problema**.

> ***Arreglar un potencial problema de seguridad descubierto mientras elaboramos los requisitos cuesta mucho menos que si lo detectamos en la fase de testing (donde probablemente tengamos que revisar el diseño o los propios requisitos), o incluso con la aplicación en funcionamiento ya desplegada (el peor caso posible).***

Y por "barato" no sólo entendemos el aspecto **económico**, sino que también implica el **tiempo** que se tarda en solucionar el problema. Como puedes comprobar, ambas cosas están muy en sintonía con el contexto de la limitación de recursos que típicamente tiene una *start-up*.

Debido a esta filosofía, en esta guía vamos a **priorizar las medidas que se puedan aplicar lo más a la izquierda posible del proceso contemplado en la *Figura 1***. Por eso, hay más contenido a nivel de requisitos que en las siguientes fases, como diseño y construcción de código. Estas fases son ya más dependientes del *software* a desarrollar y del modelo de negocio que se quiera explotar.

3. Requisitos de seguridad desde la perspectiva de la agilidad y la economía de recursos

Para facilitar el **ahorro de recursos** a lo largo del ciclo de desarrollo del *software* de una *start-up*, no solo es importante incorporar la seguridad desde el principio (*pushing left*), también lo es **elegir la mejor forma de implementarlos** de cara a conseguir el mejor balance posible entre seguridad y el coste de implementarla.

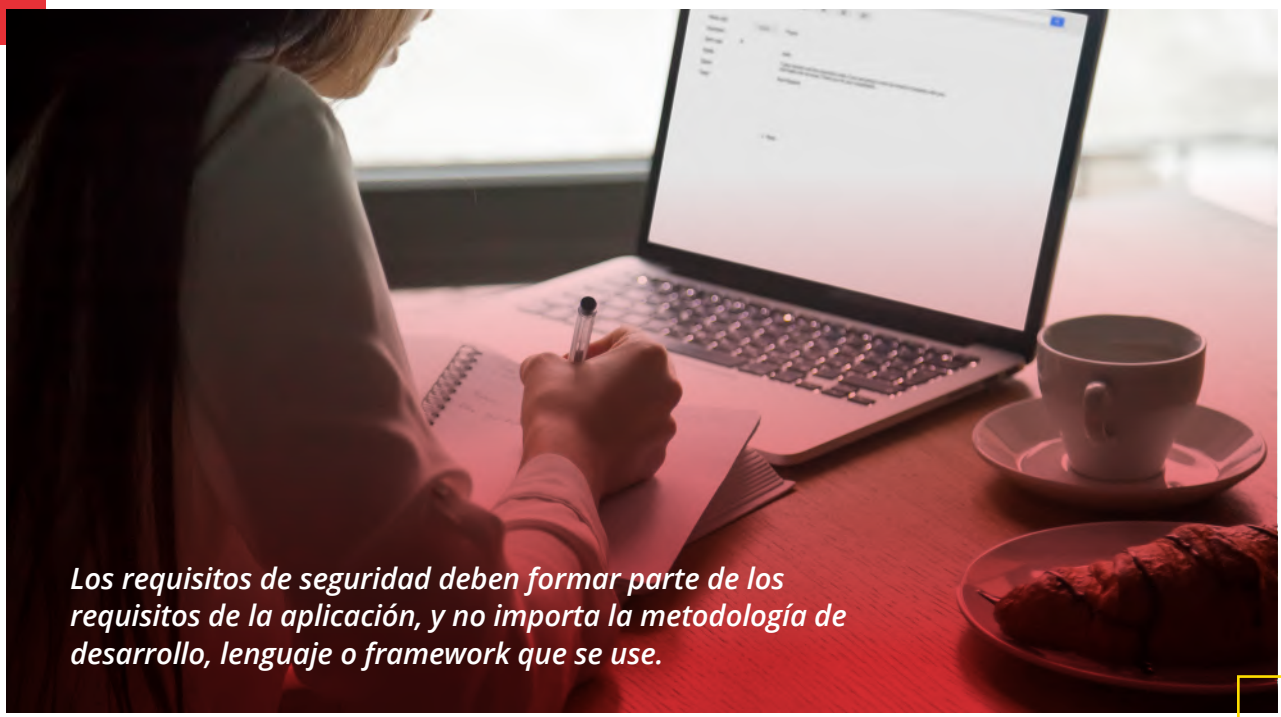
Por ese motivo, en esta guía se va a hacer especial énfasis en cómo modelar determinados requisitos **para que suponga el mayor ahorro posible**, sin que por ello se tenga que hacer un sacrificio significativo en la seguridad del *software*, al menos en las fases iniciales de su ciclo de vida. También se harán recomendaciones en la línea de encontrar la manera **más ágil** de poder llevarlos a cabo, para minimizar el tiempo que la aplicación necesita para ponerlos en marcha.

Piensa en frío y con calma tus decisiones en esta fase inicial. Con más recursos uno puede implementar un nivel mayor de seguridad de lo que se muestra en varias partes de esta guía y, de hecho, en muchos puntos se indica cómo hacerlo. Pero ¿realmente los tienes? Si inviertes mucho en eso, ¿llegarás a tiempo para sacar tu producto? ¿te quedarás sin recursos para terminar tu Minimal Viable Product (MVP)? Es un balance delicado que es complejo de calcular, pero esta guía puede ayudarte a tomar esas decisiones para el contexto de tu negocio.

La mejor forma de abordar el tema de requisitos en **separarlos por la temática** que tratarían dentro del desarrollo de tu aplicación, explicar sus objetivos generales y que, con esos recursos, seas capaz de adaptarlos a tu desarrollo de la mejor forma posible, usando además las guías y consejos que te damos. Por ello, vamos a hacer la siguiente división de requisitos de seguridad (RS) y explicarlos uno a uno.

- **RS1** Requisitos de validación de entradas.
- **RS2** Requisitos de cifrado de la información manejada.
- **RS3** Requisitos de gestión de dependencias de *software* de terceros que necesitemos para hacer una operación.
- **RS4** Requisitos de gestión de datos.
- **RS5** Requisitos tecnológicos de seguridad y de revisión del *software*.
- **RS6** Requisitos de cuentas de usuario, autenticación y autorización.
- **RS7** Requisitos de funcionalidades particulares.
- **RS8** Requisitos de log o de registro de acciones que han ocurrido en la aplicación.

En cada uno de estos apartados vamos a explicar **en qué consisten, la importancia que tienen, qué tipo de problemas evitarían y dar indicaciones para llevarlos a cabo** en las fases de diseño e implementación siguientes. Algunos de ellos también los trataremos expresamente en alguna de dichas fases posteriores dada su especial importancia.



Los requisitos de seguridad deben formar parte de los requisitos de la aplicación, y no importa la metodología de desarrollo, lenguaje o framework que se use.

Ahora es el momento de que una persona (o idealmente un equipo, aunque sabemos que en el contexto de una *start-up* esto es más difícil) se centre en desarrollar los requisitos de seguridad de la aplicación que tu *start-up* va a construir y **se haga todas las preguntas que hagan falta para elaborarlos**, como pueden ser las siguientes:

- *¿Contiene o maneja mi aplicación información confidencial, sensible o que identifica a las personas (Personal Identifiable Information, PII)?*
- *¿Puedo evitar que lo haga dejando que otro la gestione y librándome a mí de la responsabilidad y el trabajo que conlleva sin sacrificar su seguridad?*
- *¿Dónde y cómo se guardan los datos (formato, cifrado...)?*
- *¿Estará disponible mi aplicación públicamente (Internet)? ¿O no? (solo para la Intranet de mis clientes)*
- *¿Va a realizar la aplicación tareas sensibles o esenciales? (transferencias de dinero, manejo de datos médicos...)*
- *¿Hace algo considerado potencialmente peligroso o problemático? Ej.: permitir a los usuarios subir ficheros, manejar datos de para hacer pagos, tener varios roles de usuarios con distintos privilegios...*
- *¿A qué normativas legales de seguridad debo atenerme para poder desarrollar mi negocio con este software?*

El problema principal es hacerse las preguntas correctas y que no falte ninguna. Vamos a hacer un recorrido por los requisitos principales para intentar asegurarnos de ello.

3.1. RS1 Requisitos de validación de entradas

3.1.1. RS1.1 No confiar en ningún dato que nos llegue

Validar y sanear (en casos especiales) todos los datos, incluidos los de la propia base de datos de la aplicación, es un **requisito de seguridad imprescindible**, a implantar siempre. Por eso va el primero de todos. Además, debes hacerlo de forma extremadamente estricta: **Nunca confíes en la entrada del usuario.**

Asume desde ya que **toda entrada** que llegue al sistema **puede ser falsificada o manipulada maliciosamente**, y esto puede causar que una aplicación falle de formas no previstas y se vuelva inestable. El problema es que, en esta situación, un atacante puede usar nuestra aplicación para desencadenar un ataque muy peligroso, precisamente porque esta inestabilidad puede tener consecuencias impredecibles. Si una aplicación está en un estado no previsto, lo que ocurra a continuación tampoco puede serlo (y menos cuanto más complejo sea el *software*). **Esta incertidumbre se debe evitar a toda cosa**, puesto que no queremos aparecer en un aviso de este tipo como los que el INCIBE publica, por desgracia, frecuentemente (Ej. **[3]**).

> ***Si dudas de si algo tiene o no que validarse debido a que no tienes claro su origen exacto, valídalo. En esto es mejor pecar de exceso...***

3.1.1.1. Cómo validar

Por tanto, **TODA ENTRADA debe ser validada y manejada adecuadamente**, y siempre que puedas debes basarte para ello en **listas de entradas admitidas**. En otras palabras: **no intentes bloquear todas las posibles entradas inválidas como primera forma de solucionar este problema**; la mayoría de las veces no es posible.

> ***Las listas de entradas inválidas tienen su uso, pero salvo que el rango de valores de la información que admitas sea muy pequeño y conocido, normalmente no podrás usarlas para parar problemas de validación. Hablaremos más de ello aquí***

Debes especificar como requisito qué entiendes por un valor válido para cada dato que estás pidiendo o recibiendo de un usuario o sistema externo y rechazar todo lo que no encaje con esta especificación. Por tanto, a la hora de elaborar los requisitos de tu aplicación debes también especificar de la forma más detallada que te sea posible en esta fase:

- **Cuántos** datos vas a recibir para poder cumplir con las funcionalidades previstas por tu aplicación.
- **De quién** los recibes (sistemas o actores externos o internos).
- **Qué** son (su tipo).
- **Cómo van a validarse** (límites de rangos de valores, semántica de esos datos para entender qué se entiende por un dato válido y que no.).

> ***Por ejemplo: Vas a consultar con una API de mapas (de quién) la geolocalización de un sitio (cuántos). Recibirás un par de coordenadas de latitud y longitud (qué) que tendrás que validar para comprobar que son nºs reales comprendidos entre -180º y 180º (longitud) y -90º y 90º (latitud) para que sean correctos (cómo).***

No tienes que invertir tiempo en elaborar las normas para validar la sintaxis de cada posible tipo de dato de entrada. Estas normas de sintaxis, independientemente de la clase de software que construyas **son conocidas y están ya clasificadas y documentadas [4]**.

Buena parte de los procesos de validación sintáctica (es decir, la estructura de lo que recibes) pueden implementarse con ***expresiones regulares*** *(RegEx)*. Los lenguajes de desarrollo actuales tienen funcionalidades para procesarlas y validar datos con ellas. Cada lenguaje tendrá su forma de hacerlo, pero una vez entendida, siempre se aplicará igual. Aquí vemos un ejemplo con ***Java***, sacado de la referencia anterior:

```
private static final Pattern zipPattern = Pattern.compile("^\d{5}(-\d{4})?$");
public void doPost( HttpServletRequest request, HttpServletResponse response)
{
  try {
      String zipCode = request.getParameter( "zip" );
      if ( !zipPattern.matcher( zipCode ).matches()  {
          throw new YourValidationException( "Improper zipcode format." );
      }
      // do what you want here, after it has been validated
  } catch(YourValidationException e ) {
      response.sendError( response.SC_BAD_REQUEST, e.getMessage() );
  }
}
```

Tampoco tienes que buscar una expresión regular concreta que valide cada tipo de dato posible, puesto que hay muchas de ellas disponibles, revisadas y validadas **[5]** *(Figura 2). Estas expresiones usan la sintaxis PCRE (Perl-Compatible Regular Expressions), que es compatible con la mayoría de los lenguajes actuales.*

Aunque las expresiones regulares puedas encontrarlas preconstruidas, no podemos descartar un error al copiarlas o aplicarlas. Por ello, es recomendable incluir como requisito que se hagan **pruebas unitarias** para comprobar que cada entrada se está validando correctamente, incluyendo casos de entradas válidas e inválidas conocidas o típicas.

```
<regex>
    <name>url</name>
    <pattern><![CDATA[^((((https?|ftps? | gopher|telnet|nntp)://)|(mailto:news:))
(%[0-9A-Fa-f]{2}|[-    ()_.1~*';/?:@&    +$,A-Za-z0-9])+)    ([).1';/7:,][[:blank:
|:blank: ]])?$]]></pattern>
    <description>A valid URL per the URL spec.</description>
</regex>

<regex>
    <name>IP</name>
    <pattern><![CDATA[^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4]
[0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-  4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-
4][0-9]|[01]?[0-9][0-9]?)$]]></pattern>
    <description>A valid IP Address</description>
</regex>

<regex>
    <name>e-mail</name>
    <pattern><![CDATA[^[a-zA-Z0-9_+8*-]+(?:\.[a-zA-Z0-9_+&*-]+)*@(?:[a-zA-Z0-9-
]+\.)+[a-zA-Z]{2,}$]]></pattern>
    <description>A valid e-mail address</description>
</regex>

<regex>
    <name>safetext</name>
    <pattern><![CDATA[^[a-zA-Z0-9]+$]]></pattern>
    <description>Lower and upper case letters and all digits</description>
</regex>

<regex>
    <name>date</name>
    <pattern><![CDATA[^(?:(?:(?:0?[13578]|1[02])(\/|-|\.)31)\1|(?:(?:07[1,3-
9]|1[0-2])(\/|-|\-)    (?:29|30)\2))(?:    (?:1[6-9]|[2-9]\d)?\d{2})$|^(7:0?2(\/|-
|\.)2913(?:(?:(?:1[6-9]|[2-9]\d)?(?:0[48]|[2468][048]|[13579][26])|(?:(?:16|
[2468][048]|[3579][26])00))))$|^(?:(?:0?[1-9])|(?:1[0-2]))(\/|-|\.)
(?:0?[1-9]|1d|2[0-8])\4(?:(?:1[6-9]|[2-9]\d)?\d{2})$]]></pattern>
</regex>
```

Figura 2. Ejemplo de expresiones regulares validades por el OWASP

Ten en cuenta que una entrada de tu aplicación **NO debe marcarse como "confiable"** para el resto de ella hasta pasar los procedimientos de validación cuando se recibe. **NUNCA debe usarse un valor de entrada antes de pasarla a otros módulos de tu *software*.** Por procedimientos de validación se entienden todas las pruebas que aseguren que **la entrada es apropiada y exactamente del tipo que se espera recibir**, usando las herramientas que acabamos de describir.

> ***Una metáfora de este proceso la podemos ver en los hospitales: Antes de pasar a un entorno que debe ser estéril (la funcionalidad de la aplicación) debes pasar un proceso de descontaminación (la validación) para evitar dañarlo. La aproximación conceptual es la misma: trata a la funcionalidad de tu aplicación como un elemento muy sensible que recibe datos que deben revisarse a conciencia antes de que los use, y da igual de dónde se reciba: es mejor asumir que no hay origen de datos fiable nunca, aunque procedan de una API de una Big Tech.***

¿Qué ocurre si una validación no tiene éxito? Lo detallaremos más adelante, pero en general cualquier discrepancia entre lo que recibimos y lo que esperamos debe **rechazarse** como requisito. Esta aproximación no solo es mucho más segura, sino que ahorra tiempo, lo cual es muy importante en nuestro contexto. En general, también **es aconsejable intentar fijar un tamaño máximo esperable para cualquier dato manejado**, sea del tipo que sea.

¿Necesitas ejemplos concretos? Aquí tienes unos cuantos:

- Si esperas una **fecha de nacimiento**: Comprueba que es una fecha coherente (¿y se usa para calcular la edad de una persona viva?).
- Si esperas un **nombre**: Comprueba que tiene las características esperables en tu caso (nacionalidad de los usuarios (Ej.: "O'Reilly"), caracteres japoneses/chinos, etc.).
- Si esperas un **email**: Hay expresiones regulares que pueden validarlos sin error (***Figura 2***) y no perder tiempo en analizarlos (es bastante complejo). Según el contexto, puedes incluso estudiar enviar un correo para comprobar que es una cuenta activa.
- Si recibes el **resultado de una búsqueda**: Valida que tiene el tipo y formato esperado (XML, JSON, valores de un tipo dado...). Esto es algo que el propio servicio que uses debería tener documentado.
- Si llamas a una **API** y te devuelve un tipo conocido: **DEBES** comprobar que lo que recibes tiene un valor coherente, aunque sea del tipo correcto (como el ejemplo que pusimos de latitud y longitud). Esto incluye que no sea excesivamente largo.

Es un error común pensar que cualquier servicio al que llames va a devolverte siempre valores correctos y con sentido. Si tu código puede fallar y tener vulnerabilidades, el código de otros también (sí, insisto, aunque sean "otros" muy grandes).

- Si tu aplicación debe **aceptar URLs** (o fragmentos de ellas) de los usuarios y manejarlas, debes entender que estás en una **situación potencialmente peligrosa** (llamada ***open redirect***), por lo que intenta evitar ese escenario en la medida de lo posible. Modela tus requisitos para no darles a tus usuarios la capacidad de controlar a dónde redirige tu aplicación con algo que ellos mismos introduzcan en la misma.

  No siempre vas a poder evitarlo, por lo que, si tienes que hacerlo, **valida que realmente es una URL** (expresiones regulares de nuevo, ***Figura 2***), que es válida para el contexto de la aplicación (Ej.: dirige a un recurso del tipo esperado) y asegúrate de que te llegan por un canal cifrado.

  Este puede ser un ataque complejo que debes prevenir desde el principio, y para ello puedes consultar información al respecto sobre cómo hacerlo una vez elijas en qué vas a desarrollar tu aplicación (Ej.: **[6]**, ***Figura 3***):

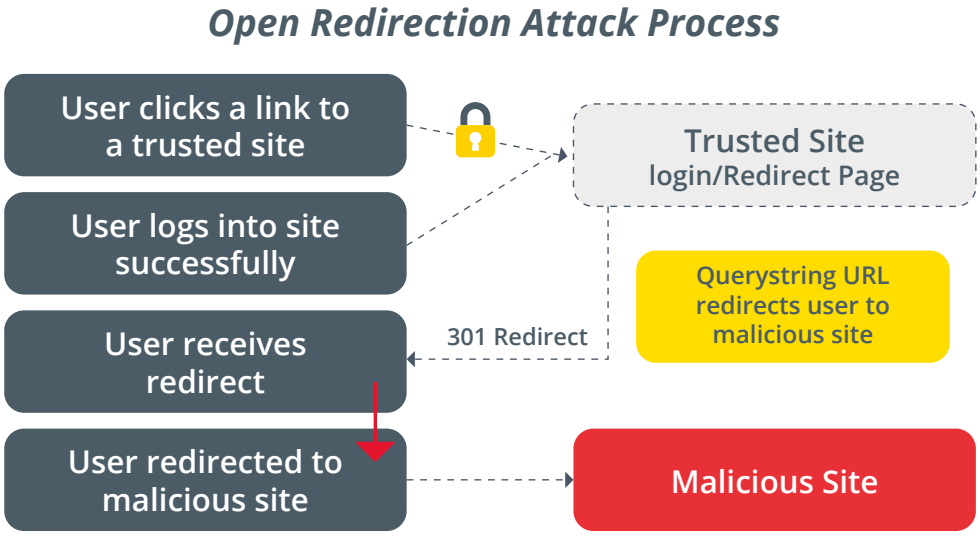


Figura 3. Esquema de una vulnerabilidad tipo Open Redirect

- Finalmente, si es posible, evita que tu aplicación llame o use código hecho con **lenguajes no *memory-safe***, como **C/C++**. Estos lenguajes priorizan el rendimiento, pero usarlos **introduce nuevas amenazas potenciales** que necesita recursos tratar de mitigar. Si necesitas un código que rinda lo mejor posible pero que evite estos problemas, pon como requisito usar ***Rust*** **[7]**.

Esto no es algo trivial, es incluso una recomendación de la NSA **[8]**. No obstante, eso evita un nº de vulnerabilidades, no todas. Los lenguajes como ***Rust*** también tienen vulnerabilidades (Ej. **[9]**), como todo *software*.

3.1.1.2. Dónde validar

Si tu producto es web, has de tener en cuenta que **validar la entrada en *JavaScript*** es bueno para tus usuarios. Te permitirá darles una respuesta más rápida si cometen un error al introducir algún dato (***Figura 4***), pero esta validación es inútil de cara a la seguridad.

Figura 4. Típico aspecto de una validación en el cliente con JavaScript

> *Lo que es muy bueno para la usabilidad, no sirve para la seguridad: **La validación SÓLO en JavaScript no puede asegurar nada**, ya que se puede desactivar, bien por acción del navegador (desactivar JavaScript, un plugin tipo NoScript...) o con programas especializados como un **proxy web** (Ej.: **[10]**). Estos proxys pueden cambiar cualquier contenido justo antes de enviarlo al servidor, lo que hace validar en JavaScript inútil.*

Por tanto, en un contexto web, cliente-servidor o siempre que pueda existir un elemento intermedio que pueda interceptar y cambiar los datos enviados por un cliente, **es absolutamente necesario establecer como requisito que la validación** de cada tipo de datos **se haga tanto en el cliente** que los genera (por usabilidad) **como en el servidor** que los recibe (por seguridad), de manera idéntica.

Por suerte, muchos *frameworks* ya permiten fácilmente crear una validación doble de esta clase si usamos sus características integradas para ello. Podemos establecer como requisito de nuestra aplicación **usar estas validaciones integradas** y asegurarse de que lo que se **valide sea lo mismo en todas las partes de esta para cada entrada [11]**.

3.1.1.3. Qué validar

Quizá llegados a este punto te preguntes: Si se sabe cómo y dónde hacer la validación, ¿cuál es el problema exactamente? ¿Por qué hay tantos casos de problemas de validación que terminan en vulnerabilidades y errores de aplicaciones hoy en día? Aunque los factores varían con el tipo de aplicación, el problema es que muchas veces no se tiene clara la **ENORME CANTIDAD** de fuentes de datos que se pueden manejar al crear la aplicación.

Por ello, hay que examinar este aspecto con cuidado y **documentar correctamente cada una de ellas**. Veamos una lista no exhaustiva de entradas manipulables de una aplicación en un escenario web, pero que se puede adaptar a cualquier otro que necesitemos y/o usar como guía para elaborar uno propio:

- La obvia, cualquier entrada de usuario que nos venga por GET o POST.
- **Valores en la URL** (parámetros), en las cookies, en los ficheros de configuración... Especialmente si las cookies no tienen los *flags "secure"* y *"HTTPS only"* (hablaremos de ellos aquí), pero en general, **en cualquier circunstancia** (tengan o no activos esos *flags*). Esto también incluye datos que se lean de sistemas de almacenamiento online (S3, *Firebase*...).

> ***A alguna gente le extraña que se recomiende desconfiar así de sistemas externos, pero piensa: ¿Y si alguien los cambia sin permiso? ¿Y si es por la acción de otra aplicación "envenenada"? ¿Tenemos realmente algún control de lo que hacen? Mejor valídalo todo por defecto y evita problemas inesperados. Un atacante va a ir a por lo que no te esperas, tiene experiencia en hacerlo...***

- **Datos o comandos** que vengan de aplicaciones en nube, sean de quienes sean (nunca sabes si pueden ser atacadas en el futuro y servirte datos corruptos o maliciosos).
- **Imágenes** que recibas de otros sitios, para comprobar que realmente son imágenes y no otro tipo de ficheros encubiertos. En general **debemos ser muy estrictos con el tipo de ficheros que admitamos** en determinadas partes de la aplicación.
- Datos que vengan de, o se calculen, con **código de terceros que hayas incluido**: Librerías, *frameworks*, código de *GitHub*, *StackOverflow*, cosas que le hayas pedido a *ChatGPT* (o similares), etc. Incluyendo cosas que copies y pegues.

> ***Siempre asegúrate de entender lo que estás copiando y no copiar cosas muy ofuscadas: No es la primera vez que se sirve malware de esta forma tan "inocente". No puedes poner tu fe en cosas que alguien desconocido (o no) ha puesto en Internet, porque no sabes sus intenciones reales (o quizá den problemas por accidente).***

3.1.2. RS1.2 Hacer toda la validación en el servidor

Como hemos dicho antes, se debe hacer obligatoriamente toda la validación de seguridad en el lado del servidor usando, en la medida de lo posible, una estrategia de listas de elementos admitidos. Yendo un poco más allá, es importante también **actuar correctamente** si detectas un error en la entrada: **siempre debes rechazar los datos inválidos e informar del error adecuadamente**. Esto conlleva varias cosas:

- **NO intentes corregir los datos: "Sanear" la entrada** intentando interpretar lo que el usuario ha querido introducir es mala idea. **Recházala y vuélvesela a pedir al usuario**. "Sanear" abre la puerta a otros errores de seguridad que no te puedes permitir controlar, porque necesitas agilizar tu desarrollo y es propenso a errores.

Si, como hemos dicho, la entrada que da un error se valida además en el cliente usando la misma validación, un usuario "normal" no debería provocar un fallo de validación en el servidor: el valor erróneo no hubiera llegado. ¿Está el usuario intentando explorar la aplicación para encontrarle algún fallo entonces? ¿Podría modelarse como requisito tomar alguna acción contra estos casos (bloqueo de cuentas, IPs, etc.)? Esto puede llevar un tiempo, que quizá no tenemos en el contexto de una start-up, pero merece la pena al menos pensarlo para el futuro.

- **Nunca des demasiada información al usuario cuando muestres el error**, ya que podría usarla en tu contra. Los errores **no deben revelar el *software* usado** para crear partes de una aplicación ni sus versiones, a partir de los cuales es muy fácil sacar los **CVEs (*Common Vulnerabilities and Exposures*) de las vulnerabilidades conocidas para un determinado producto y versión**. Se puede establecer como requisito determinar la información exacta que cualquier error producido y notificado debe mostrar al usuario. El siguiente punto explica más este problema.

Un error como el de la Figura 5 nunca debe aparecer en la aplicación.

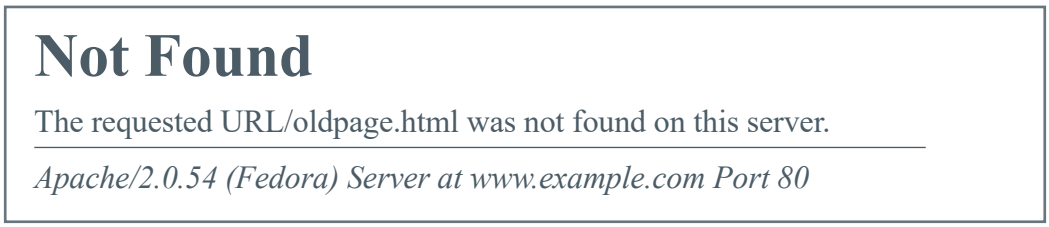


Figura 5. Error de una aplicación web que revela un producto usado en su desarrollo y su versión

- **Jamás reveles productos y versiones usados**: No solo en errores inadecuados (***Figura 5***), sino en cualquier otra parte de la aplicación (comentarios, ficheros de los *frameworks*, etc. que suban a producción sin ser estrictamente necesarios...).

  Si lo hacemos, facilitamos el convertirlo en la información de la ***Figura 6***, donde no solo revela que la web tiene una evidente **deuda técnica** (la versión de *Apache 2* es muy antigua), sino que tiene vulnerabilidades conocidas, algunas de ellas **muy graves** (incluso con la máxima puntuación) y, para colmo, una de las más graves tiene un ***exploit*** (una forma de sacarle partido) disponible públicamente (concretamente *ExploitDB*).

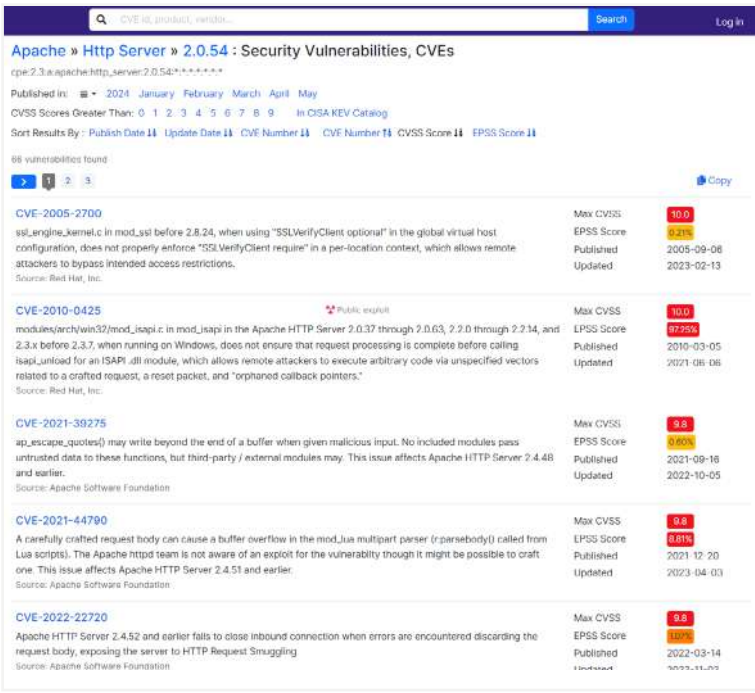


Figura 6. CVEs de una versión antigua del servidor web Apache 2

Como puede verse, de una simple información hemos derivado compromiso del servidor, lo cual es completamente inaceptable.

- **Muestra solo información general del fallo de validación**: Hacerlo no elimina los problemas subyacentes que causan el error, pero al menos no facilita encontrarlos. La ***Figura 7*** es un ejemplo de página de error adecuada, al ser informativa para el usuario y no dar datos técnicos.

- **Quédate para ti con la información técnica del error**: Una cosa es no dar información técnica al usuario y otra muy distinta que no la tengas tú para hacer depuración. Uno de los requisitos es **especificar qué clase de registro de errores de validación** llevarás internamente, de manera que solo tú y la parte del equipo encargada del desarrollo pueda usarla para depuración.

Figura 7. Página de error informativa, pero sin detalles técnicos

Esto normalmente se hace escribiendo los errores en logs internos de la aplicación, que luego se procesarán de alguna forma dependiendo del entorno de desarrollo que hayas elegido. Lo importante en esta fase son dos cosas: que existan y que no sean legibles por el público general.

- Trata de **no reproducir el dato** que produjo el error en el mensaje de error de respuesta. Es muy aconsejable que ningún mensaje de error **reproduzca datos introducidos por el usuario**, ya que abre la puerta a más posibles ataques.

En general, debería bastar con un mensaje informativo genérico Ej.: "Edad no válida, por favor introduzca un número entre 18 y 65 años" en lugar de "El valor de edad introducido: [...] no es válido". Si no te queda más remedio que hacerlo, entonces recurre a su codificación y "escapado" antes de escribirlo como parte del error (siguiente requisito).

La ***Figura 8*** es un ejemplo de una pantalla de error adecuada por no dar datos técnicos como la anterior, pero inadecuada por reproducir el dato introducido por el usuario (la URL a la que ha intentado navegar). En este caso ese dato está correctamente saneado, pero este proceso podría tener algún fallo y aprovecharse para saltárselo y propiciar algún ataque, como **XSS**. Reproducir la URL en el error es innecesario, puesto que el usuario la tiene en la barra de navegación.

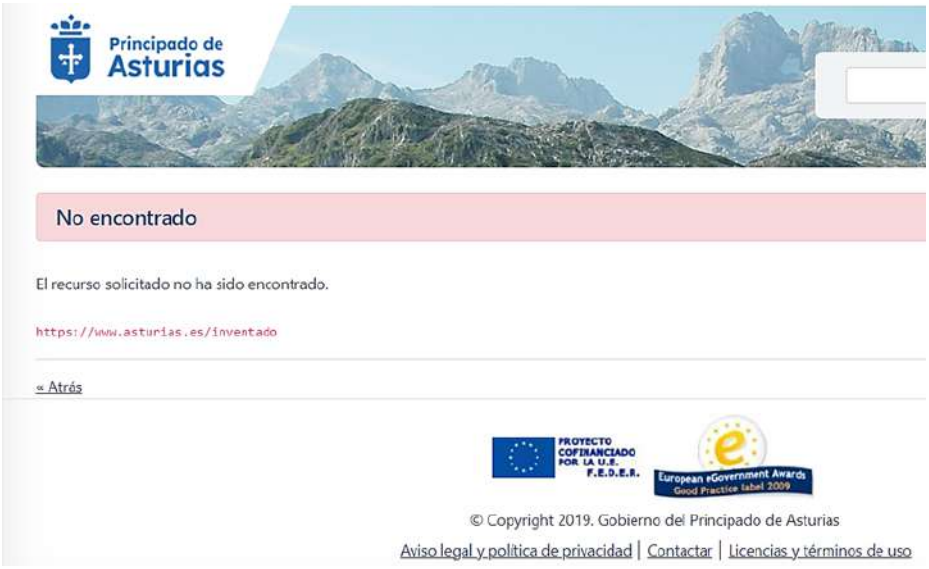


Figura 8. Página de error que reproduce un dato introducido por el usuario, lo cual no es adecuado

3.1.3. RS1.3 Codificar todas las salidas de la aplicación

"Escapar" un carácter o un valor implica **hacerle perder "el poder" que tiene en un determinado contexto**, lo que normalmente significa eliminarle la capacidad de que sea **interpretado como código**. En entornos web, habitualmente se hace añadiendo un **** al carácter, pero deben usarse las rutinas que el *framework* de desarrollo tengan para esos fines, puesto que es un tema más complejo de lo que parece.

Codificar un valor es cambiarlo por otro en un formato diferente que, normalmente, es inocuo en el contexto en el que se va a usar.

> ***Un valor codificado nunca se interpretaría como código, como cuando en HTML ">" se convierte en ">". El tipo de codificación depende del contexto en el que se vaya a usar (URL, Base64, codificación HTML…) y, para evitar problemas, los requisitos deben determinar las situaciones en las que aplicar estas técnicas.***

Todo esto se hace para **evitar ataques de inyección de código**, como el muy conocido (y peligroso) XSS. Toda esta inversión inicial en detallar estos requisitos de manipulación de datos se debe a que un ataque de esta clase puede ser **devastador** para nuestro *software*, y con ello el negocio nuestra *start-up*.

> ***Por ejemplo, si alguien consigue colar un script que robe los datos de pagos de las tarjetas de crédito cuando el cliente las introduce en la aplicación (un script tipo skimmer), ya da igual que cifremos la información privada, o que incluso invirtamos tiempo y recursos en certificarnos PCI-DSS (marco de referencia para páginas que manejan información de pago): Los atacantes tendrán los datos de las tarjetas de nuestros clientes y, en el momento que esto sea público, podemos dar la start-up por arruinada…***

Por todo ello, la inversión en validación y codificación de entradas y en reducir los puntos en los que la aplicación muestra los datos que el usuario ha introducido van en esta vía: **Evitar que el usuario introduzca información maliciosa** que, una vez reproducida por la aplicación, adquiera "poder" y pueda ser ejecutada como código. Le daríamos al atacante el control de nuestra aplicación, y con ello, la posibilidad de engañar o robar a nuestros clientes… Hablaremos más de la implementación de este proceso aquí.

3.2. RS2 Requisitos de cifrado de la información manejada

3.2.1. RS2.1 Cifrar todos los datos sensibles guardados

Esto es una máxima, tanto si están en una BBDD como en otro medio de almacenamiento. **No debemos consentir que un usuario externo pueda manipular la información que maneja nuestra aplicación** simplemente abriendo los almacenes de datos que use y modificándolos. Habitualmente esto se consigue protegiendo adecuadamente los almacenes de datos (Ej.: contraseñas de la BBDD, permisos de ficheros...), además de cifrando los datos sensibles.

La información sensible, como datos personales y las contraseñas **NUNCA se deben guardar en texto plano**. Si hay que guardar contraseñas, se usan técnicas de hashing concretas para claves (*Key Derivation Functions*), pero hablaremos más de ello en una sección específica.

También se usan técnicas de *hashing* para cualquier información cuyo valor original no necesitemos. ***¿Qué clase de información es esa?*** Aquella sobre la que solo tengamos que comprobar si el usuario ha introducido su valor correctamente o no, o tengamos que determinar si ha cambiado respecto a un estado conocido. Esto engloba operaciones para comprobar identidad, autenticación, integridad de datos o ficheros...

Cuando se tengan que recuperar los valores originales de los datos privados, **usaremos algoritmos de cifrado/descifrado fuertes**, como AES, con una clave robusta y guardada en un lugar seguro. Al final, los datos personales importantes de los usuarios no deben poder leerse simplemente abriendo la BBDD, como decíamos antes (***Figura 9***).

```
SELECT *
FROM `drupal_users`
LIMIT 0 , 30
```

	uid	name	pass	mail
	0			
	1	admin	08d15a4aef553492d8971cdd5198f314	
	15	phohikebr	a611bef396371021cac48dc28cc22e83	KPy3B8Xqz9qZi5c7c4fvdFlnw5zWUwlz9Np+mXQf4Ac=
	14	thagidritr	5fd8b38c11d707df15147ae44696a8bd	eAoDNqb1/gTkkvSlKEnwMd/bXb8r30Y0otHcUj7HIGM=
	12	wecebrohasu	f916dc6b80a3619aff0988acfc1736de	XwUfhPQhle73QeH3fSwH8+G6V2X9CF0VCVLQ2Q5voN0=
	13	gewishe	d280770f14b84ba68ba7efcfe309dbf4	X+bkdCzoYjYLNdV+EP+EkwUSK/6UghqgITeCO0WG9y4=
	11	tocregac	353d19402de6a43c2da7b5a9dc526582	grxq3jmGp2MsmDqPT03j5yBtrrY3+OO6A3M2wMW0qP4=
	10	hulejam	8400bc9b8e09a8d0855951a5e49ad5ba	cPWpFn5x4yqemhBiNZQK6znzPj4VqiwI4fqAWT48xEA=
	9	uofrenacrab	5dd5847e4cc0b579c0aa550c88281417	AbgtDQftebwI3kOdWQBGehRZpm3OYRq1A2nwttJyN8E=
	16	hecocige	efe85df02e78304b6ad0f364fc17ced6	j9t6DAplZZNHdpMdx503XS0rXDK7TnAsBaA2YlMMFEAk=
	17	merewridrecr	bad8de8fdddbaf98a0099ff3f9169e0c	m26KTrJxUQTaYlClzs7XWQYxd9i088GC8AixbC2fC+I=
	18	stetruwr	e84dd1175bd4d78a63de73022eabb286	wVT5/LlyAHSQ6+tr0YaHA7OafRwweAVDdrihySMLI3k=

Figura 9. Contenido cifrado de una BBDD

Una vez esto queda claro, hay otra pregunta que debemos hacernos: ***¿qué se cifra y qué no?*** Es decir, ***¿qué es exactamente un dato sensible?*** Esto lo comentaremos en el apartado de requisitos de gestión de datos, puesto que es algo que depende del contexto de la aplicación y de la legislación vigente.

3.2.2. RS2.2 Cifrar todos los datos “en tránsito”

Por “datos en tránsito” entendemos por ejemplo los enviados / recibidos por el usuario, BBDD, APIs, etc. **Siempre se debe cifrar la información en tránsito** (usuario <-> servidor, *backend* <-> SGBD, API <-> API, etc.). Esto garantizaría la confidencialidad e integridad de la información (no la disponibilidad), es decir, que nadie pueda alterarla sin que nos demos cuenta.

> ***Y, además, es algo coherente: no tiene mucho sentido molestarse en cifrar datos privados guardados si luego se van a transmitir sin cifrar...***

Esto no se limita solo a usar HTTPs (hablaremos de él posteriormente). Como requisito tenemos que ver, una vez seleccionados los productos que van a usarse para construir las partes de una aplicación, **qué mecanismos de cifrado proporcionan y son más adecuados para nuestro caso**. Lo importante es no olvidar que toda comunicación de red entre equipos o servicios **debe ir cifrada necesariamente**.

> ***Quizá incluso disponer de estos medios de cifrado de una manera más sencilla sea un criterio para decidir qué producto usaremos para cierta funcionalidad.***

3.2.3. RS2.3 Guardar adecuadamente las contraseñas (si es necesario guardarlas)

Posteriormente, en un apartado dedicado a la autenticación, hablaremos de alternativas para hacer la autenticación de una aplicación. Pero ya podemos adelantar que un requisito que debes intentar incorporar es intentar **NO guardar *passwords* en ella**. Esto no significa no tener usuarios, sino **delegar la gestión de estos en un tercero** (ya veremos cuáles).

> ***La gestión y almacenamiento correcto de las contraseñas y el sistema de cuentas, implementar medios de recuperarlas o reiniciarlas si se han olvidado, etc. aumenta sustancialmente la complejidad de la aplicación y, con ello, el tiempo de desarrollo, lo cual lo hace inadecuado para el contexto de una start-up. Por ello, veremos que normalmente es mejor alternativa usar un servicio de autenticación de terceros si es posible (Google, Outlook...).***

No obstante, si por el carácter de la aplicación no es posible hacerlo (Ej.: no se pueden ceder los datos de nuestros usuarios a un 3º por algún motivo), entonces debemos guardar las claves correctamente siguiendo una serie de requisitos de seguridad:

- Usar un algoritmo KDF de referencia y probado.
- Forzar a que sean largas y complejas e incorporar los elementos de seguridad llamados ***salt*** (28+ caracteres) y ***pepper***.
- Usa APIs como la de *haveibeenpwned* para comprobar que una nueva clave **no está ya comprometida** (Ej.: buscar *"Pwned passwords"*).
- **Nunca informar al usuario** de si lo que ha introducido mal es el nombre de usuario o la contraseña (***Figura 10***). Cuando intente hacer *login* sin éxito, usa un mensaje genérico: *"Datos incorrectos"*. Por supuesto, tú si puedes guardar esa información en los ***logs* internos de la aplicación**.
- Nunca permitir la **reutilización de claves** ni usar claves muy parecidas o secuenciales (Ej.: *"Pass1"*, *"Pass2"*, etc.).
- **Forzar a usar un MFA**, al menos para todas las cuentas importantes de la aplicación: Hablaremos de ello posteriormente.
- **No pongas como requisito hacer un cambio de *password* periódico forzoso**, solo cuando se descubra una brecha de seguridad que te fuerce a ello.

Figura 10. Twitter/X informa si hemos introducido mal el usuario o la contraseña, dando demasiada información

Esto está en desuso porque hace que los usuarios tengan tendencia a poner claves más débiles si se les fuera a cambiarlas.

3.2.4. RS2.4 Hacer obligatorio acceder vía HTTPs a todos los sitios web

Hoy en día que una aplicación web no use HTTPS **ya no es justificable**. Ni siquiera se puede justificar porque la aplicación no maneje datos sensibles, ya que no usar HTTPS implica que, como veremos, **se pueden falsificar las peticiones** e introducir nuevos elementos (o corromperlos) en ellas por parte de un actor malicioso.

Tampoco es justificable por el tema del **coste**, puesto que sitios como ***Let's Encrypt*** **[12] permiten la creación de un certificado válido** para cualquier web que tengamos que sacar la producción, de manera gratuita y además muy sencilla de usar.

> ***Usar tráfico cifrado es un requisito obligatorio y solamente tenemos que establecer de dónde vamos a sacar el certificado y su configuración, algo que trataremos en el punto siguiente.***

Por tanto, a falta de otros medios/inversión, podemos incluso establecer como requisito el uso de *Let's Encrypt*. El proceso de instalación de *Let's Encrypt* es muy sencillo y hay una documentación muy completa para prácticamente todo sistema operativo, proveedor de nube y servidor web. La ***Figura 11*** es un ejemplo de cómo hacerlo sobre *Ubuntu 20* y *Apache 2*.

> ***Usar bien el protocolo HTTPs ya no supone ni un desafío tecnológico ni un desembolso económico necesariamente.***

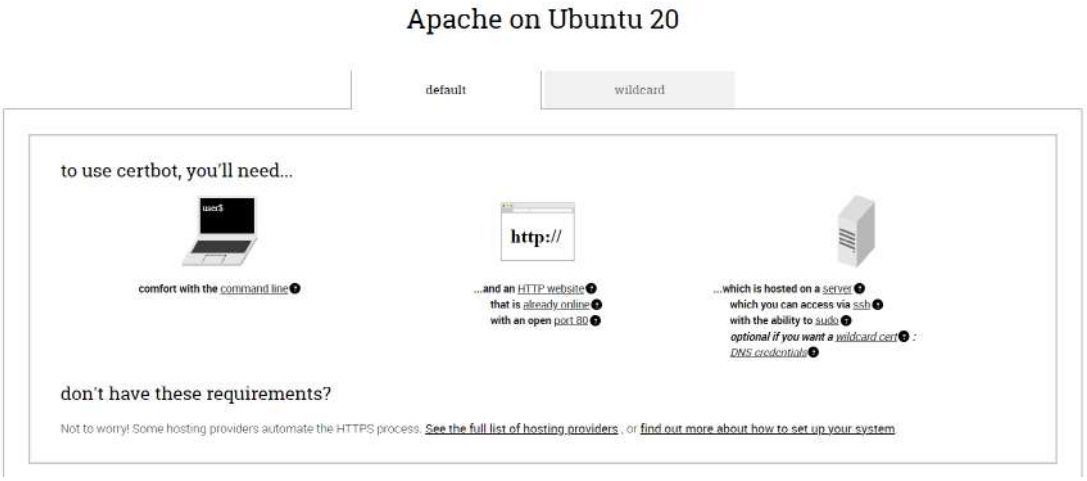


Figura 11. Instalación de un certificado Let's Encrypt sobre Ubuntu 20 y Apache 2

3.2.5. RS2.5 Asegurarse de que se usa la última versión de TLS para HTTPs

Hay que usar HTTPs **SIEMPRE**, pero con una **configuración de TLS robusta, segura y en todas las páginas** como requisito.

> ***Cuando un usuario usa nuestra aplicación, no sabemos por qué medio lo hace (cable, Wi-Fi, datos, su propio router, uno comunitario...). Por ello, el cifrado no es algo que pueda aplicarse a unas páginas sí y a otras no.***

Usar HTTPs en todas las páginas de la aplicación tiene varias ventajas:

- **Evita *network sniffing* e interceptación de datos:** Debido a que alguien usa aplicaciones especializadas en eso (Ej.: *Wireshark*).
- **Evita la modificación del tráfico:** El tráfico sin cifrar **puede modificarse "en ruta"**, inyectando *malware*, anuncios, *scripts* maliciosos o desinformación. No hay nada peor para la reputación de nuestra aplicación que un actor malicioso sea capaz de introducir anuncios u otro tipo de elementos que pueden incitar a la descarga y ejecución de *malware* dentro de nuestras páginas porque no las hemos cifrado.

> ***Podríamos llevarnos la culpa y la responsabilidad de un ataque del cual no tengamos idea, por qué la modificación en ruta de las páginas no deja registro en nuestras bases de datos o logs. Cifrar por tanto es una necesidad.***

Por tanto, la configuración del cifrado **debe ser robusta** siguiendo los últimos estándares, recomendaciones y prácticas. No obstante, no hace falta perder tiempo en ello, puesto que esta información está disponible gratuita y públicamente **[13]**. Por tanto, como requisito debemos fijar **qué modelo de configuración queremos** en función de la compatibilidad con los navegadores de los clientes previstos (*Modern*, *Intermediate* u *Old*, ver ***Figura 12***).

Figura 12. Configuración segura de TLS dada por Mozilla para cualquier combinación de servidor web y necesidades

Finalmente, dado que los algoritmos y configuraciones pueden cambiar con el tiempo, en respuesta problemas o vulnerabilidades encontradas, es aconsejable fijar como requisito también examinar el servidor en producción periódicamente para ver **si la configuración sigue siendo correcta**. Una forma gratuita de hacerlo es **[14]** (***Figura 13***).

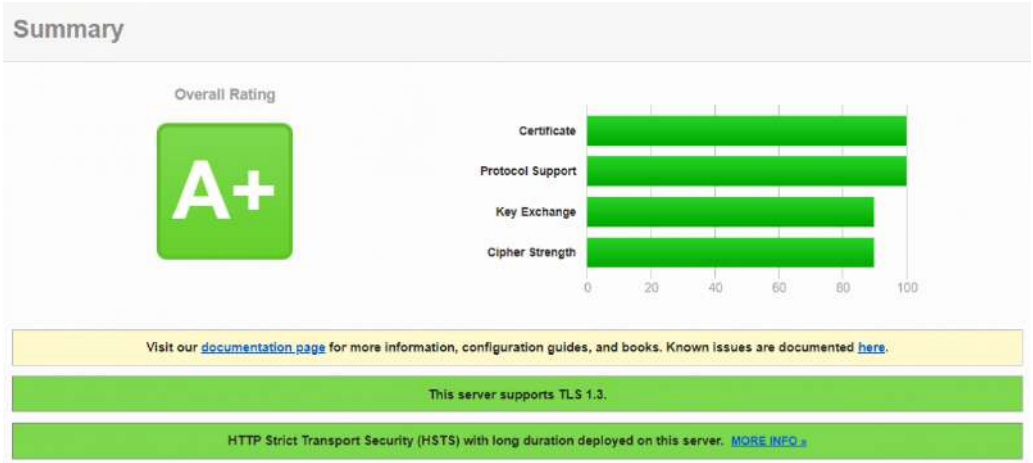


Figura 13. Análisis del uso de cifrado de una web de Qualys SSL Labs

3.3. RS3 Requisitos de gestión de dependencias de *software* de terceros que necesitemos para hacer una operación

3.3.1. RS3.1 Establecer una política de comprobación de código de terceros

En función de los objetivos del *software* a construir, tendremos que usar determinados componentes o servicios de terceros para implementarlo de forma más sencilla, rápida y barata, especialmente en el contexto de una *start-up*.

> ***Usar un componente o librería de terceros que haga una función, y que esté adecuadamente validada y probada, es casi siempre la mejor alternativa.***

La filosofía de **"no reinventar la rueda"** debe estar siempre presente en los requisitos de nuestra aplicación, y más cuando tenemos restricciones de tiempo y recursos. Deberíamos centrarnos en crear solo en aquellas partes **realmente innovadoras** del *software* que estamos planeando construir.

No obstante, usar elementos de un tercero **debe hacerse de manera supervisada, y basándose en criterios objetivos** que puedan medirse y probarse. Por ello, hay que introducir requisitos como:

- **Mantener un inventario actualizado y exacto de todos los componentes de terceros que usamos** (fabricante, versión, utilidad, licencia, web oficial...) .
- **Establecer una política para revisar periódicamente todos esos componentes de terceros a usar**. El objetivo es tener un medio para encontrar vulnerabilidades conocidas antes de usarlos, y también mientras los usamos.
- Antes de decidir usar un componente u otro, hay que examinar su **historial de vulnerabilidades** encontradas hasta la fecha (su "reputación"), su gravedad, el tiempo y la forma en que se han resuelto (lo que da idea de **cómo es de bueno y efectivo su mantenimiento**), etc. Esto será un **factor decisivo para establecer criterios objetivos para elegir la mejor alternativa** para implementar una funcionalidad.

Para comprobar esta información, si es un componente de código abierto del que no tenemos referencias podemos mirar en las ***issues*** de su repositorio, o usar **noticias o notas de prensa** que hablen de problemas que haya tenido y sus consecuencias en el pasado (esta sería la única opción disponible con componentes de código cerrado). La ***Figura 14*** muestra un ejemplo de las *issues* del conocido componente *JQuery* en *GitHub*, donde podemos consultar todos los aspectos de lo que ha pasado y cómo se ha tratado.

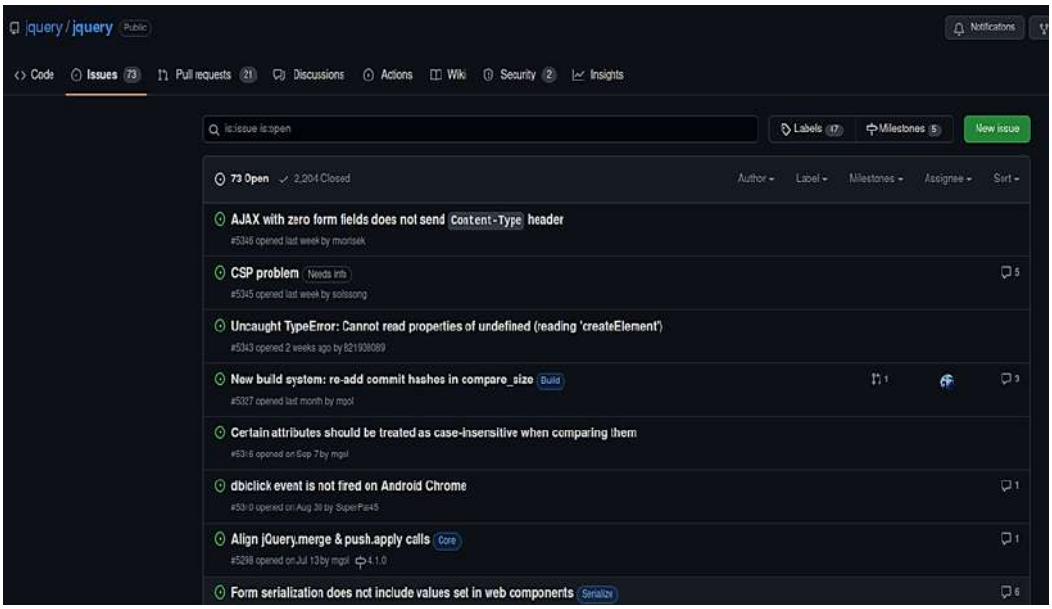


Figura 14. Issues en el repositorio de JQuery

Las **notas de prensa o noticias** de páginas especializadas (Ej.: **[15]**, ***Figura 16***) también arrojan luz sobre este aspecto. El INCIBE tiene un **servicio de Alerta Temprana** que puedes usar también para saber si algún *software* (o *hardware*) que necesites usar tiene o ha tenido vulnerabilidades graves. Puedes suscribirte vía RRSS para estar al tanto de las últimas **[16]** (***Figura 15***).

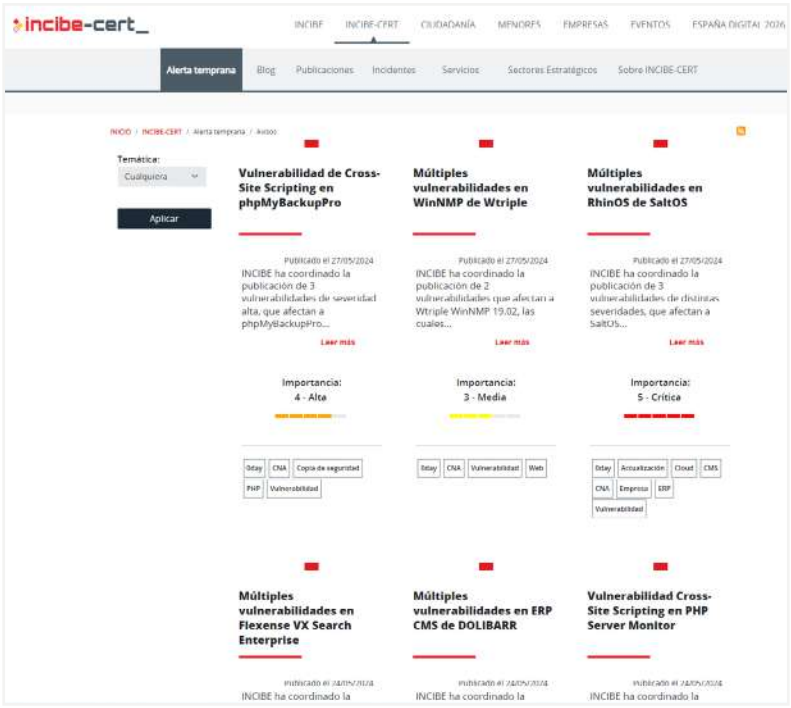


Figura 15. Servicio de alerta temprana y avisos del INCIBE, donde muestra vulnerabilidades de distinto software y su gravedad además de un botón de suscripción RSS

> ***Si te das cuenta, esto tiene mucha similitud con comprar productos en cualquier tienda: antes de decidirte por uno, hay que mirar toda la información que podamos averiguar de él.***


Página Principal > PHP > Un nuevo error de PHP Composer podría permitir ataques generalizados de la cadena de suministro

Un nuevo error de PHP Composer podría permitir ataques generalizados de la cadena de suministro

Diego Cortes Robles · jueves, abril 29, 2021 · Compartir:

Los mantenedores de Composer, un administrador de paquetes para PHP, han enviado una actualización para abordar una vulnerabilidad crítica que podría haber permitido a un atacante ejecutar comandos arbitrarios y "puerta trasera cada paquete PHP", lo que resulta en un ataque de cadena de suministro.

Figura 16. Noticia de un error grave del popular componente PHP Composer

Este requisito también encaja muy bien con lo que decíamos de **validar también los datos que recibimos de librerías externas** para comprobar que son exactamente lo que esperamos recibir. Ten en cuenta que, si alguna de esas librerías se ve comprometida y por ello nos manda información corrupta o maliciosa, una adecuada validación **puede salvarnos de determinados tipos de vulnerabilidades y sus consecuencias**, lo que hace aún más importante el requisito que hemos mencionado anteriormente.

Finalmente, debes tener muy presente que todo *software* va a tener vulnerabilidades tarde o temprano y estas se descubren a medida que pasa el tiempo. En otras palabras: **te vas a encontrar con alguna vulnerabilidad grave** en algo que estés usando mientras estás construyendo tu *software*, por muy bien que lo hayas elegido.

> ***El software tiene vulnerabilidades. No es algo que pueda o no ocurrir, va a ocurrir. La única variable es cuándo.***

Por ello, también debemos **planificar un escaneo o verificación de vulnerabilidades periódicamente** una vez incorporados a la aplicación. No obstante, esta fase de definición de requisitos simplemente tenemos que especificar que este tipo de escaneos se harán, incluyendo quizá la periodicidad con la que se hacen y otros criterios. Cómo hacerlos de **manera rápida y poco costosa** (es decir, de manera lo más adecuada para el contexto de una *start-up*) lo trataremos en un punto posterior, en la **fase de *testing* de seguridad**.

3.3.2. RS3.2 Minimizar la "superficie de ataque"

Este requisito se puede conseguir **intentando incluir el menor número de dependencias posible** y/o vigilando la reputación de estas, como vimos en el punto anterior. Pero también es una filosofía a la hora de desarrollar: **TODA** línea de código incluida de una librería, *framework*, plugin, componente de 3ºs, etc. **ES UN RIESGO** que incorporas a tu aplicación. Si algo falla en un componente de terceros o dependencia, tendrás un ataque a tu **"cadena de suministro" de *software*** (*supply chain attacks*). Por ejemplo, uno de los peores que se recuerda ocurrió en 2024, con el backdoor de la librería xz de *Linux* **[17]**.

> ***Quizá te preguntes ¿Esto no es demasiado extremo? NO, es una realidad que debes asumir (¡cada vez son más frecuentes!). Por tanto, debes tener bien claro que toda vulnerabilidad en cualquier código de terceros que incluyas es ahora TU vulnerabilidad.***

Minimizar el uso de componentes de terceros implica **tratar de usar solamente los estrictamente necesarios**. Pero que este requisito no te llame al engaño. La forma de minimizarlos **NO es desarrollando tú el código** que podría hacer la función de un componente de 3ºs por miedo a que sea vulnerado (va en contra del requisito anterior).

> ***Tu propio código va a ser más vulnerable que una dependencia de 3ºs con buena reputación y mantenimiento, ya que ha tenido menos tiempo para probarse y en definitiva menos "rodaje" que un componente así.***

De lo que más bien se trata de **no hacer *over-engineering*** y construir una aplicación más compleja de lo que se requiere, haciendo una predicción (que seguramente sea errónea) de requisitos futuros y **tratando de anticiparse a las necesidades** de un cliente que no podemos saber, o bien **complicando innecesariamente la arquitectura** de la aplicación por algún motivo (capacidad de ampliación teórica futura, por ejemplo).

Ten presente que la solución más simple a un problema concreto es más fácil de mantener, de depurar y de ampliar, e **introducir dependencias "por si acaso"** es el peor error que podemos cometer relacionado con este requisito. Además, todo esto también va en contra de la filosofía de ahorro que estamos tratando de transmitir como más adecuada para una *start-up*.

Por tanto, diseña un producto **compacto** que cumpla con los requisitos mínimos que crees que necesitas para impulsar tu negocio, y deja las ampliaciones **para cuando realmente sea necesarias**. Esto es el verdadero sentido de **minimizar tu superficie de ataque**, que también reduce el riesgo de ser afectado por una **vulnerabilidad desconocida (*zero-day*)** de alguna dependencia (menos componentes, menos probabilidades).

> ***No, no estamos hablando de hacer el código de cualquier forma. Las buenas prácticas y los buenos diseños son claves para una aplicación segura [18]. Solamente hablamos de no pensar en futuros hipotéticos a la hora de hacer los requisitos, manteniendo unas prácticas de diseño correctas. Ten en cuenta que, si el negocio crece y es necesario replantear la arquitectura de clases para incrementar su modularidad, el refactoring [19] puede solucionar el problema.***

3.4. RS4 Requisitos de gestión de datos

3.4.1. RS4.1 Clasificar y etiquetar todos los datos de la aplicación

Es necesario **etiquetar** tanto los datos guardados como los recopilados y creados **según su criticidad**. Esto debe hacerse principalmente por dos motivos:

- **Privacidad de datos:** Debe existir una política especificando los datos que se recogen en las *cookies*, **adaptada a la legislación actual** sobre las mismas en el país **[20]**.
- **Clasificación de datos:** Los datos de la aplicación deben ser clasificados y etiquetados de acuerdo con lo privados que son. De esta forma, cualquiera que esté desarrollando la aplicación sabrá fácilmente **con qué tipo de datos está trabajando** y, por tanto, qué cosas debe proteger especialmente a la hora de plantear su lectura o escritura para que se cumplan los requisitos de privacidad.

El tipo de clasificación de los datos depende del país/legislación aplicable (Ej.: Classified, Secret, Top Secret) y dependerá de si conocer un dato puede dañar a una persona, una empresa o incluso un país entero, en función de la finalidad de la aplicación.

Si no hay reglas aplicables, se pueden seguir las del **NIST [21]**, si bien se debe consultar la normativa española al respecto, contactando si es posible con **asesoría legal** que puedas obtener a través de una aceleradora de *start-up* o bien servicios de soporte de los que puedas disponer por pertenecer a un programa de fomento de estas.

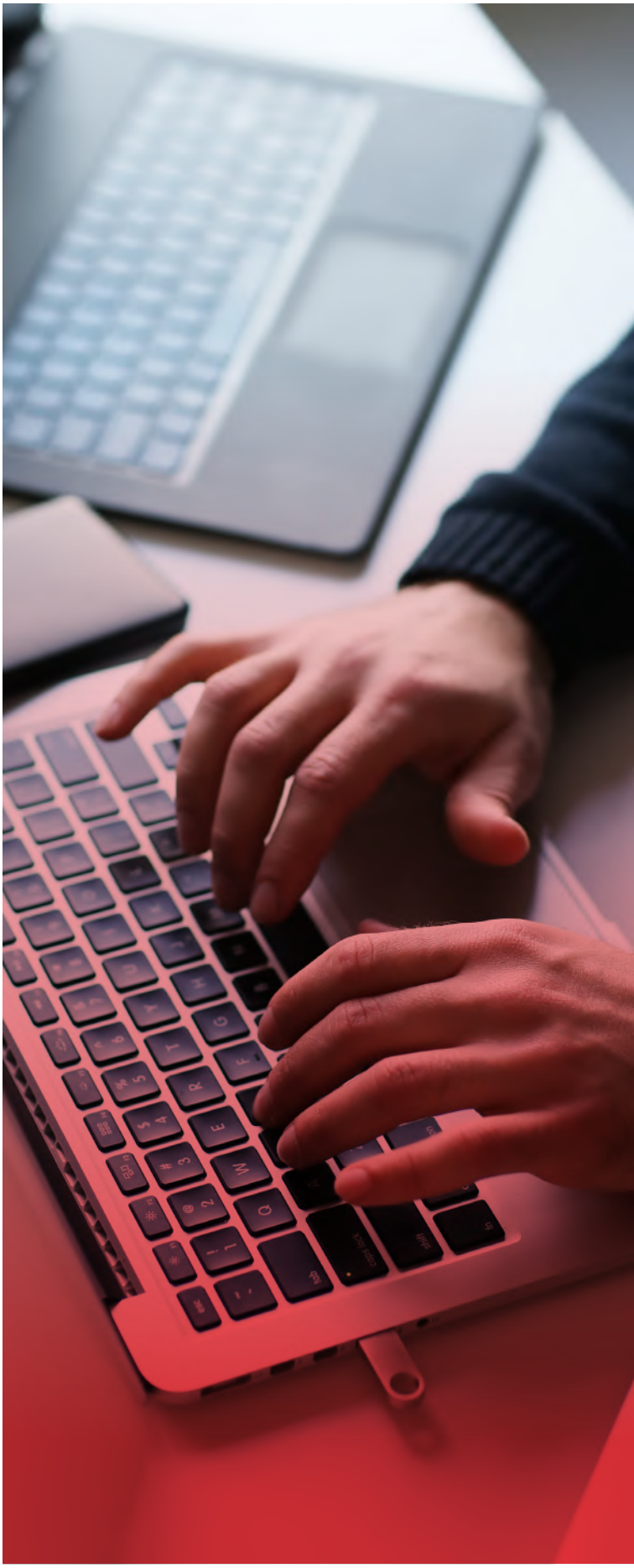
El ***Esquema Nacional de Seguridad (ENS)*** español **[22]** establece la política de seguridad para la protección adecuada de la información y los servicios digitales prestados por las *Administraciones Públicas*, pero **nada impide que lo implementen empresas privadas**, especialmente si quieren trabajar con la administración pública. Si bien en el contexto de una *start-up* plantearse cumplir con el ENS es algo que **no encaja con nuestro objetivo de ahorro de tiempos y recursos**, al menos en su fase inicial, sí que se pueden tomar ciertos elementos de él para elaborar los requisitos y facilitar su adopción futura.

Aunque no se menciona explícitamente una política de "etiquetado" de datos personales, el ENS sí establece principios básicos y requisitos mínimos que garantizan la seguridad de la información tratada. Otra de las cosas que contempla es ajustar los requisitos del ENS para garantizar su adaptación a la realidad de ciertos colectivos o tipos de sistemas, lo que podría implicar **medidas específicas para el manejo y etiquetado de datos personales**, dependiendo del contexto.

No obstante, es importante tener en cuenta que cualquier tratamiento de datos personales en España también debe cumplir con el **Reglamento General de Protección de Datos (RGPD)** **[23]**, que proporciona reglas detalladas sobre el manejo y etiquetado de datos personales. No obstante, en aras a ahorrar tiempo y recursos, como decíamos lo mejor es **consultar a un experto legal** para obtener una interpretación precisa y actualizada de estas regulaciones que podamos aplicar en nuestro *software* si prevemos tratar datos personales.

El requisito de no tener nuestro propio sistema de autenticación para minimizar los datos privados guardados de los usuarios nos facilita mucho esta labor. La ***Figura 17*** resume las precauciones con los datos personales. No obstante, el **INCIBE** tiene material de ayuda en ese aspecto, que puedes usar para perfilar tus requisitos de gestión de datos mejor **[24]**.

Ten en cuenta además que, si vas a manejar datos personales de tus clientes, tu aplicación debe cumplir con unas **obligaciones legales** para permitir que los usuarios puedan borrarlos, por ejemplo, y eso debe ser establecido también desde los requisitos si es tu caso. El **INCIBE** tiene en **[25]** su propia política de protección de datos de los usuarios, que puedes usar para modelar los requisitos de esta funcionalidad usándola como base.



1

Tu consentimiento para proporcionar tu información debe ser explícito. Ahora hay que dar un consentimineto a las empresas mediante una aprobación muy clara. ¡Se acabaron los formularios pre-rellenados!

2

Las empresas tienen derecho a recopilar lo estrictamente necesario. Por cada Newsletter o promoción por SMS, las empresas solo necesitan tu dirección de correo electrónico o tu número (que tú habrás aceptado enviarles).

3

Puedes pedir que se transfieran todos tus datos. Si cambias de operador móvil, puedes solicitar que toda tu información sea transferida de tu antiguo operador al nuevo (y que el antiguo la borre definitivamente).

4

Puedes solicitar que tus datos sean borrados en cualquier momento. Si lo deseas, toda tu información personal puede ser borrada en todo momento.

5

15 años: la edad legal para inscribirse en redes sociales (sin autorización de los padres). 15 años será la edad mínima para que los menores puedan consentir por sí solos el tratamiento de sus datos personales online.

6

Posibilidad de acción colectiva frente a la justicia. Si sientes que se ha violado alguno de los derechos relativos de la protección de tus datos personales, puedes llevar a cabo una acción judicial colectiva ("class action").

7

Sanciones a las empresas. Para las empresas que no cumplan con la normativa, el riesgo de hacer frente a una sanción es alto: hasta 20 millones de euros o un 4% de su facturación mundial anual.

Más información:

El ENS como parte del RGPD

La protección de datos en el Esquema Nacional de Seguridad

7 claves
para entender el RGPD

Figura 17. Claves para entender el RGPD

¿Qué es eso del RGPD?

Destinado a reemplazar la antigua Ley Orgánica de Protección de Datos (LOPD), el nuevo Reglamento General de Protección de Datos (RGPD) establece la manera en la que las grandes empresas del mundo digital gestionan la información que tienen sobre los internautas.

3.4.2. RS4.2 Guardar todos los secretos de la aplicación en una *secret store*

Las ***secret stores*** son unos almacenes en los que los desarrolladores guardan los secretos de la aplicación y que permiten acceder a ella de manera segura.

> ***Son algo similar a los gestores de claves, pero que las use el software, no las personas.***

Por tanto, un requisito es que exista un **archivo cifrado** (o *vault*) o bien un **servicio externo** que guarde y permita el acceso seguro a los secretos de la aplicación: Claves, *hashes*, certificados, *tokens* de servicios de nube, cualquier cosa que requiera protección... La ***Figura 18*** es un esquema que muestra el uso general de un servicio de esta clase.

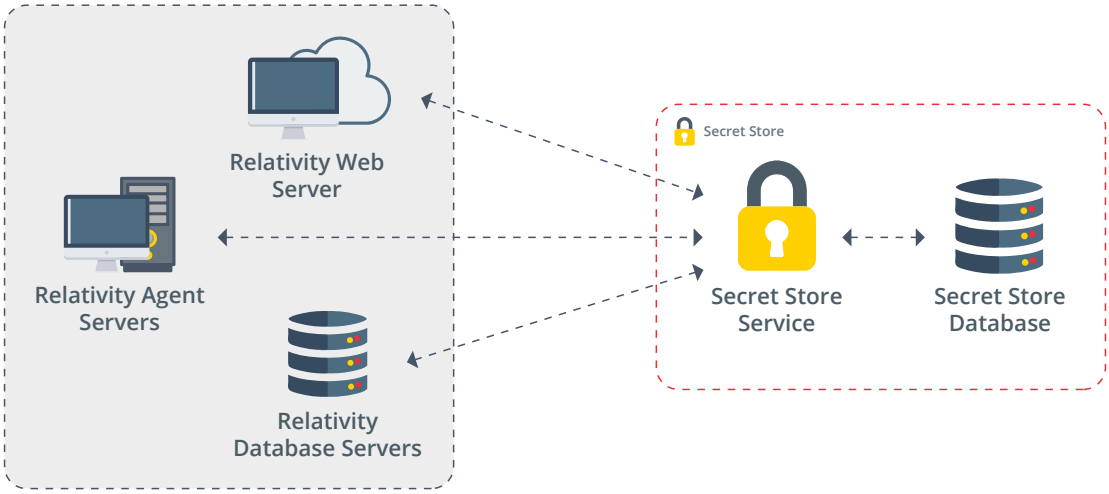


Figura 18. Esquema general de un servicio externo que hace de secret store

Quién hará la gestión de secretos dependerá de los productos usados para el desarrollo. Algunos tienen ya su ***secret store* integrada**, y, en caso contrario, hay que escoger un **servicio externo** que haga estas funciones, por lo que en esta fase no será posible especificar nada concreto del mismo.

En esta fase solo hay que dejar claro si el acceso programático a los secretos se hará desde el código de aplicación o desde el proceso de construcción de un pipeline CI/CD (si se usase este tipo de tecnología para el desarrollo).

> ***A nivel de requisitos, lo fundamental es establecer QUÉ debe ir en una secret store, no cómo se guardará.***

3.4.3. RS4.3 Prohibir terminantemente el "*hardcoding*" de secretos

En línea con el requisito anterior, este requisito básicamente **obliga a que se use una *secret store***, asumiendo que en cualquier *software* que creemos se usará algún dato que se pueda calificar como privado. Esto es así porque se debe **prohibir terminantemente el "*hardcoding*"** de datos que puedan ser sensibles en la aplicación.

> ***No es algo a tomar a la ligera, porque hay muchos problemas que han surgido por olvidar datos que se incluyeron en el código "solamente para pruebas" y "temporalmente". Incluso este tipo de datos de depuración deben ir en la secret store definida como requisito.***

3.4.4. RS4.4 Asegurarse de que no hay ningún dato sensible en los comentarios de la aplicación

Enlazando con los dos requisitos anteriores, obviamente los datos secretos como cadenas de conexión, claves, etc. **no deben ir nunca en comentarios**, por el riesgo de que salgan a producción. Es más, a nivel de requisitos se puede establecer que el código fuente que salga a producción se procesará con alguna herramienta automática para **eliminar cualquier comentario del *front-end*** de la página que se despliegue.

> ***En otras palabras, los comentarios quedarán en el código en desarrollo, pero su salida a producción requerirá una "purga" de los mismos.***

Ten en cuenta que esto tiene más ventajas además de las de seguridad:

- **No se delatará tan fácilmente la tecnología de desarrollo usada** (que nos puede llevar a la situación de los CVE vista antes).
- **Las páginas tendrán un tamaño más pequeño**, cargando más rápido.

3.4.5. RS4.5 Usar solo *prepared statements* (con *parametrized queries*) para el acceso a datos

Este requisito está aquí para asegurarse de que **ningún usuario o aplicación pueda interactuar directamente con la base de datos**, como medio de prevención de ataques de ***SQL Injection*** **[26]**. Usar este tipo de consultas es una mitigación muy potente contra estos ataques, ya que las entradas del usuario son los parámetros de sentencias preconstruidas y así es muy difícil que puedan cambiar su propósito original.

> ***Esto entra en contraposición con la muy mala práctica de construir sentencias SQL concatenando una parte fija de la sentencia con la entrada del usuario sin validar adecuadamente, típica razón para ser vulnerable a estos ataques.***

Puedes consultar toda más información relacionada con la prevención de ***SQL Injection*** en **[27]** y en **[28]**, que además incluye ejemplos ilustrativos de lo que son y cómo se usan. Además, en la ***Figura 19*** aparecen otras medidas adicionales que puedes tomar al respecto, algunas de las cuales puedes integrar como parte de los requisitos de esta sección.

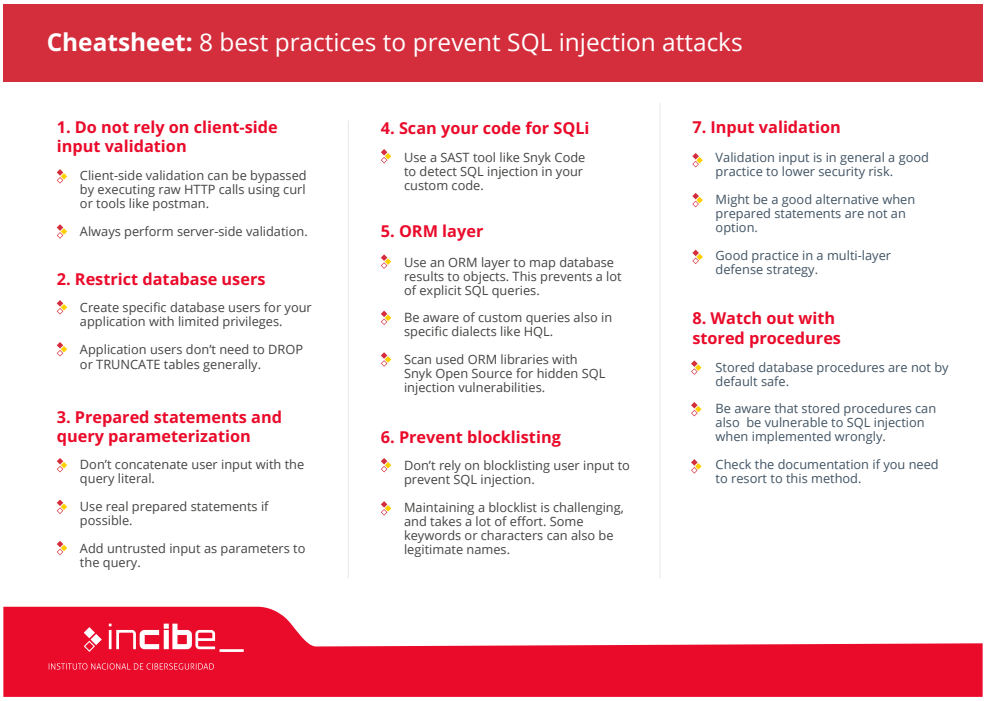


Figura 19. Medidas para prevenir el SQL Injection

Por tanto, a nivel de requisitos debe **forzarse el uso de este tipo de sentencias** en todo momento y establecer **cómo se construyen** en función de los datos, ya que la forma en la que se implementan depende de las funcionalidades del *framework* usado.

3.4.6. RS4.6 No pasar datos importantes en los parámetros de una URL

Los parámetros de las URL de son uno de los puntos donde se pueden filtrar datos inadvertidamente si no tenemos cuidado. Por tanto, a nivel de requisitos se debe establecer **qué parámetros pueden aparecer en URLs** que se usen para navegar a las distintas páginas, y restringirlos solo a **datos irrelevantes** (por ejemplo, el idioma de la página).

Una práctica podría ser establecer como requisito una **lista de posibles parámetros permitidos en las URLs y su significado**, de forma que posteriormente se pueda validar fácilmente si el código de la aplicación cumple con esta lista de parámetros permitidos y detectar fácilmente posibles discrepancias.

De otra forma, no solo le estaríamos dando facilidades a los usuarios maliciosos, sino que el valor de estos datos privados quedaría registrado en el *log* de la aplicación y podría ser leído por otros sistemas que los procesen. Como veremos posteriormente, esto es inaceptable a nivel de seguridad. Puedes consultar **más información** acerca de este problema y sus consecuencias en **[29]**.

3.5. RS5 Requisitos tecnológicos y de revisión del *software*

3.5.1. RS5.1 Usar todas las cabeceras de seguridad aplicables si se trata de una aplicación que use el protocolo HTTP

Uno de los elementos de seguridad más importantes para conseguir una capa de seguridad adicional si nuestra aplicación usa el protocolo HTTP es el **uso de las cabeceras de seguridad HTTP**.

> ***HTTP tiene definidas una serie de cabeceras de seguridad que permiten proteger a la aplicación de distintos ataques, siendo el más relevante el conocido XSS [30].***

Por ello, y dado que todo *framework* de desarrollo web moderno ya debería incorporar soporte para ellas, se debe estudiar primero qué cabeceras están en la actualidad en uso (al ser un tema que tiene novedades periódicas **[31]**) y cuáles son las que más nos conviene usar o no en función de lo que vamos a hacer o no **[32]**. Cuidado, no obstante, con las cabeceras que están **marcadas como obsoletas y van a ser retiradas**.

Se recomienda sobre todo consultar el apartado de ***Content Security Policy*** o CSP **[33]**, cuyo funcionamiento aparece ilustrado en la ***Figura 20***.

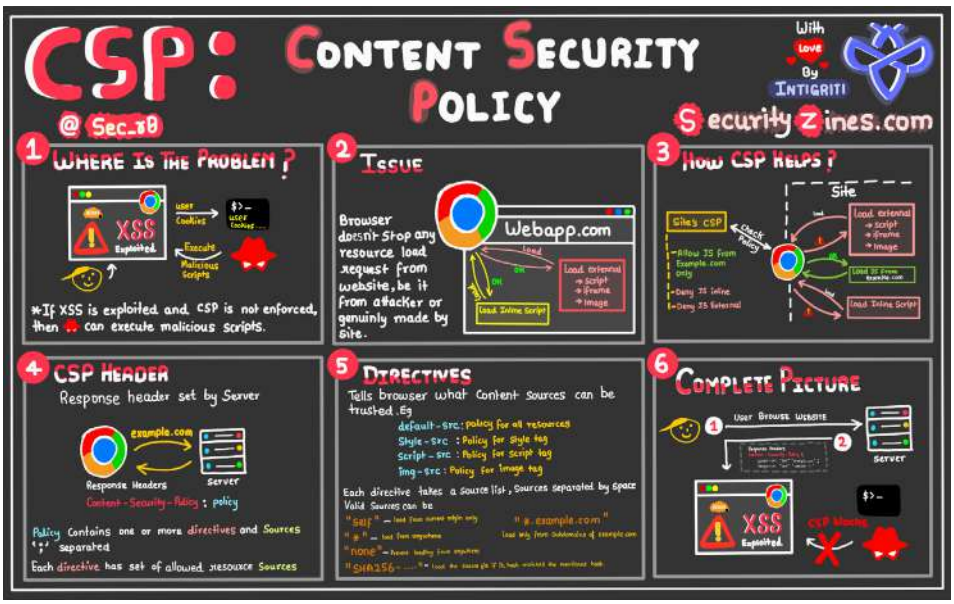


Figura 20. Esquema general del funcionamiento de CSP de securityzines.com

Debemos también tener en cuenta sus posibles aplicaciones **[34]** y valores (***Figura 21***).

Directiva	Ejemplo	Descripción
default-src	'self' cdn.redondo.com	Política por defecto para cargar JavaScript, imágenes, CSS, fuentes, peticiones AJAX, Frames o contenido HTML5 si falta una especifica
script-src	'self' js.redondo.com	Especifica orígenes válidos de JavaScript
style-src	'self' css.redondo.com	Especifica orígenes válidos de CSS
img-src	'self' img.redondo.com	Especifica orígenes válidos de imágenes
connect-src	'self'	Se aplica a XMLHttpRequest (AJAX), WebSocket o EventSource. En caso de no permitirse, el navegador emula un HTTP 400 Bad Request
font-src	fonts.redondo.com	Especifica orígenes válidos de fuentes
object-src	'self'	Especifica orígenes válidos de plugins (como los tags <object>, <embed> o <applet>)
media-src	multimedia.redondo.com	Especifica orígenes válidos de audio y video (como los tags de HTML5 <audio> y <video>)
frame-src	'self'	Especifica orígenes válidos para cargar frames. Está obsoleta y debe reemplazarse por child-src
sandbox	allow-forms allow-scripts allow-popups	Implementa una sandbox para el recurso asociado que aplica una same origin policy única, elimina popups y bloquea la ejecución de plugins y scripts. Si se deja vacía, aplica todas las restricciones, pero permite desactivarlas individualmente con los valores: allow-forms, allow-same-origin, allow-scripts, allow-popups, allow-modals, allow-orientation-lock, allow-pointer-lock, allow-presentation, allow-popups-to-scape-sandbox and allow-top-navigation
report-uri	/reports	Le dice al browser que mande informes de violaciones de la poliica vía POST a esa URL
child-src	'self'	Especifica orígenes válidos de web workers y nested browsing contexts cargados con tags como <frame> o <iframe>
form-action	'self'	Sitios válidos que pueden usarse como action de una <form>
frame-ancestors	'none'	Especifica orígenes válidos de recursos usados desde tags <frame>, <iframe>, <object>, <embed>, <applet>. El valor none es aprox. equivalente a X-Frame-Options: DENY
plugin-types	application/pdf	Define tipos MIME válidos para los plugins cargados vía tags <object> o <embed>

Posibles valores de las directivas *(se pueden combinar varios separados por un espacio)*

Valor	Ejemplo	Descripción
*	img-src *	Cualquier URL que no use los esquemas data: blob: o filesystem:
'none'	object-src 'none'	Bloquea el uso de ese tipo de recurso
'self'	script-src 'self'	Permite cargar desde el mismo origen (mismo esquema, host y puerto)
data:	img-src 'self' data:	Permite la carga de URLs con el esquema data: (ej: imágenes codificadas en Base64)
domain.redondo.com	img-src domain.redondo.com	Carga el recurso desde el dominio indicado
*.redondo.com	img-src *.redondo.com	Carga el recurso desde cualquier subdominio de redondo.com
https://cdn.com	img-src https://cdn.com	Solo carga el recurso desde el dominio indicado, y solo bajo HTTPs
https:	img-src https:	Solo permite la carga de recursos bajo HTTPs, aunque cualquier dominio
'unsafe-inline'	script-src 'unsafe-inline'	Permite usar orígenes de recursos en línea, como atributos style, onclick o tags <script> (depende del contexto del origen). También URLs con esquema javascript:
'unsafe-eval'	script-src 'unsafe-eval'	Permite la evaluación dinámica de código, con funciones como eval() de JavaScript
'nonce-'	script-src 'nonce-azaj345'	Solo ejecuta un script si tiene un atributo nonce cuyo valor es idéntico al de la cabecera. Ej.: <script nonce = "azaj345"> alert(...);</script>
'sha256-'	script-src 'sha256-qznLcsR0x$GACP2dm0UCKCzCG+HiZ1guq6zZDob/Tng='	Ejecuta un style o un script si su hash es la especificada con el algoritmo indicado. No funciona en URLs javascript:. El hash del ejemplo ejecuta ('Hello, world');

Figura 21. Posibles directivas y valores de una cabecera CSP

Otra de las cabeceras HTTP a las que deberíamos prestarle especial atención es la que permite limitar el acceso a periféricos y funciones del dispositivo del cliente. Esta es la ***Permissions-Policy*** **[35]**, que nos permite establecer programáticamente que nuestra aplicación **no va a hacer uso de sensores, periféricos**, etc. (***Figura 22***) que no va a necesitar, por si es manipulada maliciosamente para hacerlo.

Directiva	Función
ambient-light-sensor	Obtener información acerca de la cantidad de luz ambiente en el entorno del dispositivo usando AmbientLightSensor
autoplay	Autorreproducir elementos multimedia cargados con HTMLMediaElement. El atributo autoplay de tags <audio> y <video> se ignora
accelerometer	Obtener información acerca de la aceleración de un dispositivo a través de Accelerometer
Battery	Controlar si se permite usar la Battery Status API
camera	Usar dispositivos de entrada de video mediante getUserMedia()
display-capture	Usar getDisplayMedia() para capturar pantallas
document-domain	Modificar document.domain
encrypted-media	Usar el API Encrypted Media Extensions (EME) mediante la función Navigator.requestMediaKeySystemAccess()
execution-while-not-rendered	Controlar si las tareas se deben ejecutar en los frames cuando estos no se están renderizando
execution-while-not-rendered	Controlar si las tareas se deben ejecutar en los frames cuando estos están fuera del área visible por el usuario
fullcreen	Usar la función Element.requestFullScreen()
geolocation	Usar el API el Geolocation y funciones como getCurrentPosition() y watchPosition()
gyroscope	Obtener información de la orientación del dispositivo mediante Gyroscope

Directiva	Función
magnetometer	Obtener información de la orientación del dispositivo mediante Magnetometer
microphone	Usar dispositivos de entrada de audio con la función MediaDevices.getUserMedia()
midi	Usar la Web MIDI API y la función Navigator.requestMIDIAccess()
payment	Usar la Payment Request API y su constructor PaymentRequest()
picture-in-picture	Reproducir un video en modo Picture-in-Picture
speaker	Reproducir audio con cualquiera de los métodos que lo permiten
sync-xhr	Hacer peticiones XMLHttpRequest síncronas
usb	Usar el WebUSB API
wake-lock	Usar la Wake Lock API para evitar que los dispositivos entren en modos de bajo consumo
webaulthn	Usar la Web Authentication API para crear, guardar y leer credenciales basadas en criptografía de clave pública
vr	Usar la WebVR API y el método Navigator.getVRDisplays()
xr-spatial-tracking	Usar la WebXR Device API para interactuar con una sesión WebXR

Figura 22. Elementos a los que podemos regular el acceso con la cabecera HTTP Permissions-Policy

Una vez que hemos decidido qué cabeceras son las más convenientes en nuestro caso, debido a que es un campo que está en evolución, se puede fijar como requisito **examinar periódicamente el dominio** en producción con alguna herramienta que verifique si su implementación sigue siendo adecuada, teniendo en cuenta los cambios que se puedan haber producido a su definición y especificación, y actuar en consecuencia en caso negativo. Una de las herramientas gratuitas que se pueden usar para eso es ***Security Headers*** (***Figura 23***).

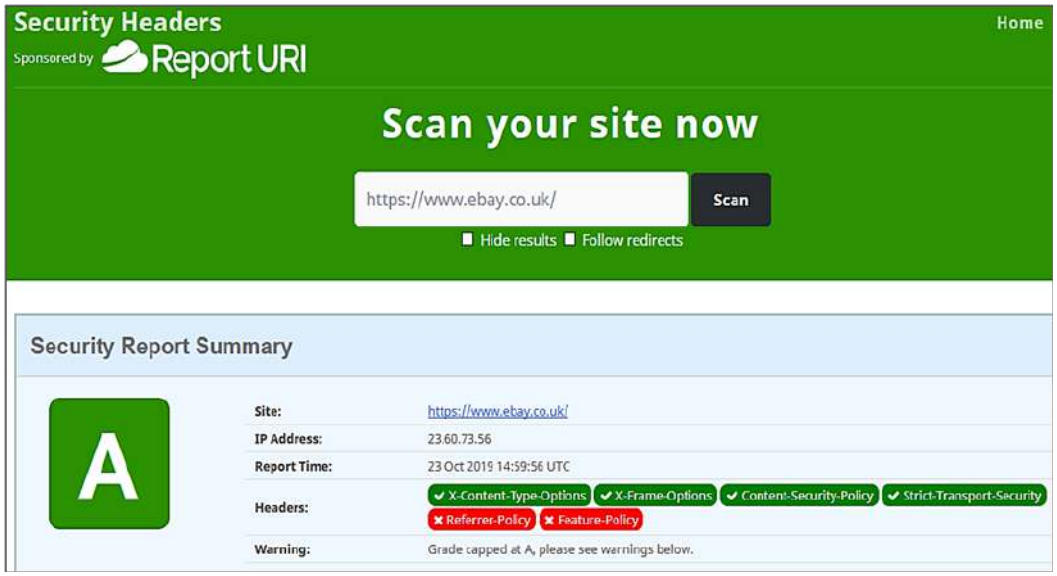


Figura 23. Resultado de un escaneo de Security Headers

3.5.2. RS5.2 Usar una configuración de *cookies* segura

Las *cookies* son el mecanismo que se integró en el protocolo HTTP para poder tener estado y, al residir su contenido en el cliente, deben establecerse unos requisitos claros para manejarlas de forma segura. Ten en cuenta que **no es lo mismo una *cookie* que el local *storage* del navegador**. Ambos tienen unas reglas de seguridad **muy distintas [36]**. Si se van a usar *cookies*, los requisitos a seguir para ellas son:

- Todo lo que llegue al servidor que sea parte del contenido de una *cookie* **debe ser validado** antes de usarse, y rechazarse si no es correcto. Las *cookies* residen en el cliente, y por tanto **pueden ser manipuladas maliciosamente**.
- El **ID de sesión** debe pasarse **SIEMPRE** en una session *cookie*, nunca en una *persistent cookie* (también llamada *tracking cookie*). Una *session cookie* se destruye al final de una sesión.
- Las *cookies* deben enviarse **siempre bajo conexiones cifradas** (*Flag Secure*, `Set-Cookie: Secure`).
- No debe poder **accederse a las *cookies* desde *JavaScript*** (*Flag HTTPOnly*, `Set-Cookie: HttpOnly`). Esto previene algunos vectores de ataque XSS.
- Las *cookies* persistentes **deben caducar** (atributo `Expires`).
- **Establecer un Dominio (*Domain*):** Por defecto las ***cookies* de un dominio solo pueden ser accedidas por webs alojadas en dicho dominio**. La configuración es segura por defecto (*secure by default*), pero si deseamos que otros dominios lean nuestras *cookies*, podemos hacerlo con el atributo `Domain` (Ej.: `Set-Cookie: Domain: ads.redondo.com`)

  En el ejemplo anterior, si en lugar de eso ponemos `redondo.com`, cualquier página de ese dominio podría leer nuestra *cookie*. Se recomienda dejar la opción por defecto o restringir los dominios permitidos al máximo como requisito.
- **Atributo `Path`:** Muchas aplicaciones realmente son un **conjunto de aplicaciones distintas que forman una mucho más grande**, todas bajo una misma URL base. Si este es tu escenario, cada cookie debería ser leída solo por su "trozo" de aplicación como requisito. Cada uno de esos "trozos" se especifica con su ruta dentro del a URL (Ej.: `Set-Cookie: path=/contabilidad`).

  Si la aplicación que vamos a construir se prevé que esté compuesta por un conjunto de aplicaciones muy independientes, conviene que sus cookies se separen entre ellas de esta forma.
- **Atributo `Same-Site`:** Es una **medida contra ataques CSRF**.

  El valor `Strict` (recomendado) hace que las *cookies* solo puedan provenir del mismo sitio que las usa, no de un enlace o de otra página (`Set-Cookie: same-site=Strict`). El valor `Lax` bloquea las *cookies* de otras páginas que vengan vía POST, pero permite enlaces.

- ***Prefijos de Cookie*:** Es una medida de defensa en profundidad que asegura que solo determinados subdominios puedan acceder a una *cookie*. Más información

> ***Lo cierto es que las cookies tienen muchas opciones y manejarlas correctamente puede resultar lioso al principio. Un resumen de todo lo necesario para manejar bien cookies para el desarrollo web lo podemos encontrar en [37], lo que nos puede ayudar a perfilar nuestros requisitos.***

La infografía de la ***Figura 24*** resume también un conjunto de información que debes tener en cuenta a la hora de entender y modelar *cookies*. Además, en **[38]** puedes encontrar un *white paper* completo acerca de la seguridad de *cookies* que puedes usar para facilitar establecer tus requisitos respecto a las mismas.

No obstante, en la fase de requisitos también tenemos que modelar como vamos a gestionar las *cookies* en la web y mostrar todos los avisos requeridos legalmente para hacerlo correctamente. En el **INCIBE**, **[39]** describe las obligaciones existentes con las *cookies*, y puedes seguir esa guía para elaborar tus requisitos. También muestra una política de transparencia total con las *cookies* que usa la página que puedes usar como modelo para la aplicación a desarrollar **[40]**.

Figura 24. Aspectos variados sobre las cookies

3.5.3. RS5.3 Usar las características de seguridad de los *frameworks* elegidos

Hoy en día todos los *frameworks* para la creación de aplicaciones (y más concretamente los aplicaciones web) **vienen con una serie de características de seguridad** (validación, medidas contra ataques conocidos...) que nos pueden ayudar a implementar todos los requisitos de esta sección 3 de una manera **correcta, probada y validada**.

Por ello, de cara a la elaboración de los requisitos de nuestra aplicación, tiene mucho sentido hacer un estudio de los posibles *frameworks*, librerías o, en general, **cualquier *software* de terceros que podamos usar**, apuntar cuáles de sus características pueden sernos más útiles para implementar los requisitos de seguridad que estamos viendo, y así tener claro dónde acudir para facilitar su implementación.

La cantidad y calidad de las medidas de seguridad de un framework es otro factor clave de cara a elegir cuál es el mejor para crear las distintas partes de la aplicación. Si el código que uses se toma la seguridad en serio, mejorará la seguridad del conjunto de la aplicación.

3.5.4. RS5.4 Usar la última versión estable, soportada y actualizada de los *frameworks*

Aunque en este punto todavía no hayamos elegido el *framework* de desarrollo que vamos a usar para cada una de las partes del *software* a construir, sí que es posible que tengamos ya una **lista de posibles candidatos a estudiar**. Por ello, podemos establecer como requisito que, en todos los *frameworks*, librerías o aplicaciones de terceros posibles a elegir, **siempre debemos usar una versión estable** con soporte que se extienda mucho más allá de la fecha prevista de despliegue de la aplicación.

Esto es muy importante, puesto que, de cara a mantener un negocio operativo con éxito, la aplicación y todo el *software* adicional tendrá que mantenerse **estable y mantenida durante mucho tiempo**. Por ejemplo, si parte de nuestra aplicación va a estar basada en *WordPress*, no tiene sentido escoger una versión que solo tenga soporte activo durante 9 meses desde la fecha en que lo seleccionamos.

Si parte de las herramientas que usamos para dar nuestro servicio pierden el soporte de su fabricante, tendremos problemas graves que se deberán solucionar, costándonos tiempo y recursos, que es algo que estamos tratando de minimizar en este contexto de start-ups. Por ejemplo, imagina que te quedas sin soporte de tu versión de React en medio de una ronda de captación de inversiones...

Además de lo dicho, conviene establecer como requisito que **no se permitirá el uso de versiones alfa, beta o similares de ningún producto**. Este es un criterio prioritario al de usar la última versión disponible. Hablaremos más de cómo implementar este requisito aquí.

Más vale un software probado y estable que no el último disponible en este contexto.

3.5.5. RS5.5 Mantener las dependencias actualizadas

Se debe definir como requisito establecer **procedimiento para hacer una revisión periódica de todas nuestras dependencias** (*software* de terceros que usemos en cualquier punto de la aplicación) de forma correcta y predecible.

El control de nuestras dependencias es de importancia capital para mantener una seguridad en toda la aplicación a lo largo de su ciclo de vida, y no estar trabajando con versiones obsoletas que puedan poner nuestro negocio en riesgo, al ser algo seguramente fatal para la start-up.

No obstante, en este punto solo vamos a contemplar la necesidad de hacer una revisión periódica y no cómo hacerlo, puesto que se describirá más adelante en esta guía.

3.5.6. RS5.6 Asegurarse de que se usan herramientas de seguridad en la aplicación antes de su puesta en producción

Por lo menos debe establecerse como requisito que las aplicaciones web o expuestas al público general (Internet) **se escaneen con herramientas de detección automática de vulnerabilidades y errores**, tanto antes de salir a producción como periódicamente, de cara a poder detectar posibles problemas que puedan aparecer con el tiempo.

Mejor lo hacemos nosotros antes de que lo haga algún atacante malicioso que pueda comprometer nuestra aplicación...

Aunque mencionaremos aplicaciones que nos puedan dar esta funcionalidad de manera gratuita más adelante, conviene fijar como requisito **este tipo de operaciones periódicas de mantenimiento**, dada su importancia, para no olvidarlas más tarde.

3.5.7. RS5.7 Hacer *code review* de la aplicación antes de que sea puesta en producción

Se debe definir como requisito un procedimiento para hacerlo de forma correcta y predecible. La revisión de código antes de la salida producción **puede ayudar a la detección de vulnerabilidades**, o de funcionalidades que no se han implementado correctamente y puedan comprometer la seguridad de nuestra aplicación (***Figura 25***).

Si bien es demasiado pronto para especificar las herramientas con las que se puede hacer, sí que conviene fijar como requisito que este tipo de operaciones se van a realizar, cuándo y, en general, qué tipo de cosas se van a revisar si es posible. En **[41]** tenemos ayuda de cómo hacer estos procesos de revisión.

Figura 25. Beneficios generales de invertir en hacer code reviews

3.5.8. RS5.8 Asegurarse de que la aplicación captura todos los errores

Y que también trata los fallos adecuadamente, sin caer en estados inesperados. Los errores públicos (que ve el usuario) **no deben tener NUNCA trazas de pila o errores de BBDD**, y deben capturarse de manera que **no se dé información técnica al usuario**, como ya hablamos antes en esta guía. Si el error es parte de una transacción, ésta debe deshacerse y cerrarse (***fail-closed***), estableciendo cómo requisito qué ocurre en este caso.

Por otro lado, si tenemos la suerte de poder integrar nuestro producto en el ámbito de un **SIEM** o una herramienta de vigilancia existente (ahorrándonos así el coste de desplegar una), deben establecerse como requisito escribir los *logs* en el lugar y con el formato admitidos por el mismo para poder analizarlos. Tampoco es mala idea hacerlo igualmente en un formato ampliamente usado para facilitar el uso de estas herramientas en el futuro. Hablaremos de una herramienta de esta clase que podríamos usar en esta sección.

Está claro que, debido al contexto de una start-up con el que trabajamos en esta guía, probablemente no tengamos ni el tiempo ni los recursos económicos para montar un SIEM que vigile nuestros sistemas y la aplicación desarrollada pero, si contamos con una implementación ya en marcha que nos pueda ayudar, entonces debemos hacer todo lo posible desde la fase de requisitos para interaccionar con ella y maximizar sus servicios, lo cual muchas veces pasa por escribir nuestros logs en un formato y lugar correctos.

3.5.9. RS5.9 Deshabilitar el cacheo en páginas que contienen información privada

Esto se puede hacer a través de la cabecera HTTP correspondiente. Debido a que esto puede facilitar la filtración de datos importantes, es también un requisito deseable **especificar qué funcionalidades concretas dentro de cada parte de la aplicación no van a contemplar el cacheo de páginas** para que, de esta manera, cuando use máquinas compartidas u otras situaciones, un usuario pueda ver datos privados de otro por accidente, debido a un mal manejo de este tipo de mecanismos.

Como veremos posteriormente, uno de los tipos de páginas qué no deberían cachearse son todas aquellas en las que se introducen usuarios y contraseñas, por ejemplo. La ***Figura 26*** muestra el aspecto típico de una ventana de inicio de sesión con los elementos necesarios situados correctamente, que puede servirte de ayuda para diseñar las tuyas:

- **No se han cacheado ni nombre de usuario ni contraseña**. *¿Has pensado en el peligro que supondría que todo el mundo usase la misma cuenta de usuario en el sistema operativo que se use para acceder a tu aplicación? ¿Quieres prevenirlo? ¿O asumes ese riesgo?* (esto está más relacionado con el *threat modeling* que veremos en la fase de diseño).
- Inmediatamente bajo el nombre de usuario un **enlace para recuperarlo**.
- Una recomendación de **acceder por navegación privada** si el dispositivo no es del usuario, para que así el navegador no recuerde sus datos.
- **Un enlace a crear una cuenta accesible** desde la pantalla de *login*, por si un usuario nuevo llegó ahí por error.
- **Un *login* en dos pasos:** Si el nombre de usuario tiene algún problema (no existe, está bloqueado, etc.) ni siquiera se da la opción de introducir una contraseña.

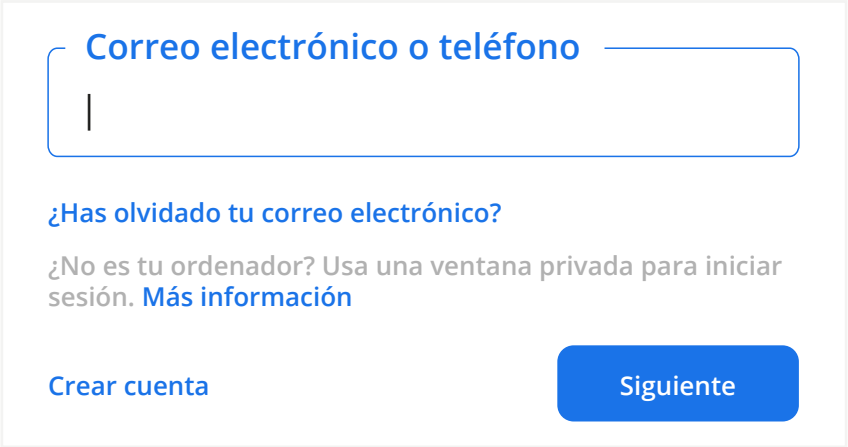


Figura 26. Ejemplo de inicio de sesión con todos los elementos necesarios

Si bien el mecanismo para impedir el cacheo de páginas depende de las tecnologías con las que vamos a desarrollar la aplicación, es bueno especificar como requisito qué partes de la aplicación no van a contemplar el cacheo de antemano, para evitar luego sorpresas desagradables en ese sentido.

3.6. RS6 Requisitos de cuentas de usuario, autenticación y autorización

3.6.1. RS6.1 Asegurarse de que todas las cuentas usadas en la aplicación son cuentas de servicio

Cuando nuestra aplicación haga peticiones a alguna parte de su sistema operativo, lo hará en nombre de una identidad de usuario. En esos casos, un requisito a establecer es que nunca debe usar cuentas de usuario interactivo (es decir, de usuarios que puedan entrar en sesión) para ello, sólo cuentas de servicio que tenga el SO.

Las cuentas de servicio son cuentas de usuario que están destinadas a ser usadas por el sistema, nunca usuarios "humanos". Por ejemplo, si una aplicación necesita acceder a una base de datos, nunca debe hacerlo en nombre de un usuario "humano" que pueda hacer login. De esta forma, si un empleado se va, no se compromete la aplicación por no poder acceder a través de su cuenta si esta se borra.

Si desarrollamos varias aplicaciones o sub-aplicaciones, **cada aplicación debe tener su propia cuenta de servicio para este tipo de accesos como requisito**. Esto facilita separar su monitorización respecto a las otras. Además, así se distingue perfectamente lo que hace cada aplicación de lo que hace cada usuario y pueden restringirse sus permisos al máximo, **algo que es SIEMPRE aconsejable**.

Se pueden encontrar **instrucciones** de cómo crear cuentas de servicio en proveedores de nube y *on-premises* fácilmente, que podemos aplicar cuando hayamos seleccionado la arquitectura.

3.6.2. RS6.2 Fomentar que todos los usuarios de la aplicación usen *password managers* y prohibir la reutilización de claves

Idealmente, todos los usuarios de nuestra aplicación deberían usar un **gestor de contraseñas** (*password manager*) (Ej.: ***KeePass***) para evitar recordar muchas *passwords* (minimizando el riesgo de olvidarlas) y **asegurarse de que todas sean complejas**. Otras ventajas que tienen algunos de ellos (y que serán aspectos claves a la hora de seleccionar el gestor con el que trabajar) son comprobar automáticamente si una cuenta **forma parte de una brecha de datos conocida**.

Si bien establecer como requisito esto para los clientes de nuestro software es, en la mayoría de los casos, inviable (no podemos controlar lo que hacen), sí que es algo viable (y exigible, por las ventajas que hemos mencionado) que lo hagan los miembros de nuestra start-up.

Podemos establecer como requisito que solo las personas que tengan un gestor de contraseñas configurado de la forma adecuada (pudiendo establecer ahora cuál es esa forma) podrán interactuar con el proceso de desarrollo. Incluso se puede prefijar un gestor de claves concreto si, tras una comparativa, se determinase uno como el más adecuado para nuestro contexto. El **INCIBE** tiene documentación acerca de cómo son los gestores de contraseñas si necesitas investigar más su funcionamiento antes de diseñar sus requisitos **[42]**.

3.6.3. RS6.3 Forzar el uso de claves largas (*pass phrases*) y con un mínimo de complejidad

Esta funcionalidad la da un *password manager* **[43]** pero, como hemos dicho, no podemos exigir a un cliente que use uno. Lo que si podemos es **establecer como requisito de la aplicación que no admita contraseñas que no pasen una serie de reglas** que aseguren su robustez (longitud, complejidad...), usando para ello protocolos y procedimientos bien establecidos en la actualidad **[44]**, o bien funcionalidades existentes en el *framework* de desarrollo o *software* fiable de terceros que evalúe las nuevas contraseñas introducidas.

3.6.4. RS6.4 Verificar que la clave de un usuario no ha sido ya filtrada usando un servicio externo que se encarga de recopilar filtraciones

Esto también lo dan ya muchos gestores de contraseñas, pero se puede considerar forzar en la aplicación con el uso de APIs como la de ***haveibeenpwned*** que vimos anteriormente si nuestros recursos lo permiten.

3.6.5. RS6.5 Activar MFA en todas las cuentas consideradas importantes

Los esquemas MFA se basan en que los usuarios aporten 2 de 3 "algos" (***Figura 27***):

- Algo que **Saben**, como una clave o PIN de una cuenta.
- Algo que **Tienen**, como un dispositivo físico (móvil) o una aplicación que genera OTPs (*One-Time Passwords*).
- Algo que **Son**, una característica biológica única como una huella, la voz o la retina.

Figura 27. Típico conjunto de elementos que implementa cualquier MFA

Si no tienes claro qué es un MFA, el INCIBE te lo explica aquí **[45]**. Hay muchas formas de implementar MFAs y todas tienen sus desventajas, pero todas incrementan la seguridad de las cuentas. Todos los métodos **fuerzan al usuario a introducir una 2º clave** (***Figura 28***) tras la 1º para acceder al sistema, y esta segunda clave **se genera dinámicamente y cambia constantemente**.

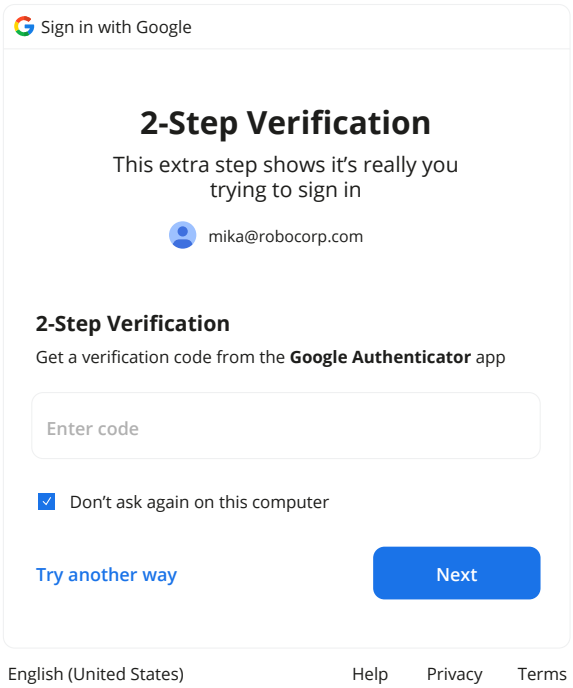


Figura 28. Pantalla típica para introducir un MFA

De los métodos usados actualmente (***Figura 29***) Se recomienda evitar el uso de métodos que estén en el nivel 5 (lo que incluye el token SMS, por Email y la verificación de identidad por teléfono) o inferior, para tener un nivel de seguridad adecuado.

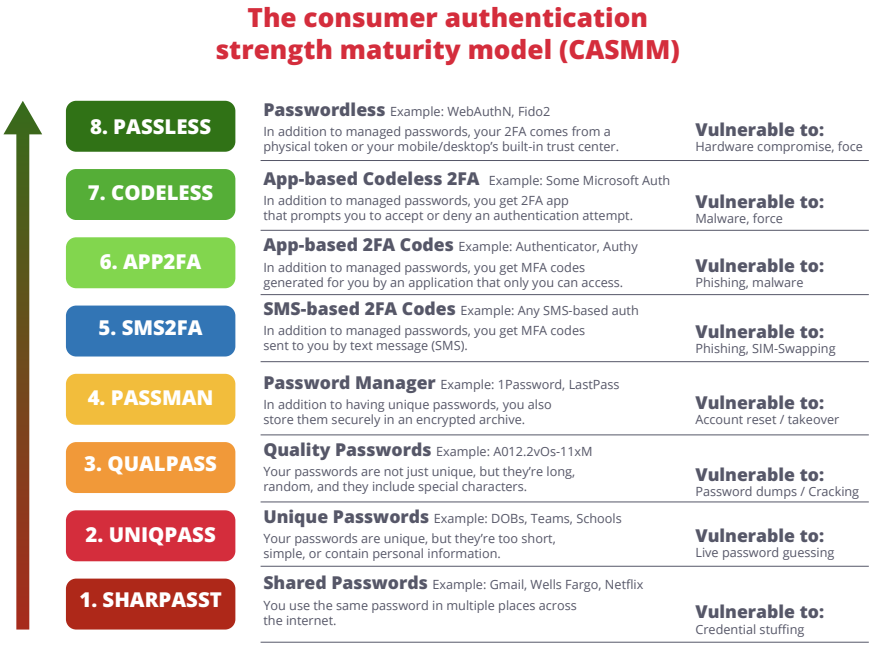


Figura 29. Modelo CASMM v6 de Daniel Messler para la autenticación

Por tanto, esquemas MFA aceptables que podemos incorporar a nuestra aplicación son:

- ***Token hardware:*** los usuarios tienen un dispositivo (como un USB) que contiene un token de acceso generado dinámicamente y que debe conectarse al PC. *Google Titan* es una aproximación similar, pero usando una **infraestructura PKI**.

  Es un método cuya popularidad está en aumento, y ya hay dispositivos baratos que cumplen con los **protocolos FIDO**. No obstante, obligaríamos a los clientes a adquirir un dispositivo de esta clase, lo cual puede hacernos perder clientes potenciales.

- ***Token software:*** Los usuarios instalan una aplicación gratuita que genera tokens dinámicos. Este es el **método recomendado por tener mejor relación coste / beneficio**. Requieren la instalación de un programa de terceros (algo que podría estar restringido en ciertos entornos) que, si se compromete, entonces nuestro sistema de autenticación también lo estará. Ejemplos: *Google Authenticator, Microsoft Authenticator* (gratuitos)

- **Verificación biométrica:** el usuario es el *token*. Aunque pueda parecer el método más seguro, guardar datos biométricos (Ej.: Huellas) plantea **muchas dudas sobre privacidad** y se pueden suplantar usuarios con datos robados. Por otro lado, requiere dispositivos especiales para verificarlos: Cámaras, *scanners*... que suponen un coste adicional.

  En un entorno de *start-up* requerirían una inversión de tiempo (para guardar correctamente los datos biométricos) y económica (para los dispositivos) que no encaja bien con los objetivos de esta guía.

Teniendo en cuenta coste / beneficio y que estamos asumiendo un entorno que favorezca la puesta en marcha de una start-up minimizando el coste, el token software es el método más recomendable.

Si un MFA además se combina con **externalizar la autenticación a un proveedor de terceros**, como aconsejamos anteriormente, y este proveedor ya lo implementa, **nos liberamos de hacerlo nosotros**, ahorrando costes. Podemos tener así un alto nivel de seguridad sin asumir un coste significativo por ello.

A modo de ejemplo, la ***Figura 30*** muestra la forma usada por *Google Authenticator* para habilitar MFA, que es similar a otras aplicaciones del mismo tipo (usa nºs aleatorios generados por el algoritmo TOTP (*Time-Based One Time Password*). **Más información**.

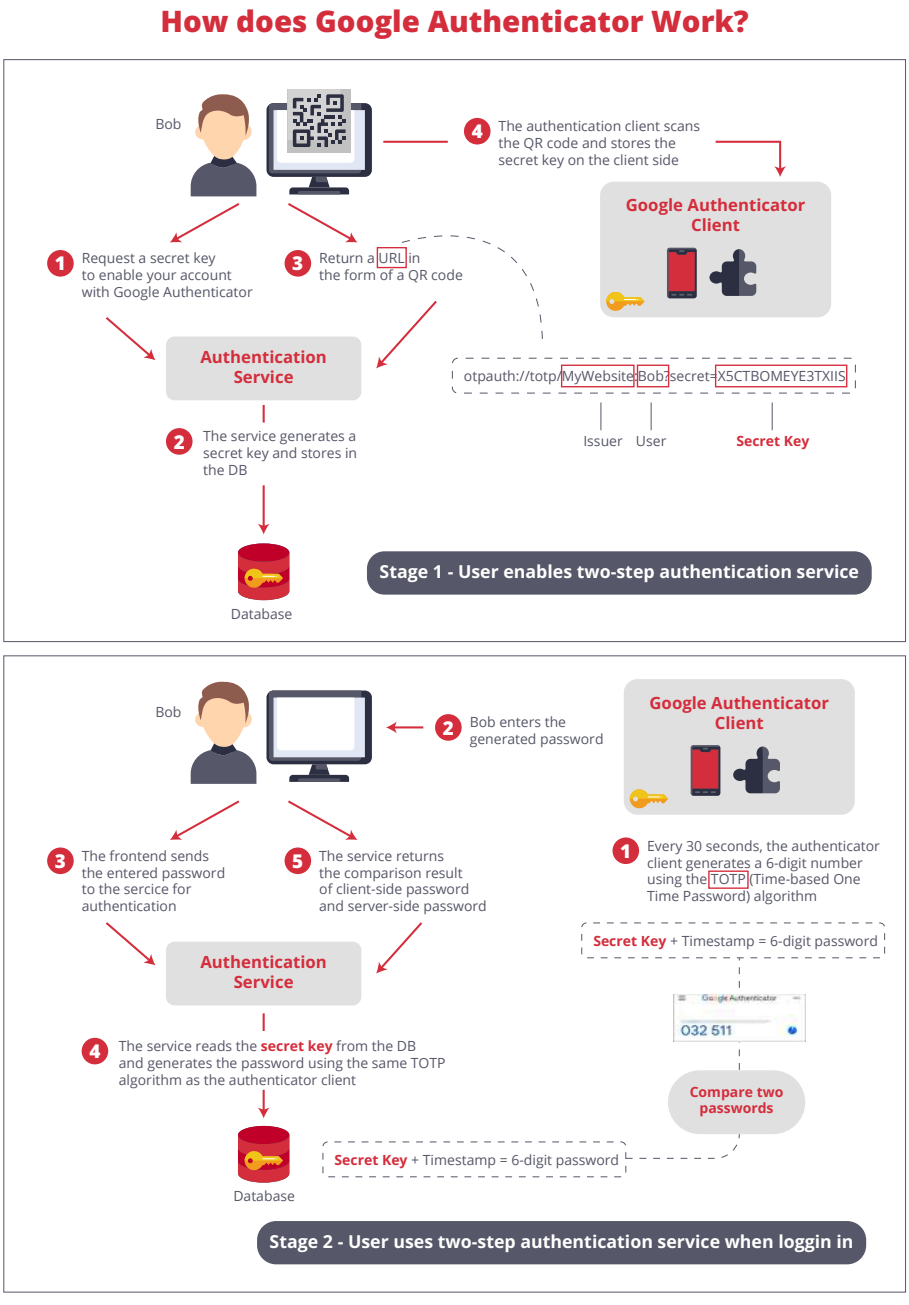


Figura 30. Esquema de cómo funciona Google Authenticator

3.6.6. RS6.6 Requisitos de cambio de claves

Tal y como se dijo en el RS2.3, cambiar periódicamente la clave de forma forzosa está desaconsejado, y **solo debe forzarse tras detectar una brecha de datos**. En esos casos, si como requisito del *software* tenemos que implementar un reinicio de claves porque el sistema de autenticación es nuestro, el siguiente algoritmo (***Figura 31***) indica **cuándo se puede usar dicha funcionalidad para perpetrar un ataque** contra nuestra aplicación y, por tanto, podemos fijar como requisito el control de las situaciones que indica para producir una alerta.

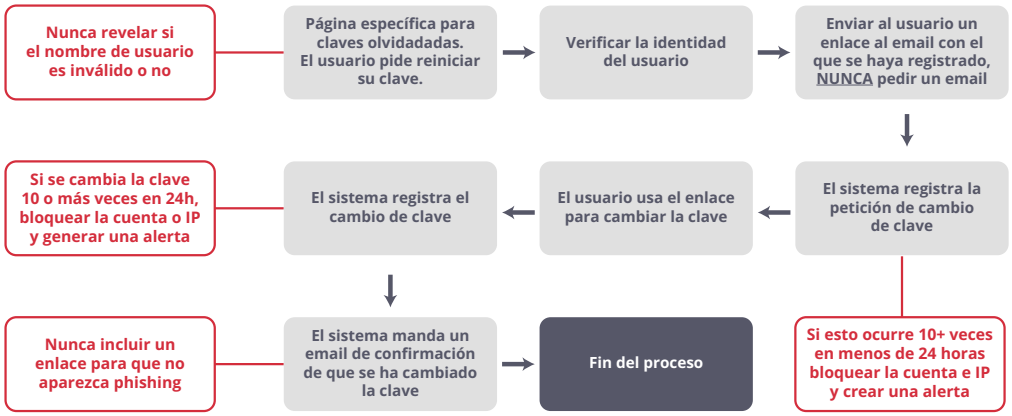


Figura 31. Algoritmo que indica cuando un cambio de contraseña puede ser usado como método ofensivo

> ***Si seguimos el consejo de externalizar toda la gestión de autenticación usuarios a un tercero, entonces será ese tercero el que controle la política de gestión de las claves: Su reinicio y recuperación, cambios, 2FA, etc.***

3.6.7. RS6.7 Definir roles de usuario, asignárselos a los mismos y definir los permisos asociados a esos roles

Para la autorización (*AuthZ*) y autenticación (*AuthN*), si la aplicación a diseñar tiene diferentes tipos de acceso o permisos para distintos grupos de usuarios (**roles**), estos **deben definirse durante la fase de requisitos**.

Esto se hace muy frecuentemente con un esquema tipo ***Role-Based Access Control*** **(RBAC)**. En otras palabras, se trata de establecer los roles de los usuarios claramente desde el principio, y especificar lo que pueden o no pueden hacer sobre las distintas partes del sistema. También debe relacionarse con el método de autenticación de los usuarios, es decir, cómo se van a tratar sus identidades, especialmente si hemos establecido que vamos a externalizarlo para ahorrar costes. Si no estás familiarizado con estos conceptos, el **INCIBE** te los explica aquí **[46]**.

La idea que hace al RBAC algo muy adecuado para el contexto de una *start-up*, es que permite asignar permisos a los usuarios de una forma muy sencilla, y que se adapta a los cambios que pueda haber en la aplicación en cuanto a usuarios y lo que pueden hacer. La ***Figura 32*** explica dicha idea:

- A nivel de requisitos **definimos los roles que los usuarios de la aplicación pueden tener**. En una fase temprana, esto es mucho más sencillo de derivar de las ideas que tengamos que pensar en usuarios específicos.
- Una vez definidos los roles, se determina **qué pueden hacer y qué no** sobre las distintas partes de la aplicación. Aunque aún no tengamos dichas partes definidas del todo, sí que podemos tener una idea de los diferentes subsistemas que va a tener la aplicación que queremos diseñar y de las funciones que desempeñarán cada uno de ellos.

Es importante en esta fase definir al menos qué funciones son una mera consulta de información y qué funciones van a consistir en modificar (o eliminar) alguno de los datos guardados, para así tener bien claro cuando un rol puede destruir o alterar determinada información.

- Una vez que tenemos definidos los permisos generales de cada rol en cada una de las partes de la aplicación, sí tenemos información suficiente, podemos incluso **detallar los permisos concretos de cara rol en forma de lista**. Si esto es posible, el diseño e implementación de este requisito se podrá facilitar posteriormente.
- Una vez que hemos creado estos tres elementos, podemos entender correctamente las **ventajas de un sistema de control de acceso basado en roles**:
    - Si un usuario **deja de usar nuestra aplicación por cualquier motivo**, no es necesario que le borremos ningún permiso específico, simplemente tenemos que quitarle el rol o roles que le habíamos asignado y automáticamente perderá todos los privilegios asociados a los mismos.
    - Si por el contrario un usuario **cambia de rol** (o adquiere nuevos), por una promoción o un movimiento de los privilegios asignados a cada uno de ellos, una mera modificación de los roles asignados a cada usuario reasignará los permisos que tiene.

Esta aproximación es mucho más flexible y sencilla de implementar que asignar permisos concretos a identidades de usuario específicas. Cuando esos usuarios dejen de existir por algún motivo (especialmente si dependemos de un sistema externo para definirlos), la aplicación podría ser mucho más difícil de mantener, lo que es contrario a lo que pretendemos en esta guía.

Obviamente, para que todo esto funcione toda la aplicación debe comprobar el acceso a sus características **haciendo uso del rol del usuario actual** en lugar de la identidad concreta del usuario.

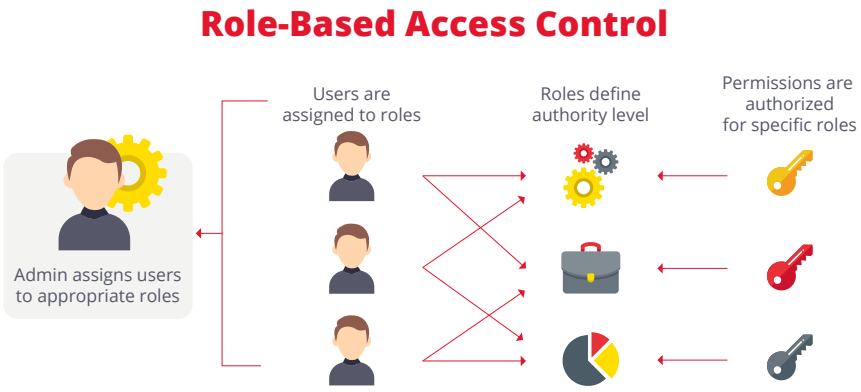


Figura 32. Cómo entender un RBAC

Por tanto, a nivel de requisitos tiene mucho sentido especificar, aunque sea a nivel abstracto, los **distintos roles de usuario** que la aplicación va a tener y, con ellos, qué tipo de permisos se van a conceder, aunque sea describiéndolos a nivel general si en este estadio inicial del proceso de construcción del *software* de la *start-up* aún no podemos dar ningún tipo de indicación concreta.

> ***Por ejemplo, podemos fijar desde el principio que van a existir usuarios con roles de administrador, usuario estándar, contable, alumno, estudiante, etc. y lo que se espera que puedan hacer todos ellos.***

Otra aproximación es el **control basado en atributos (ABAC) [47]**. El ABAC es más dinámico y granular que el RBAC, ya que tiene en cuenta características concretas del usuario a la hora de determinar el acceso a un recurso. No obstante, **resulta más costoso de gestionar**, por lo que en el contexto de esta guía hemos optado por la versión más sencilla en primera instancia.

Tanto RBAC como ABAC pueden extenderse más para hacer un modelo más adaptable a cada necesidad, como se puede ver aquí (***Figura 33***).

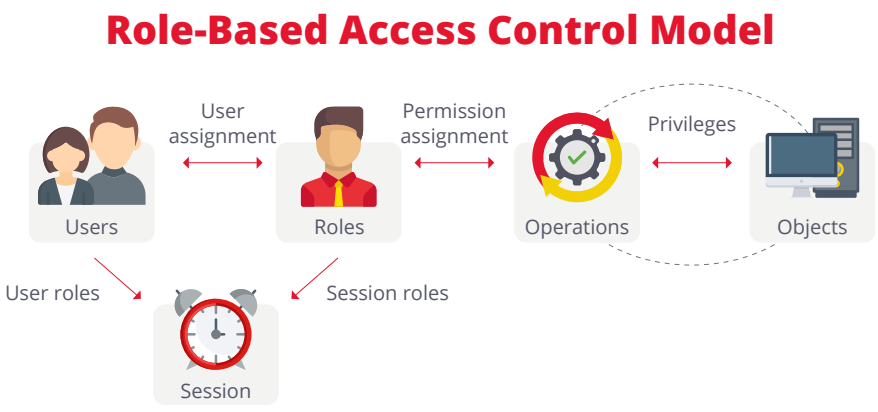


Figura 33. Detalle de un modelo de control de acceso RBAC

3.6.8. RS6.8 Fijar un único método de autenticación y gestión de identidades para toda la aplicación

Se trata de **centralizar el método de autenticación como requisito** (dónde va a estar dentro de la estructura de la aplicación) y especificar que toda petición de autenticación va a pasar por él. Se desaconseja que los usuarios se autentiquen por varios medios diferentes, salvo que queramos implementar una autenticación externalizada por varios proveedores, sí uno solo no es aconsejable.

Por ejemplo, si los usuarios no tienen cuenta en él y no queremos obligarlos a crearla, esto nos permitiría no perder usuarios potenciales.

No obstante, una vez autenticado el usuario y que se sepa su identidad (y rol), conviene **homogeneizar el tratamiento que se le hace en un único punto** para determinar si un rol tiene acceso a una operación o no. En otras palabras, una vez el usuario esté autenticado, debe ofrecer una identidad única, e independiente del método de autenticación, usado para el resto de la aplicación.

Este requisito también depende del framework de desarrollo usado, puesto que cada uno puede tener distintas formas de poner en marcha un RBAC. Si ya se tiene claro, es posible especificar más los requisitos para adaptarse a él, en lugar de dejarlo en un plano algo más abstracto.

A modo de ejemplo, con **ASP .NET** los roles que pueden acceder a una función de una aplicación se especifican de forma ortogonal a la misma con **atributos**, de manera que, una vez definidos los roles, solo hace falta etiquetar los métodos de la aplicación adecuadamente para que solo los roles autorizados puedan llamarlos sin necesidad de introducir código extra para ello en la lógica de la aplicación. El siguiente código fuente es un ejemplo de esta aproximación:

```
[Authorize(Roles = “HRManager,Finance”)]
public class SalaryController : Controller
{
    public IActionResult Payslip() =>
                    Content(“HRManager || Finance”);
}
```

Esta forma de trabajo sigue dejando la autorización en un solo punto (los atributos de cada método), pero con una aproximación más sencilla de revisar (*code review*) y **comprobar si se adapta al RBAC definido**. Ten en cuenta que, si has hecho una lista de roles que pueden hacer determinadas operaciones, traducirla a una implementación de esta clase es muy sencillo, simplificando el desarrollo.

Más información:

Understanding Role-Based Access Control with ASP.NET Web APIs

Implementación del control de acceso basado en roles

3.6.9. RS6.9 Forzar siempre el mínimo privilegio para todas las cuentas de usuario o servicio

El **principio del mínimo privilegio** debe forzarse en todas las partes de la aplicación, pero sobre todo en **accesos a BBDD o APIs**.

> ***Este principio consiste en solo dar a los roles de usuario los privilegios que sean estrictamente necesarios para cumplir su función, y ninguno más.***

Un ejemplo de cómo modelar este requisito lo tenemos en las operaciones que interactúen con un SGBD. Por ejemplo, debemos usar **dos cuentas de usuario vinculadas a dicho SGBD** (si no lo tenemos definido aún, se pueden especificar de manera abstracta).

1. **Un usuario del SGBD "lector":** Una cuenta de usuario con permisos sólo de lectura, para ejecutar sentencias `SELECT`. Si se quiere más detalle, se puede crear incluso distintos usuarios lectores con acceso a ciertas tablas, aunque eso complicará y alargará el desarrollo, y debe estudiarse bien si merece la pena invertir en ello en el contexto de esta guía.
2. **Otra cuenta de usuario "escritor"**, con permisos para implementar un CRUD de las entidades, pero **NUNCA *database owner* (DBO)** o equivalente (una cuenta de administrador, vamos).

La primera debe usarse siempre en todas las operaciones posibles, dejando la segunda solo para altas y modificaciones, que se deben poder identificar claramente a nivel de requisitos.

Además, se debe crear una **cuenta de servicio** **para cada API** que la aplicación use (por aislamiento), y cada una de esas cuentas debe también tener solamente los permisos estrictamente necesarios para su función. De esta forma no solo minimizamos el impacto de posibles compromisos a esas APIs, sino que podemos saber más fácilmente quién ha sido el culpable de haber ejecutado una operación que nos haya causado problemas. Si esto se establece en los requisitos, es mucho más fácil de implementar que si luego tiene que "retro-adaptarse" una vez parte de la aplicación esté construida (*pushing left*).

Es importante destacar que hay más cosas que se puede detallar a nivel de requisitos para prevenir ataques a la cadena de suministro siguiendo el ***OWASP Software Component Verification Standard*** **(SCVS)** **[48]**. No obstante, esto incrementaría el coste en recursos sustancialmente, lo que va en contra de los objetivos de esta guía.

> ***En esta primera fase inicial del producto software a construir, podemos empezar estableciendo como requisito el solo otorgar permisos de lectura, escritura o supervisión y aprobación de cambios en el repositorio de código a determinados desarrolladores, para en el futuro poder establecer más partes del SCVS.***

3.6.10. RS6.10 Habilitar copiar / pegar en los elementos de interfaz que usan *passwords*

Aunque este requisito parezca algo trivial, es muy importante, ya que **posibilita que los usuarios puedan usar sus *password managers* en nuestra aplicación** (como se especificó en un requisito anterior) y hace la aplicación más usable. No obstante, lo que si debe considerarse es **deshabilitar la función de autocompletar** del navegador en ellos, para evitar problemas en caso de que el usuario use una cuenta compartida por varios usuarios en su SO.

3.7. RS7 Requisitos de funcionalidades particulares

3.7.1. RS7.1 Hacer *threat modeling* de la aplicación

La actividad de ***threat modeling*** es imprescindible de cara al diseño de la aplicación para entender mejor los riesgos a los que se va a someter el *software* que se va a construir. En esta fase de requisitos, lo que se debe establecer claramente es que se va a hacer, y las condiciones y normas que regirán ese proceso. Posteriormente, ya hablaremos de cómo se pone en práctica.

Por ejemplo, se puede definir **quiénes** van a hacer el proceso, **cuándo** se va a hacer (una sesión, varias sesiones en hitos concretos del desarrollo...) y **cómo** (presencial, virtual, siguiendo una metodología, informal...), siempre desde la perspectiva de ahorro de costes que queremos llevar en esta guía.

Aunque se haga forma limitada, lo verdaderamente importante es que se haga, puesto que los beneficios del proceso son muy altos si se realiza correctamente y puede ahorrar muchos problemas (y sus costes derivados) en el futuro.

3.7.2. RS7.2 Evitar si es posible la funcionalidad de subida de archivos

La subida de archivos en una aplicación, aunque sea simplemente una foto de perfil, **introduce un conjunto de riesgos de seguridad importante**, para los que tendremos que invertir tiempo y recursos, siendo una de las necesidades que más va en contra de lo que queremos en esta guía.

En general, se recomienda **intentar evitar esta funcionalidad**, al menos en las primeras versiones del *software* de tu *start-up*, para poder sacar el MVP más rápido y con menos riesgos. Quizá puedas sustituirla por la selección de una foto de perfil entre una selección predeterminada de ellas.

Si debe haber una subida de archivos, entonces hay que seguir las **recomendaciones del OWASP [49]** para disminuir los riesgos de esta funcionalidad altamente problemática.

Se desaconseja completamente desarrollar esta funcionalidad nosotros mismos. De tener que hacerlo, usa un componente de 3ºs o del framework de desarrollo probado y verificado.

3.7.3. RS7.3 Hacer backups periódicos de todo el código y recursos de desarrollo

Todos los datos que estén implicados en el desarrollo de la aplicación, así como su código, deberían **guardarse en un *backup* periódico**, cuya periodicidad y estructura se puede establecer como requisito. Se debe añadir también otro semanal en una **localización geográfica distinta**, para cumplir con la **regla 3-2-1** que es la recomendada en estas situaciones (***Figura 34***).

Figura 34. Estrategia 3-2-1 de copias de seguridad

Un aspecto muy importante que se suele pasar por alto habitualmente es el proceso de restauración de dichos backups: debe probarse y practicarse también. Lo peor que nos puede pasar es creer que tenemos backups y, a la hora de usarlos para solventar cualquier incidencia, encontrarnos que no se han hecho correctamente o que están corruptos.

De cara al escenario de ahorro de costes que estamos planteando en esta guía, puedes basar tu estrategia de copias de seguridad usando estas directrices. Es importante intentar tener un sistema de copia de seguridad adecuado, aunque debamos limitar los recursos invertidos en él, ya que **una pérdida de datos puede acabar con nuestra *start-up***. Pero ya no solo porque los datos se destruyan sino porque, aunque no lo hagan, el retrasar los procesos y aumentar los costes puede hacer nuestro negocio inviable. Unos posibles requisitos para el sistema de *backup* pueden ser:

- **Hacer al menos 3 copias de tus datos** (1 principal y 2 copias de la principal): 2 de esas copias deberían estar almacenadas en **diferentes medios o plataformas** (por ejemplo, en un disco duro externo y en la nube).

Podrían ser más, pero aumenta la complejidad y los costes...

- **Asegurarse de que hay una copia almacenada "desconectada" o fuera de tu entorno de trabajo:** Eso quiere decir que en un momento dado **ninguno de los equipos de trabajo pueda acceder a dicha copia**, es decir, al menos tendrás una copia de seguridad completamente desconectada y aislada de tu red.

> ***El motivo es sencillo: los ransomware modernos buscan y cifran backups conectados para destruirlos, y que de esta forma no puedas restaurar tus datos. Si tienes que usar esa copia, tendrás que conectarla tú expresamente, y por tanto no es necesario que esté siempre accesible.***

- **Automatización:** Buscamos ahorrar costes y tiempo, por lo que los *backups* debería ser **periódicos y automatizados**. La frecuencia dependerá de cuánto cambia cada dato que copies (puede haber distintas políticas según el escenario) y cuánto tiempo pasa desde entre copias (que determinará cuantos datos puedes perder en caso de ataque). Para esto puedes **elegir un *software* de copia de seguridad** que pueda hacer el proceso transparente.
- **Verificación:** Como dijimos antes, si no compruebas que tus copias funcionan correctamente, **no estás haciendo una estrategia de copia correcta**. Un buen *software* de copias debería permitirte hacerlo fácilmente.

> ***Puedes aprovechar a incluir este proceso en un plan de respuesta a incidentes inicial para tu empresa: un procedimiento a seguir si pierdes el acceso a tus datos para intentar restaurarlos y recuperar la actividad lo antes posible. El tiempo que se tarda en recuperar la actividad ante un incidente es muy importante en el contexto de una start-up.***

- **Haz simulacros:** Haz una restauración de prueba sobre alguna de tus máquinas de vez en cuando (puedes establecer el periodo como requisito), para comprobar que el sistema de copia funciona si es necesario. De nuevo, un buen *software* de copias debería permitirte hacerlo fácilmente.
- **Protege el acceso:** Las copias de seguridad deberían estar protegidas con contraseñas fuertes y 2FA si es posible, para que **solo las personas y procesos autorizados puedan leerlas o modificarlas**. Entre las técnicas de protección puedes considerar, si estimas que es un riesgo que te roben los datos, que las copias se hagan cifradas. Un buen *software* de copias también debería permitirte hacerlo fácilmente.
- **Se debe actualizar el *software* de *backup*** para asegurarse de que las copias se hacen sin fallos y que el *ransomware* (u otro *malware*) no sacan partido de sus vulnerabilidades para eliminar o corromper nuestras copias.
- **Monitorización:** Si puedes permitírtelo, conviene que hagas una vigilancia especial sobre el uso de la red en el lugar donde almacenas las copias, por si un aumento repentino de ese tráfico indicase que tienes un *ransomware* en marcha. Una forma barata de hacerlo es recurrir al panel de administración de tu *router*, ya que muchos tienen esta funcionalidad integrada hoy día. Al final de la guía veremos más opciones de monitorización.

3.8. RS8 Requisitos de log o de registro de acciones que han ocurrido en la aplicación

3.8.1. RS8.1 Registrar todos los intentos de acceso

Se debe establecer como requisito que quede un **registro de quién accedió a cada sistema en un día y hora concretos** (sean o no intentos fallidos). **Jamás incluir en ese registro información sensible** (claves, nºs de tarjeta ni, en general, nada que se considere información privada de una persona).

Esto permite tener una trazabilidad de quien hizo cada cosa en la aplicación, de manera que se pueda facilitar encontrar al responsable de un incidente, o bien hacer un estudio del uso de la aplicación que luego se pueda emplear en otras cosas.

3.8.2. RS8.2 Si se detectan accesos indebidos, lanzar alertas de seguridad

Si se detecta alguna actividad sospechosa, la aplicación no solo debe registrarla, sino también **mandar una alerta como requisito** (para ahorrar recursos, por email, pero a uno del equipo responsable de seguridad, no a una persona concreta...). Esto puede hacerse definiendo qué eventos que resultan obviamente anómalos en el contexto de la aplicación, permitiéndonos estar al tanto de posibles **ataques típicos**. Por ejemplo:

- **Más de X intentos de *login* fallidos** en un tiempo T predeterminado.
- **Acceso a ciertas partes de la aplicación** que requieren permisos elevados, pero por parte de un usuario cuyo rol sea más bajo y no lo permita.
- Lo mismo con **documentos** o informes generados.

Se puede establecer como requisito también crear un catálogo de alertas que se van a notificar con sus causas. No obstante, probablemente esto sea más fácil hacerlo tras el proceso de threat modeling del que hablaremos en la fase de diseño.

4. Elementos de diseño seguro

4.1. Modelado general de las operaciones de la aplicación

Todos los requisitos de seguridad vistos deben ser puestos en práctica en el diseño, validando expresamente que se han incluido todos los que son aplicables al *software* diseñado. Esto evita en lo posible los fallos de diseño, que no deben confundirse con errores de seguridad.

- Un **fallo de diseño** permite a los usuarios hacer cosas que no deberían.
- Un **error** es un fallo en el código de la implementación de una característica.

> ***En otras palabras, a la hora de terminar el diseño debe hacerse una lista de comprobación que indique dónde se ha incluido cada requisito, para estar 100% de que no falta ninguno.***

Recuerda la aproximación *pushing left* que hemos mencionado al principio de esta guía: cuanto 1º se descubra un fallo de diseño relativo a seguridad, más barato será arreglarlo. Esto se puede mejorar haciendo ***Threat modeling*** que veremos en este apartado.

4.1.1. Diseño de la protección de datos personales

Para incorporar el requisito de protección de datos sensibles visto, debemos **mirar si ya hay políticas de empresa de cómo tratar datos sensibles para seguirlas y cumplirlas**. Por ejemplo, si no es el primer producto que la *start-up* diseña con esa necesidad, o bien si una empresa se asocia con nosotros para ayudarnos a arrancar el proyecto. Si no existen, es una buena idea crearlas o establecerlas en este punto.

Esto es un aspecto que puede resultar complejo si no estás familiarizado con el tema, pero por suerte el **INCIBE** tiene una serie de documentos para ayudarte con este aspecto. Se trata de la página de **políticas de seguridad para la pyme [50]**, que es un conjunto de políticas que puedes usar para **diseñar las distintas partes que forman parte de tu *start-up* y del *software*** que debes diseñar, en formato editable para que puedas adaptarlas a tus necesidades.

Hay una gran cantidad de documentos distribuidos por perfiles: para empresarios, para personal técnico y para empleados y, si los examinas en detalle, verás que cubren gran cantidad de los requisitos que hemos mencionado en la sección anterior, por lo que son ideales para darles forma en la fase de diseño. Te recomendamos encarecidamente que los analices porque te pueden ayudar mucho en "dar el paso" del análisis al diseño. Tampoco olvides que en tu caso estás constituyendo una start-up alrededor de un producto software, por lo que las políticas que apliques deben cubrir ambas cosas.

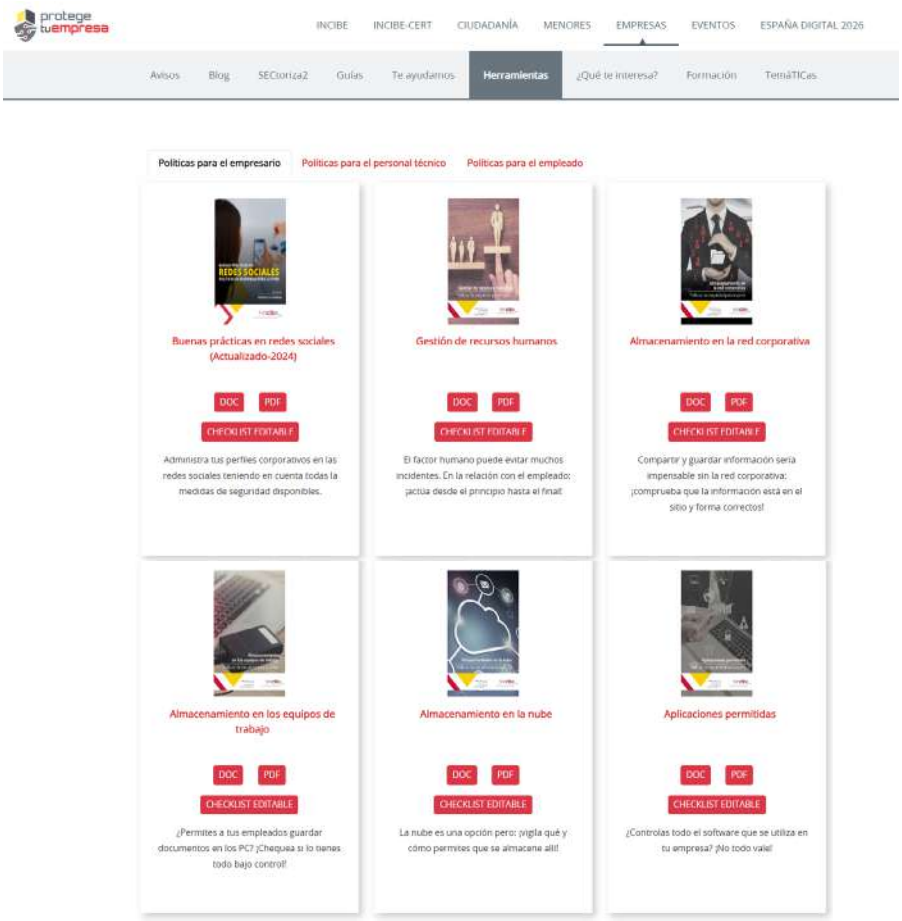


Figura 35. Gracias a toda esta documentación, el INCIBE te facilitar mucho crear tus políticas de seguridad

Si se sigue el consejo de usar una guía existente para ello, hay que prestar atención a cuánto tiempo se guardan datos sensibles y borrarlos automáticamente una vez transcurrido. Si se van a poner datos en la nube (en la ***Figura 35*** se puede ver una política específica para esos casos, al igual que para muchos otros), hay ser especialmente cuidadoso con cuáles, según la legislación vigente.

Salvo que en el equipo de la start-up haya una persona experta en temas legales, nuevamente se aconseja pedir una asesoría legal sobre esto a profesionales especializados, para cumplir con el RGPD y evitar problemas futuros.

4.1.2. Protección de filtraciones

Otra decisión que se debe tomar en la etapa de diseño, cuando ya sepamos a ciencia cierta los datos que vamos a tratar y su confidencialidad, es intentar **contratar servicios profesionales** por si hay filtraciones de nuestros datos sensibles online **[51]**.

Si esto tiene un coste inasumible para tu *start-up*, se puede empezar a implementar algo en esa vía con opciones más económicas. Por ejemplo, si parte de nuestros datos son direcciones de correo de nuestros usuarios (o de nuestros empleados), una suscripción a un servicio como ***Have I Been Pwnd?*** **[52]** o similar nos puede alertar si dichos correos se han filtrado.

Por ejemplo, podemos adquirir inicialmente una clave para la API del servicio que nos permitirá buscar hasta 10 cuentas de email filtradas por minuto y dominios hasta con 25 cuentas de email filtradas, como se indica en la propia web (***Figura 36***).

Figura 36. Tarifas de acceso a la API de Have I Been Pwnd?

Con el acceso a esa funcionalidad, podemos programar un escaneo periódico usándola cada X días, y enviar una **alerta** en caso de encontrar alguna filtración en los emails de nuestros desarrolladores y empleados.

> ***¿Te das cuenta de que si haces esto también puedes prevenir filtraciones a nivel personal de tus empleados y todas las complicaciones (probablemente impredecibles) que ello podría traer a corto o largo plazo? Recuerda que al final tus empleados son un eslabón más de tu cadena de seguridad...***

En las primeras fases de la constitución de nuestra *start-up* tiene sentido no contratar un servicio más exhaustivo y complejo, pero sí estar atento a posibles filtraciones que pudieran acabar con el proyecto en una fase temprana. Si el negocio crece y disponemos de más recursos, será el momento de estudiar **cómo invertir en una estrategia completa de prevención de pérdida de información [53]**.

4.1.3. Filosofía de diseño general

Hay tres filosofías que deben imperar durante toda la fase de diseño para cumplir con los requisitos anteriores:

- ***Never Trust, Always Verify / Zero trust / Assume breach:*** Consiste en usar solo datos previamente validados en el servidor para tomar decisiones. Esto implica **asumir que cualquier dato podría estar comprometido** al diseñar las funcionalidades de la aplicación.

> *Esto sería una buena forma de traducir el requisito RS1.1 a la aplicación.*

- **Denegar por defecto:** Diseñar todas las funciones para que comprueben siempre antes si el llamador está autorizado a usarlas, y verificar siempre la **identidad** (***AuthN***) antes de **comprobar si se tiene permiso** para algo (***AuthZ***). Se pueden encontrar elementos específicos sobre como diseñar en detalle un proceso de autenticación correcto en **[54]**.

> *Se debe volver a verificar el acceso en cada página o característica de la aplicación, incluso aunque solo se recargue una página web.*

- **Transacciones:** Recuerda que se deben cerrar las transacciones correctamente en caso de fallo (requisito RS5.8) y nunca caer en estados desconocidos, definiendo en fase de diseño **qué pasa si una transacción no se puede ejecutar completamente** para cada una de las transacciones previstas.

4.1.4. Diseño de APIs

Si diseñamos un API como parte del proceso de creación del software de la *start-up*, también hay que verificar primero la identidad (***AuthN***) y si se tiene acceso a una característica (***AuthZ***) o llamada de la misma después.

> *Esto debe hacerse en ambos sentidos (llamadas de una API a la aplicación y de la aplicación a la API).*

4.1.5. Aislamiento por diseño

Finalmente, en el diseño se deben incorporar **medidas de aislamiento**:

- **Bloquear el acceso** a cualquier protocolo, puerto, verbo o método HTTP que la aplicación no use dentro del propio código de la aplicación o su configuración si es posible.

> ***Este tipo de cosas se pueden hacer a nivel de configuración, pero si lo podemos establecer nosotros en el software también tendremos una capa más de seguridad por si acaso (defensa en profundidad).***

- **Diseñar el sistema para intentar desplegar solo una aplicación por servidor**, PaaS o contenedor en la medida de lo posible, de forma que una caída en uno de ellos afecte lo mínimo posible al resto.
- **Intentar que las funcionalidades tengan un bajo acoplamiento y una alta cohesión:** Las funciones relativas a una determinada entidad o funcionalidad deben estar en un mismo sitio, separadas de todas las demás. Esto no solo es bueno para mejorar la calidad del código y de la arquitectura del programa. Detallaremos más este punto en la sección siguiente.

4.2. Estructura segura del código de la aplicación

Para cumplir con los requisitos vistos, hay que hacer una separación de código por diseño: **separar el código que lidia con aspectos de seguridad del código funcional** (clases separadas, otra aplicación, usar un sistema de autenticación de un proveedor externo (Ej.: *Google*), etc.). Esto también incluye lo que se comentó en los requisitos acerca de centralizar toda la gestión de identidades en un único punto.

4.2.1. Sobre la posible arquitectura de la aplicación

Si ya se sabe la arquitectura donde se va a desplegar la aplicación, puede pensarse en una **separación física del código**, introduciendo partes de la arquitectura de un programa en un servidor o contenedor separado del resto como dijimos antes, pero protegido por un ***firewall*** que restrinja el acceso solo a las aplicaciones autorizadas.

También pueden separarse las clases que se encargarán de la recolección de *logs* y aspectos de alerta y monitorización, que harán más fácil la interacción con IDS/IPS, de existir alguno. Otra separación por diseño se puede hacer por tipos de funcionalidad: **Separar el código de los administradores de la funcionalidad "estándar" de la aplicación**.

El punto anterior es más importante de lo que parece, puesto que si se conoce la arquitectura que se va a usar para desplegar las distintas partes de la aplicación se puede diseñar desde el principio para que los distintos elementos de la aplicación se adapten a la misma, **ahorrando tiempo en el despliegue**. Por ejemplo, una arquitectura como la de la ***Figura 37*** separa la aplicación completa en **cinco redes aisladas**:

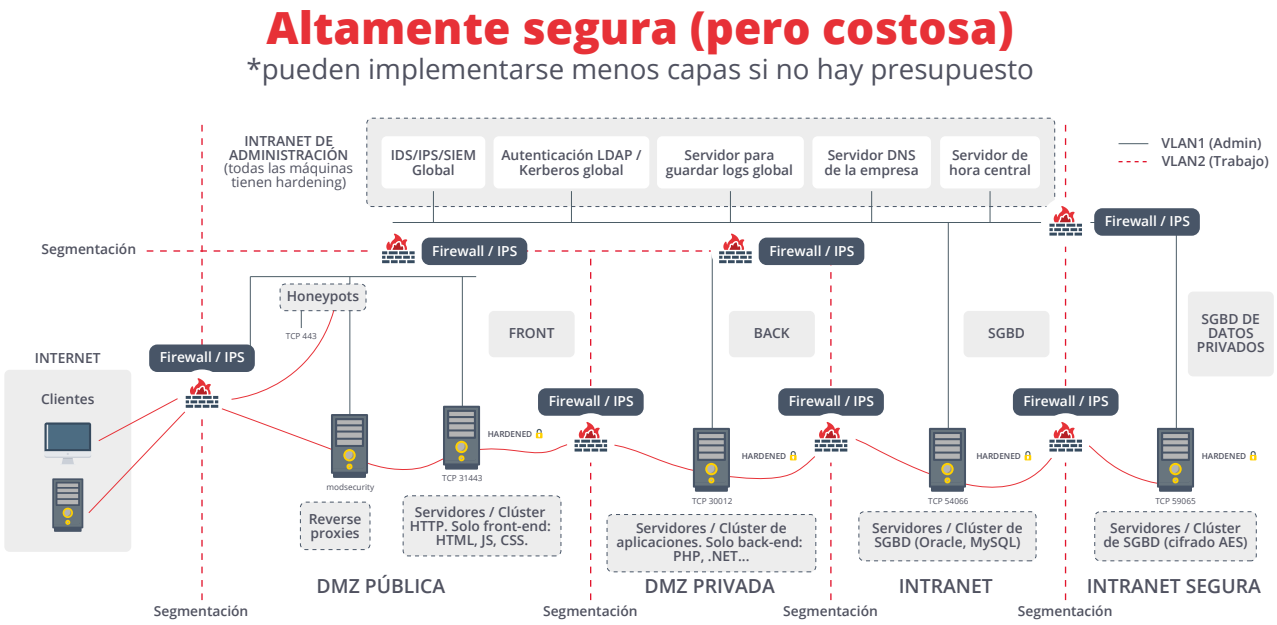


Figura 37. Arquitectura de una aplicación altamente segura

- **Acceso a la infraestructura:** Controlado por un *firewall* que puede filtrar tráfico malicioso. Puede complementarse con el uso de un servicio como *CloudFlare*, como veremos luego.
- **DMZ Pública:** Solo estará el código del *front-end* de la aplicación (HTML + *JavaScript* + CSS).
- **DMZ Privada:** Con el código del *backend*, que se invocará desde el *front-end*.

> ***Entre ambas redes habrá un firewall que regirá de manera estricta qué máquinas concretas del front pueden invocar servicios concretos sobre ciertas máquinas del back, para evitar que máquinas de una red accedan a las de otra (o a alguno de sus servicios) de manera imprevista.***

- **Intranet:** Con la BBDD de la aplicación, separada de la DMZ privada con un *firewall* del mismo tipo que el descrito antes.
- **Intranet segura:** Una cuarta red donde residirán los datos más privados de la aplicación, en una **BBDD muy monitorizada y con el acceso desde máquinas o aplicaciones particulares muy restringido y controlado**, estando todos los datos cifrados con un algoritmo fuerte.

En nuestro contexto invertir en algo así solo tiene sentido si guardamos datos muy sensibles (nºs de tarjetas de crédito, historiales médicos…) debido al coste que podría tener.

- **Administración / monitorización:** Una red independiente encargada de recoger los *logs* de monitorización de las máquinas de las otras redes, examinando la información, **detectando y previniendo intrusiones** y mostrando alertas a las personas a cargo de la vigilancia de las máquinas. Un producto que puede hacer esto gratuitamente se describirá aquí.

La monitorización de nuestra infraestructura también es algo que puede tener un coste elevado e ir en contra de los objetivos de esta guía. No obstante, al final de esta guía te daremos ideas para intentar que puedas tener monitorización con un producto profesional gratuito.

Si de antemano sabemos que la arquitectura está estructurada así, sabremos que el *front-end* va a hacer llamadas a una determinada dirección para acceder a su *backend* correspondiente, y el código de la aplicación se debe **diseñar de una forma completamente desacoplada desde el principio**. También sabremos que cada parte de la aplicación (*front*, *back*, BBDD…) estará aislada, y podemos diseñar la estructura de paquetes, clases, etc. para adaptarse a ese despliegue físico. En cualquier caso, es una información que podremos aprovechar para mejorar el diseño.

No obstante, la arquitectura completa presentada en la ***Figura 37*** **tiene un coste elevado**, y no resultaría adecuada para el contexto de una *start-up*, especialmente al principio de su andadura. Si bien dota a todo el despliegue de una alta seguridad, mantener toda la infraestructura que permite ese aislamiento y monitorización (*firewalls*, VLANs, sistemas de detección de intrusos…) requiere mucho tiempo y recursos económicos.

Por ello, se puede pensar en algo más modesto para empezar basándose en estos principios (Ej.: **3 capas *front-back-BBDD* o 2 capas *front-back+BBDD***) con un sistema de monitorización más sencillo subcontratado o de un alcance más local (probablemente no separado en su propia subred). Si el diseño se plantea bien, y con su posible extensión en mente, a medida que la empresa vaya ganando recursos puede evolucionarse la infraestructura a un modelo más seguro más fácilmente.

La planificación aquí es muy importante: Si tienes un objetivo final y despliegas inicialmente algo de coste reducido con él en mente, es mucho más fácil llegar a él cuándo tengas los recursos necesarios.

4.2.2. Prevención de posibles ataques en tiempo de diseño

En lo relativo al diseño de la **gestión de secretos de la aplicación**, una vez tenemos claro el requisito RS4.2 de usar una ***secret store***, en esta fase **debemos definir dónde guardamos TODOS los secretos de la aplicación** (cadenas de conexión a la BBDD, certificados, *API Keys*...). Esto implica definir la ***secret store***, incorporarla a la cadena CI/CD o definir cómo acceder por programa y dónde se guardará la misma en la arquitectura.

En la fase de diseño también se pueden incorporar **medidas para combatir ataques de CSRF [55]**. Concretamente, consiste en identificar los puntos en los que el usuario hace operaciones sensibles (compras, cambios de clave...) para pedirle su clave, un captcha **[56]** o un *token* que solo él conozca. También se pueden combinar con las medidas contra este tipo de ataques que pueda tener el *framework* de desarrollo que se haya elegido. Recuerda siempre que, si el *framework* lo soluciona de alguna forma, es la solución que debes priorizar aplicar.

Finalmente, y también relativo a la separación, se debe diseñar **cómo proteger el código fuente de la aplicación a toda costa** (salvo que sea *open-source*). Esto implica definir las políticas y permisos de acceso a los repositorios, ofuscación del código pasado a producción (Ej.: *JavaScript* **[57]**), *backup*, monitorización y *log*...

Esto, que ya se comentó en el requisito RS6.9, es un aspecto muy importante: los ***supply chain attacks*** son tan serios y frecuentes que se necesita una estrategia concreta para proteger tus repositorios de código. También puede definirse en que parte de la infraestructura se guardarán dichos repositorios.

Para hacerlo correctamente se debería seguir el ya mencionado ***OWASP Software Component Verification Standard* [48]**. No obstante, poner en marcha todo lo que tiene este estándar entra en conflicto con la reducción de tiempo de desarrollo y costes que exige una *start-up*. Por ello, como primera aproximación se puede empezar definiendo **quién tiene acceso** al código de entre todos los empleados y **para qué** (lectura, escritura, revisión), para pasar más adelante a un modelo más complejo a medida que la *start-up* vaya creciendo.

> ***En definitiva, se trata de ofrecer más seguridad que el hecho de que todo el mundo tenga acceso pleno para hacer cualquier operación en el repositorio, o para sobrescribir el trabajo de otros miembros de la start-up. Aparte de los posibles errores, un compromiso en cualquier cuenta de usuario puede suponer la pérdida (o compromiso) total del software y con ello de nuestro negocio.***

4.3. La importancia del *threat modeling*

Como ya avanzamos en el requisito RS7.1, este es un proceso que ayuda mucho en fase de diseño a la hora de que la aplicación tenga menos errores de seguridad.

> ***Consiste en la identificación de amenazas potenciales (para el negocio de la start-up, el sistema... pero, sobre todo en esta fase, para producto o aplicación que estamos construyendo).***

Es un ***brainstorming*** donde se piensan, buscan y definen los posibles peligros a los que nos podemos enfrentar, por ejemplo:

- **Intercepción / filtrado de datos**, y su posterior venta, y el impacto que puede tener.
- **Cómo protegerse ante posibles ataques** que identifiquemos en este proceso.
- Cualquier peligro, cualquier problema que imagines.

Se trata por tanto de identificar riesgos, de manera que al identificar alguno debemos:

- **Cambiar el diseño** de la aplicación para contrarrestarlo.
- Que un responsable del equipo de ello **acepte el riesgo** documentadamente.

Las etapas generales y una descripción más en detalle de lo que es un *threat modeling* podemos verlas en la ***Figura 38***.

Figura 38. Fases típicas de un proceso de threat modeling

Este es un proceso que requiere práctica y detalle, y existen **metodologías que facilitan su implementación:**

- **Attack trees:** https://www.mytechiebits.com/AttackTrees
- **STRIDE:** https://www.koenig-solutions.com/blog/stride-methodology-in-threat-modelling
- **PASTA:** https://owasp.org/www-pdf-archive/AppSecEU2012_PASTA.pdf

Para ello, lo mejor es **reunir a un representante de cada uno de los *stakeholders* de la aplicación** (es decir, cualquiera que tenga un interés en la aplicación a desarrollar): El potencial **cliente**, un **especialista en seguridad ofensiva**, un **arquitecto del sistema**, responsables de la **plataforma de despliegue**..., etc., siempre que estén disponibles y tenga sentido en el contexto de desarrollo que cada aplicación tenga.

Una vez fijemos quien formará parte de las reuniones, deben planificarse las mismas. Si se sigue un proceso de diseño iterativo, estas reuniones es mejor tenerlas en cada iteración, y procurar que sean cortas. Cada grupo debe aportar los riesgos que ve en el sistema según su visión, haciéndose preguntas como estas:

- *"Si fueras a atacar el sistema, ¿Cómo lo harías?"*
- *"¿Quiénes crees que atacarían el sistema?"*
- *"¿Cómo te prepararías para un ataque?"*
- *"¿Qué podría ir mal?"*
- *"¿Cómo protegemos al usuario?"*
- *"¿Qué es lo peor que puede pasar?"*

Los que hagan este proceso **deben verlo desde la mentalidad "del mal"**, como si fueran adversarios. Al final se trata de convertir las ***"user stories"*** (casos de uso) en ***"abuse stories"*** o ***"abuse cases"*** *("casos de uso negativos")*. Ej.: Si una aplicación se supone que hace X, *¿qué pasa si hace Y?*

Hay bastante literatura acerca de cómo diseñarlos a partir de los casos de uso de un proceso de análisis y diseño estándar, y varias formas de representarlos, pero en un contexto como el nuestro nos vamos a centrar en que se desarrollen y usar la misma representación que ya tengamos para los casos de uso típicos de la aplicación para ahorrar tiempo.

Ejemplos de representación los podemos ver en la ***Figura 39*** y en la ***Figura 40***. Los artículos de investigación asociados pueden ayudarte también a saber cómo representarlos adecuadamente en tu caso particular.

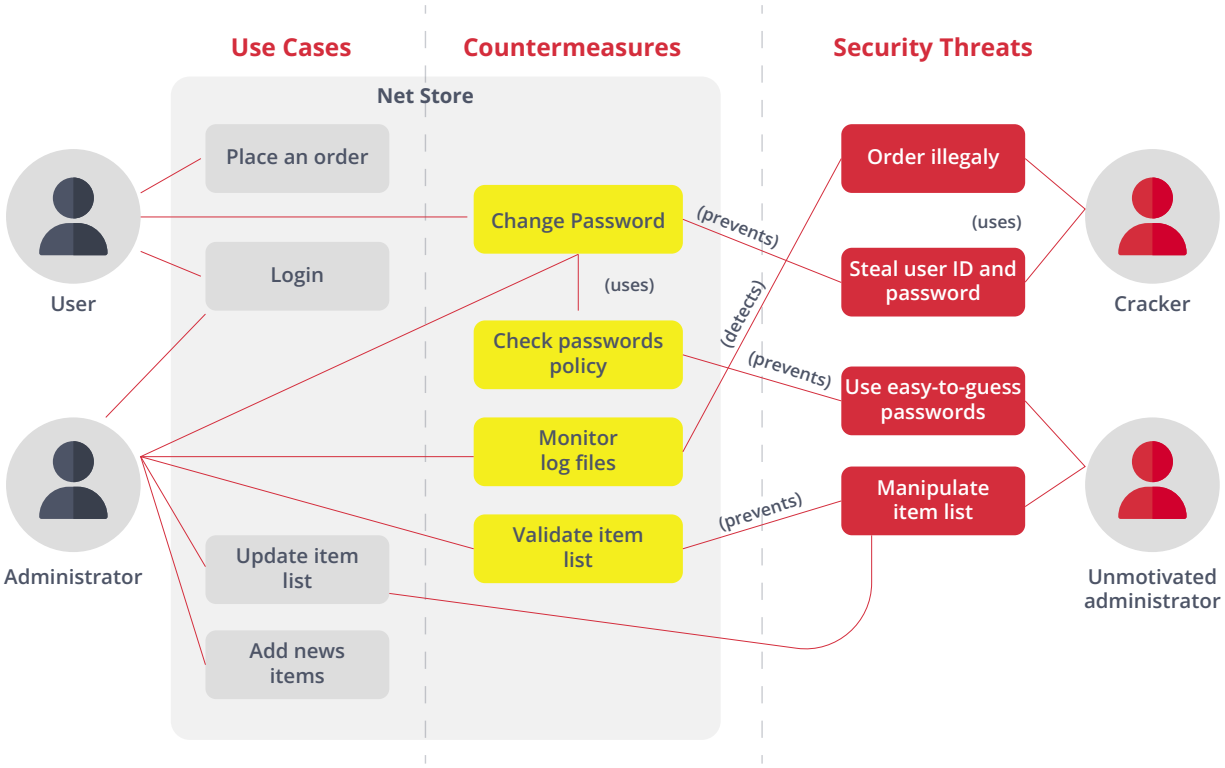


Figura 39. Ejemplo de caso de uso negativo de una aplicación, especificando sus contramedidas

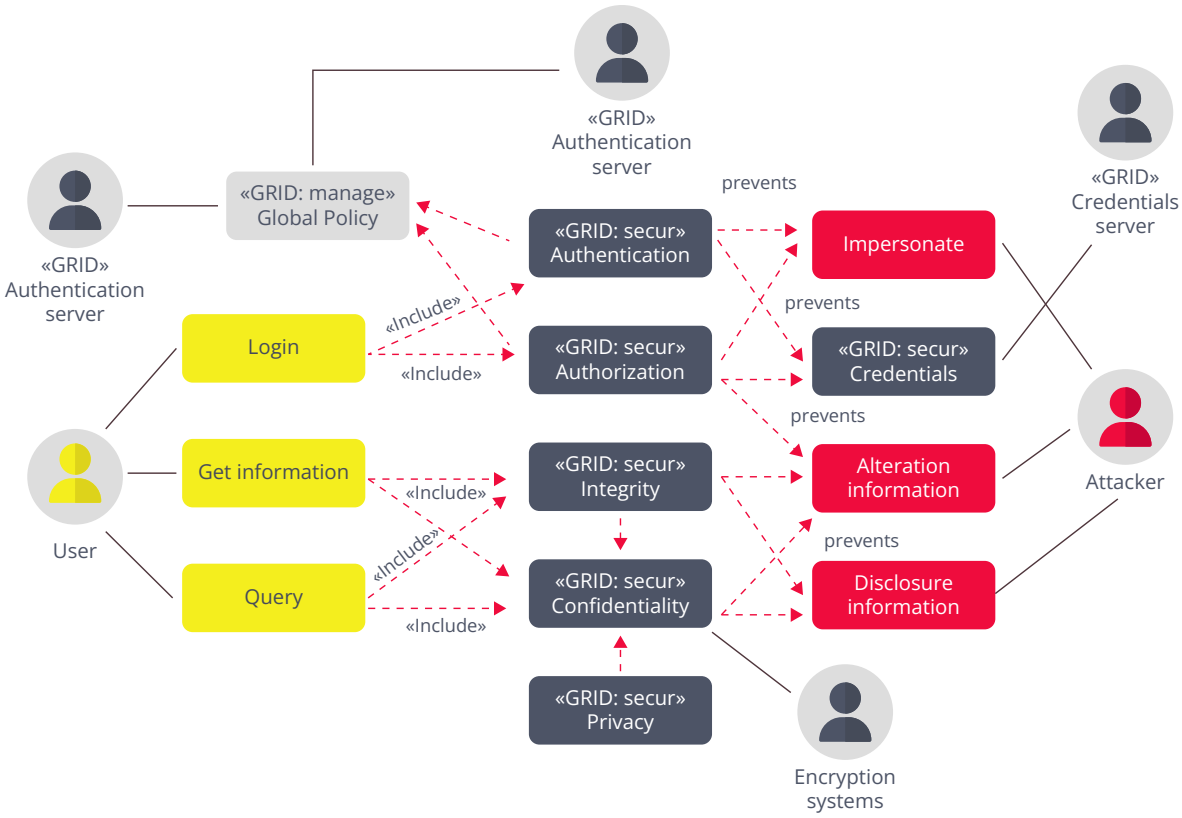


Figura 40. Otro ejemplo de caso de uso negativo de una aplicación, especificando sus contramedidas

El principal problema con el que nos encontramos con el threat modeling es que, en el contexto de una start-up de esta guía, probablemente se deba hacer de forma más modesta por la limitación de recursos al principio del proceso, aunque siempre será mejor que no hacerlo...

Usemos o no metodologías de ***threat modeling***, el objetivo es que el equipo pueda hacerse las preguntas necesarias para cubrir en esta fase todo lo que puede ir mal y que, como vemos en las imágenes anteriores, esos hechos y sus posibles contramedidas puedan aparecer representados en los documentos de diseño.

El ***threat modeling*** permite pensar en cosas como qué tiene nuestra **aplicación de valor** (dinero, datos...) y cómo alguien podría robarlo, manipularlo (reputación, DoS...), romperlo o usarlo indebidamente, y **elaborar respuestas** en una fase temprana del desarrollo (recuerda la filosofía *pushing left* de la que hablamos).

¿Qué son estas respuestas? una **"lista de problemas"** potenciales de la aplicación detectados. Hay que evaluar cuáles son **riesgos** y darles una **puntuación**, para elegir si **mitigarlos** (cambiar los requisitos, el diseño, el código...) o **aceptarlos**. Esto solo debe hacerlo las persona o personas que actúen como dirección del proyecto y todo este proceso debe quedar **documentado**.

Ten en cuenta que, aunque se haga de manera informal o abreviada por falta de tiempo y recursos, solo el hecho de planteárselos ya nos puede hacer ganar mucho en aspectos de seguridad en la aplicación a diseñar.

Además, tener una base de casos de uso negativos ya desarrollada e integrada puede ayudarnos mucho si el negocio de la *start-up* crece y con él la aplicación, apareciendo nuevos requisitos y funcionalidades sobre los que tengamos que desarrollar el mismo proceso, y que puedan tener unos casos de uso negativos similares a los que ya tenemos.

Más información:

Threat Modeling

Abuse Case Cheat Sheet

5. Aspectos clave para la creación de código seguro

En esta sección vamos a dar una serie de prácticas básicas para hacer que el **código del *software* de tu *start-up* sea lo más seguro posible**. Esto requiere tener un nivel técnico para poder entender todos los conceptos involucrados en la misma. Para ayudarte con esta fase, en caso de que el *software* sea una web, el **INCIBE** material adicional de ayuda que te puede ayudar si no estás familiarizado con estos procesos demasiado. Por ejemplo, está el ***Programa: Programación segura de sitios web*** **[58]** que puede servirte de gran ayuda para afianzar conceptos de desarrollo web si al leer esta sección ves que no tienes alguno claro.

5.1. Selección de los *framework* a usar o su versión concreta

Si tenemos la opción de elegir todos o parte de los *frameworks* / librerías / código de terceros (Ej.: no hay que integrar el producto en un ecosistema existente), hay que seguir unas reglas a la hora de determinar cuáles usar, implementando el requisito RS5.4:

- **Usar productos con buena reputación que tengan un respaldo** de una empresa o una comunidad detrás sólido y que se mantenga a lo largo del tiempo.
- Mirar **cuánto tiempo va a estar soportada la versión elegida** (cuanto más, mejor, como las versiones LTS (*Long-Term Support*)).

> ***Nunca debe usarse algo cerca del fin de su ciclo de mantenimiento. En general, suele ser adecuado usar la última versión de soporte "largo" disponible, aunque no disponga de las últimas características.***

- Estudiar el **historial de vulnerabilidades en un repositorio de CVE**, como vimos anteriormente en esta guía.
- **No usar "el ultimo *framework* o lenguaje de moda"** hasta que no esté establecido y se sepa que tendrá un soporte constante y duradero. También incluye versiones alfa, beta o en desarrollo, **por muy novedosos y prometedores que sean**. Haríamos prácticamente de *beta testers* del *framework*, nos encontraríamos con errores de implementación imprevistos y todo esto, al final, causa retrasos en todo el desarrollo significativos que como *start-ups* no nos podemos permitir.

Una start-up debe crear un producto innovador en un mercado donde de respuesta a una necesidad que le permita crecer para vender su producto, pero eso no implica usar frameworks innovadores demasiado nuevos, que estén en un estado incompleto y/o no listo para crear aplicaciones estables aún.

- Hay que usar **TODAS** las características de seguridad del *framework* elegido **SIEMPRE**, para cumplir con el requisito RS5.3.

Una start-up debe crear un producto innovador en un mercado donde de respuesta a una necesidad que le permita crecer para vender su producto, pero eso no implica usar frameworks innovadores demasiado nuevos, que estén en un estado incompleto y/o no listo para crear aplicaciones estables aún.

Además de todo esto, se debe documentar cómo se ha tratado de minimizar la cantidad de librerías, componentes, *frameworks*, etc. usados (superficie de ataque) y establecer cuándo escanear la versión usada de cada una de ellas con una herramienta **SCA** adecuada para ver si tiene vulnerabilidades si es posible (veremos más sobre esto después).

5.2. Gestión de PII y secretos de la aplicación

Para implementar el requisito RS4.3 y ser coherentes con el diseño, recordamos una vez más (dada su enorme importancia) que el **código fuente de una aplicación no debe tener:**

- **PII** (*Personal Identifiable Information*): Nºs de tarjeta, emails, IPs, credenciales, nºs de teléfono, direcciones, URLs, IDs de seguimiento...

No declares una variable con un valor de este tipo, ni aunque sea para pruebas.

- **Secretos:** *Passwords*, claves privadas, URLs "*hardcodeadas*" a secciones teóricamente no accesibles, *hashes* de datos o claves, datos codificados en *Base64*, sentencias SQLs, IPs de máquinas no accesibles al público...

Recuerda que la aplicación incorpora una secret store como requisito para guardar toda esta información.

Toda esta información ello puede ser localizada y usada en nuestra contra, con el riesgo de **dañar nuestro negocio en una fase muy temprana**, lo que podría acabar con él: escaneo contra objetivos poco protegidos, OSINT...

Volvemos a mencionar aquí este aspecto porque no podemos descartar que aparezcan datos sensibles por error o porque parte del equipo de desarrollo no tenga (o no aplique) la concienciación contra estas prácticas adecuada. Por ese motivo, es necesario **escanear automáticamente el código** buscando este tipo de información antes de ir a producción con **herramientas específicas**: *The Silver Searcher*, *Dumpster Diver*, *CRASS*... o cualquier funcionalidad similar que nos de nuestro entorno de desarrollo.

GitHub por ejemplo tiene una acción para ello.

Si para este proceso se necesitan expresiones regulares para hacer las búsquedas, recuerda no desarrollarlas tú mismo y usar fuentes verificadas para las mismas (RS1.1):

OWASP Validation Regex Repository

Regular Expression Library

Relativity One

5.3. Algoritmo general para implementar la validación de datos

Aunque ya hemos hablado de las condiciones en las que debe hacerse la validación (RS1.2), no hablamos de cómo implementarlas.

Recuerda: siempre validar del lado del servidor, hagamos o no validaciones en el cliente con JavaScript, antes de procesar cualquier entrada del usuario.

Puedes seguir el esquema general de la ***Figura 41*** como guía de implementación.

Figura 41. Algoritmo general de validación de datos

A la hora de implementarla, de acuerdo con lo dicho en RS1.1, hay que tener en cuenta los siguientes **criterios**:

- **La entrada de los usuarios / archivos / APIs externas** debe tratarse con precaución, al ser uno de los vectores de ataque principales. Los datos que provienen de una entidad externa nunca deben ser de confianza (*"All input is evil"* **[59]**).
- **Las entradas deben validarse con APIs centralizadas**, probadas y de acuerdo con su contexto (rangos, valores coherentes...). En cualquier caso, la implementación de la validación de las entradas debe:
  - Aplicarse a **todos** los datos de entrada.
  - Definir un conjunto permitido de **caracteres aceptados**.
  - Definir una **longitud mínima y máxima**: Incluidos dígitos decimales y no decimales, el tamaño máximo de los nombres...
  - Definir un **tipo o rango esperado** si es posible.
- La validación de datos debe ser además **sintáctica y semántica**:
  - La **sintáctica** obliga a los datos a tener una sintaxis y estructura correctas: Fechas, moneda, DNIs, teléfonos...
  - La **semántica** obliga a los valores a ser correctos en un contexto o de acuerdo con las reglas de negocio dadas.

Por ejemplo, la fecha de inicio debe ser anterior a la fecha de finalización, el precio de un artículo está dentro de un rango admitido, el tipo de vía residencial está dentro de los tipos admitidos (calle / plaza, etc.) ...

Si estas cosas se comprueban lo antes posible, las entradas no autorizadas se pueden detectar rápidamente y actuar en consecuencia. Por otro lado, recuerda que, si es un caso claro de manipulación maliciosa, se puede optar por **implementar respuestas contra el usuario** (no solo descartar los datos) **en el *software*** (suspensión, bloqueo...) y lanzar alertas de seguridad (RS8.2).

5.3.1. Listas de elementos permitidos vs Listas de elementos bloqueados

Como se estableció en RS1.1, siempre se ha de intentar **favorecer el uso de listas de elementos permitidos** (donde el resto de los elementos están bloqueados) sobre listas de elementos bloqueados (donde el resto de los elementos se permiten). No obstante, para ciertas aplicaciones el uso de listas de bloqueo es adecuado. Esto ocurre cuando el conjunto de elementos a bloquear es **finito, conocido en su totalidad, y no muy grande**. Por ejemplo:

- **Bloqueo de clientes / redes específicas conocidos** en entornos con un nº restringido/ conocido de ellos (LAN).
- **Bloqueos regionales (por país/zona):** países conocidos por ser fuente de un gran número de ataques y que se quieran descartar como origen de posibles clientes.
- **Conexiones de clientes desde la red ToR:** El número de nodos de salida en la red ToR en un momento determinado es limitado y conocido y se pueden bloquear individualmente.
- **Bloqueo de dominios conocidos como maliciosos**.
- **Bloqueos de páginas específicas** por diferentes razones, si el número de ellas es limitado.

Validar con una lista de elementos permitidos suele ser más adecuado para entradas de usuario, ya que es mucho más fácil **definir exactamente lo que está autorizado y prohibir el resto**. En el caso de **datos bien estructurados** (fechas, números de seguridad social, códigos postales, correos electrónicos...) es factible definir un patrón de validación muy fuerte, **generalmente con las expresiones regulares mencionadas**, que suelen estar **predefinidas y estandarizadas para tipos de datos comunes**.

No te olvides además que, si la entrada solo admite un conjunto fijo de opciones, como una lista desplegable, debe coincidir exactamente con uno de los valores ofrecidos al usuario o el usuario estará haciendo cosas indebidas...

5.3.2. Formas de implementar la validación

La **validación de entradas** se puede implementar con cualquier técnica que garantice la corrección sintáctica y semántica; por lo general es una de estas:

- **Validadores nativos** de cualquier *framework* moderno usado para crear el sitio web:

- **Validación con esquemas JSON y XML (XSD)** para las entradas que siguen estos formatos de fichero.
- **Conversión de tipos** (Ej.: `Integer.parseInt()` en *Java*, `int()` en *Python*...) **y control estricto de excepciones**. Esto es más importante de lo que parece, y para muestra tenemos estos dos ejemplos reales:
- **Comprobación del valor mínimo y máximo** para los parámetros y fechas, y longitud para cadenas.
- **Comprobación estricta sobre un conjunto de valores permitidos:** especialmente para parámetros que sólo pueden tener un pequeño número de valores posibles (días de la semana, meses...).
- **Expresiones regulares para cualquier otro dato estructurado.** Se debe validar toda la cadena de entrada, evitando el uso de comodines.

El **texto de formato libre**, especialmente ***Unicode***, puede ser difícil de validar ya que tiene un gran conjunto de caracteres.

> ***Si puedes evitarlo, intenta no tener esta funcionalidad en una primera fase para acelerar el tiempo de desarrollo y disminuir los problemas de seguridad potenciales.***

Si no queda más remedio, **es importante codificar el texto de acuerdo con el contexto de la aplicación**. Por ejemplo, si el texto admite legítimamente caracteres potencialmente peligrosos (', <, >...), deben codificarse correctamente a lo largo del ciclo de vida de esos datos. Para validar las entradas de texto de forma libre se usa:

- **Normalización:** Asegurarse de que hay una codificación canónica que se usa en toda la aplicación, y que no hay caracteres no válidos presentes en ningún momento.
- **Categorías de caracteres permitidos:** *Unicode* tiene categorías como "dígitos" o "letras". No sólo cubren el alfabeto latino, sino también otros alfabetos globales (Ej.: árabe, cirílico...), lo que podemos usar a nuestro favor.

- **Lista de caracteres individuales permitidos en casos especiales.** Ej.: Permitir letras y apóstrofes (Ej.: nombres irlandeses), pero no toda la categoría de signos de puntuación para un dato determinado.

No obstante, este aspecto es más complejo en muchos casos y requiere una inversión de tiempo para estudiar la forma correcta de validar el texto. Puedes encontrar más información en **[60]**.

5.3.3. Sobre la codificación de datos

De acuerdo con el requisito RS1.3, recuerda que debería evitarse todo lo posible reproducir la entrada del usuario en las respuestas HTML. Si hay que hacerlo, **todos los datos controlados por el usuario recibidos en el servidor deben ser codificados** cuando forman parte de la respuesta HTML a una petición. Por ejemplo, `<script>` sería devuelto como `&lt;script&gt;`.

Recuerda: Si no hacemos esto, facilitamos ataques de inyección de código.

El tipo de codificación que necesitan los datos recibidos de los usuarios **depende del contexto** de la página donde se insertan los mismos:

- Por ejemplo, la **codificación de entidades HTML** es apropiada para los datos colocados en el cuerpo HTML de la respuesta.
- Sin embargo, datos de usuario puestos dentro de scripts de una página de respuesta HTTP requerirían **codificación específica de *JavaScript***.
- Lo mismo sucede con los datos de usuario colocados dentro de CSS, URLs... la **codificación de entidades HTML por sí sola no es suficiente**.

Este es un aspecto más complejo de lo que parece y al que se debe prestar atención salvo que, como veremos al final de esta sección, el *framework* de desarrollo tenga herramientas para automatizar este proceso. Puedes encontrar una guía completa sobre cómo hacer bien la codificación en **[61]**. Aquí solo vamos a dar una introducción rápida que sirve para una gran cantidad de casos.

Aunque hay muchos vectores de **ataque XSS**, seguir una serie de reglas simples dadas en la referencia anterior nos permite prevenirlos. Lo primero es que debemos tratar las páginas HTML como una **"plantilla" con *placeholders***; El desarrollador **sólo puede colocar datos que no son de confianza dentro de estos *placeholders* y nunca en otro sitio que no sea alguno de ellos**.

Para cada uno de esos *placeholders*, implementamos reglas de seguridad ligeramente diferentes ¿Por qué? Porque cada *placeholder* tiene un contexto diferente y las acciones para evitar que el contenido se interprete como código cambian según ese contexto. Colocar un dato fuera de los *placeholders* requiere un esfuerzo extra para asegurarse de que no son un problema es demasiado alto, y en el contexto de esta guía queremos precisamente ahorrarnos estos problemas. En realidad, **las reglas 2 y 3 del enlace OWASP anterior cubren la mayoría de los casos comunes** (***Figura 42***).

XSS Prevention Rules Summary

The following snippets of HTML demonstrate how to safely render untrusted data in a variety of different contexts.

Data Type	Context	Code Sample	Defense
String	HTML Body	`<span>UNTRUSTED DATA </span>`	HTML Entity Encoding (rule #1).
String	Safe HTML Attributes	`<input type="text" name="fname" value="UNTRUSTED DATA ">`	Aggressive HTML Entity Encoding (rule #2), Only place untrusted data into a list of safe attributes (listed below), Strictly validate unsafe attributes such as background, ID and name.
String	GET Parameter	`<a href="/site /search?value=UNTRUSTED DATA ">clickme</a>`	URL Encoding (rule #5).
String	Untrusted URL in a SRC or HREF attribute	`<a href="UNTRUSTED URL ">clickme</a> <iframe src="UNTRUSTED URL " />`	Canonicalize input, URL Validation, Safe URL verification, Allow-list http and HTTPS URLs only (Avoid the JavaScript Protocol to Open a new Window), Attribute encoder.
String	CSS Value	`HTML <div style="width: UNTRUSTED DATA ;">Selection</div>`	Strict structural validation (rule #4), CSS Hex encoding, Good design of CSS Features.
String	JavaScript Variable	`<script>var currentValue='UNTRUSTED DATA ';</script> <script>someFunction('UNTRUSTED DATA ');</script>`	Ensure JavaScript variables are quoted, JavaScript Hex Encoding, JavaScript Unicode Encoding, Avoid backslash encoding (`\"` or `\'` or `\\`).
HTML	HTML Body	`<div>UNTRUSTED HTML</div>`	HTML Validation (JSoup, AntiSamy, HTML Sanitizer...).
String	DOM XSS	`<script>document.write('UNTRUSTED INPUT: ' + document.location.hash );<script/>`	DOM based XSS Prevention Cheat Sheet

Figura 42. Reglas de prevención de XSS recomendadas por OWASP

Tampoco es en absoluto recomendable codificar manualmente los datos. Hay bibliotecas públicas, gratuitas y muy probadas que harán un mejor trabajo. Algunos **ejemplos** son:

Usa **motores de plantillas** (*template engines*) para prevenir XSS con codificación de caracteres.

Es importante también consultar ***Spring security*** para prevenir varios tipos de ataques.

Vamos a poner como ejemplo práctico el proyecto ***OWASP Java Encoder*** **[62]**, que es otra alternativa de la lista anterior. Proporciona muchas funciones de codificación para ayudar a defenderse contra XSS en una **variedad de contextos diferentes**: HTML, JavaScript, XML y CSS, sin dependencias y con buen rendimiento. La codificación se realiza con la clase `Encode`, que contiene métodos para los múltiples contextos de codificación admitidos. Ejemplos de codificación dependiente del contexto son:

5.3.3.1. HTML

- **Básico:** `<body><%= Encode.forHtml (DATOS NO FIABLES) %></body>`
- **Contenido:** `<textarea name="text"><%= Encode.forHtmlContent (DATOS NO FIABLES) %></textarea>`
- **Atributos:** `<input type="text" name="name" value="<%= Encode.forHtmlAttribute (DATOS NO FIABLES) %>"/>`

5.3.3.2. CSS

- `<div style="width:<= Encode.forCssString (DATOS NO FIABLES) %>">`

5.3.3.3. JavaScript

- **Bloque:** `<script type="text/javascript">var txt="<%=Encode.forJavaScriptBlock (DATOS NO FIABLES) %>"; … </script>`
- **Variable:** `<button onclick="alert('<%= Encode.forJavaScriptAttribute (DATOS NO FIABLES) %>');"> Submit </button>`

5.3.3.4. URL

- **Valores de parámetros:** `<a href="/search?value=<%= Encode.forUriComponent (DATOS NO FIABLES) %>&order=1#top">`
- **Parámetros de una URL REST:** `<a href="/public/<%= Encode.forUriComponent (DATOS NO FIABLES) %>">`
- **Manejo de una URL completa que no es de confianza:** Al controlar una dirección URL completa, comprobar primero que es una dirección URL legal: `String url = validateURL(untrustedInput)`

  A continuación, **codificar la dirección URL como un atributo HTML** al escribirla en la página, pero el texto enlazable debe codificarse en un contexto diferente:

```
<a href="<%= Encode.forHtmlAttribute(URL no fiable) %>">
<%= Encode.forHtmlContent(Nombre del enlace no fiable) %>
</a>
```

5.3.4. "Saneamiento" de la entrada HTML (*HTML sanitizer*)

Si bien, como comentamos en los requisitos, es muy importante no tratar de "arreglar" una entrada cuando detectamos que es inválida, hay escenarios donde el "saneamiento" de la entrada es necesario.

> ***Esto ocurre por ejemplo cuando se admite como entrada válida un subconjunto de HTML, por ejemplo, para permitir comentarios con estilos HTML básicos (negrita, cursiva, subrayado...)***

En esos casos, es necesario **sanear/limpiar el HTML** introducido por el usuario cuando únicamente es válido un subconjunto de este en un contexto determinado y, una vez más, debemos confiar en herramientas probadas de terceros para realizar este trabajo.

Uno de las más populares es ***OWASP Java Html Sanitizer*** **[63]** que define una serie de políticas que determinan los elementos HTML permitidos (puede definir elementos personalizados) y se usa como aparece en la ***Figura 43***.

- `BLOCKS:` Permite elementos de bloque comunes, incluyendo `<p>, <h1>` etc.
- `FORMATTING:` Permite elementos de formato comunes, incluidos `<b>, <i>` etc.
- `IMAGES:` Permite elementos <img> de HTTP, HTTPS y orígenes relativos.
- `LINKS:` Permite HTTP, HTTPS, MAILTO, y enlaces relativos.
- `STYLES:` Permite ciertas propiedades CSS seguras en atributos `style= "..."`.
- `TABLES:` Permite elementos de tabla comunes.

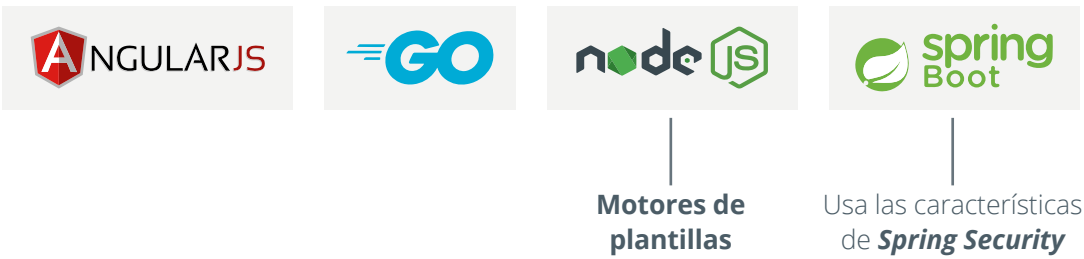
```
import org.owasp.html.Sanitizers;
import org.owasp.html.PolicyFactory;
PolicyFactory sanitizer = Sanitizers.FORMATTING.and (Sanitizers.BLOCKS);
String cleanResults = sanitizer.sanitize ("<p>Whatever <b>it takes</b>");
```

Figura 43. Ejemplo de Java Html Sanitizer

5.3.5. Plantillas de codificación automática para contextos web

Muchos *frameworks* de aplicaciones (especialmente los *frameworks* web) proporcionan **plantillas que codifican automáticamente elementos según el contexto**. Obviamente, este tipo de tecnologías nos ahorran muchos problemas, ya que nos olvidamos de la complejidad de las operaciones de codificación que vimos en la sección 5.3.3, y nos permite ahorrar costes de tiempo y recursos.

Por tanto, elegir un *framework* que soporte estas tecnologías debería ser una de nuestras prioridades. Algunos **ejemplos** son:



5.4. Autenticación con servicios de terceros

Es común tener un sistema de identidades basado en ***Active Directory*** de **Microsoft** si vamos a diseñar un producto para un ecosistema de aplicaciones *Windows*. Otros proveedores de nube típicos tienen sus sistemas propios de autenticación. Por ejemplo:

Google amazon

Debemos elegir y usar uno que sea compatible con nuestros requisitos y los potenciales clientes de nuestra *start-up*. Si esto no fuera una opción por tener unos requisitos muy particulares (Ej.: Administración pública), lo mejor es autenticarse con un protocolo establecido como ***OAuth*** a través de una **librería** adecuada adaptada a nuestro *framework* de desarrollo que nos facilite la implementación.

Ten en cuenta que los cuatro mecanismos de autenticación más usados son los de la ***Figura 44***, y deberíamos elegir uno de ellos para cualquier parte de nuestra infraestructura.

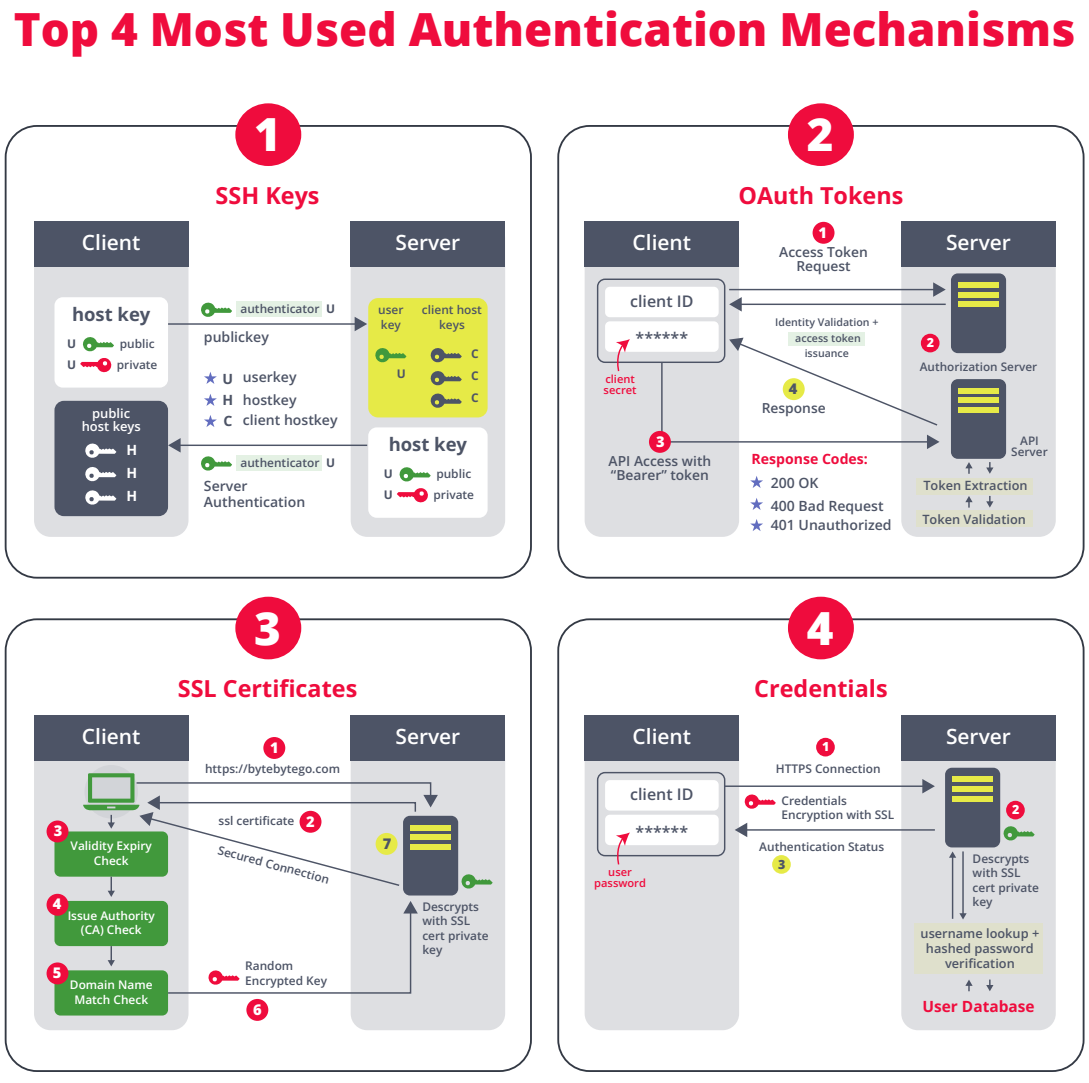


Figura 44. Métodos de autenticación más usados (bytebytego.com)

Por tanto, tal y como hemos mencionado en RS2.3, **siempre hay que tratar de usar una autenticación implementada por un tercero reconocido** (***Figura 45***).

> ***Google, Amazon o Microsoft principalmente, si bien pueden estudiarse otros proveedores complementarios de redes sociales como LinkedIn, Twitter/X, Facebook...).***

Esto, como ya mencionamos, nos da una serie de ventajas importantes que pueden acelerar el proceso de construcción del *software* y disminuyen el coste, estando muy en sintonía con el contexto de una *start-up*. Concretamente, las **ventajas principales** son:

- **Se implementa rápidamente:** hay guías de implementación para cada framework o lenguaje usado.
- Nos libramos de la responsabilidad de **mantener o testear ese código**, así como de guardar datos privados de los usuarios, teniendo por tanto menos responsabilidades legales de las que preocuparnos e invertir tiempo en cumplir.
- **El código de autenticación es más seguro**, está más probado e incorpora automáticamente mecanismos de MFA, recuperación de contraseñas, cambio de contraseñas, etc. que nosotros **NO tenemos que hacer**.

> ***A consecuencia de lo anterior, los usuarios pueden confiar más en estos sistemas que en un sistema clásico de usuario / contraseña que implementamos nosotros. Muchos usuarios sabrán que no es posible que a nuestro software le hagan una filtración de usuarios y contraseñas clásica (si no guardamos claves, no pueden robárnoslas).***

- **No les obligamos a recordar otro usuario o clave nuevo para nuestra aplicación**. Esto mejora la usabilidad y facilita que los usuarios prueben la aplicación.

Figura 45. Autenticación con múltiples servicios de terceros

Las principales **desventajas de un sistema** como este son:

- Algunos de nuestros clientes podrían **no querer compartir datos** entre distintas empresas y exigir tener *logins* separados para cada sitio.
- Algunos clientes pueden **no usar ninguno de estos proveedores** (se puede considerar usar varios para contrarrestarlo, ***Figura 45***).
- El uso de un proveedor de autenticación de terceros podría conllevar un **coste para nosotros** que impacte en el presupuesto de la *start-up*.
- Estos servicios **pueden recoger información personal de los usuarios**, lo que es inaceptable en ciertos contextos.

También debemos tener presente que una brecha en el sistema de 3ºs es automáticamente una que TENEMOS que asumir. No obstante, si alguna vez hay una brecha así en, por ejemplo, Google que eso sea un problema en nuestra aplicación será una preocupación menor para el usuario...

5.4.1. Implementación de MFA

La **implementación de un MFA** puede parecer una operación muy avanzada y difícil, pero es tan común hoy en día que hay APIs que simplifican mucho su implementación. Debemos buscar la mejor API para nuestro *framework* / lenguaje y, si es posible, utilizar soluciones de MFA de terceros reconocidas y populares. Algunos **ejemplos de implementación** son:

5.5. Autenticación sin servicios de terceros

Si, por la razón que sea, no es posible implementar un proveedor de autenticación de 3os, *OAuth* es tu solución (***Figura 46***). Es un estándar abierto para flujos de autenticación en webs y aplicaciones en general. Busca una librería en tu *framework* para facilitar su uso con el servicio al que quieras acceder (Ej.: ***Google OAuth2*** tiene librerías para una gran cantidad de lenguajes para acceder a las APIs de Google).

> ***Es muy importante que evites implementar tu propio sistema de autenticación. No solo va en contra de los principios de esta guía (ahorro de tiempo y costes) sino que estarás provocando una futura brecha de seguridad casi con total probabilidad: nunca vas a hacer un sistema de autenticación mejor que los que ya existen, probados y validados por millones de usuarios y años de experiencia y resistencia a ataques.***

En cualquier caso, recuerda: si, por la razón que sea, guardas datos privados del usuario, esto conlleva una **serie de obligaciones legales de tratamiento que debes atender** y, para ello, debes gastar tiempo y recursos (RS4.1), lo cual no va en sintonía con lo que se pretende en esta guía. Úsalo solo si no te queda más remedio para poder atender al cliente objetivo de tu *software*.

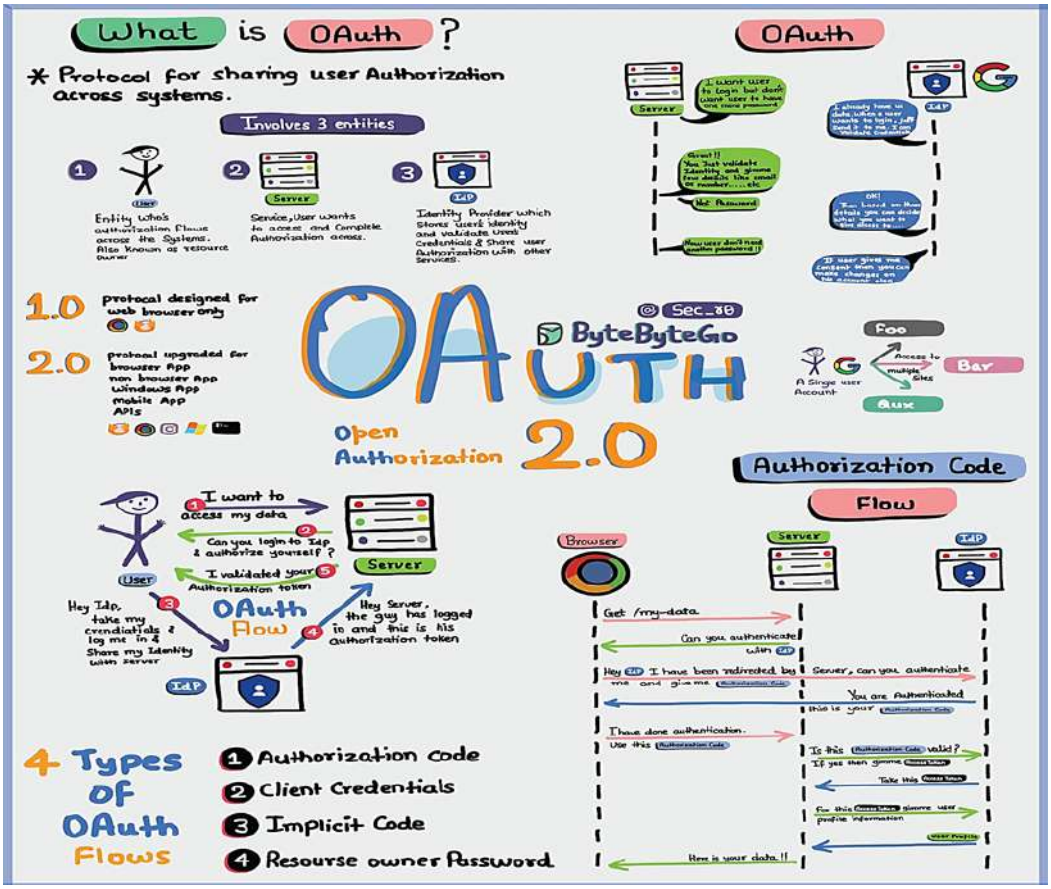


Figura 46. Funcionamiento de OAuth 2.0

5.6. Manejo de sesiones si la aplicación es web

No debes perder la perspectiva de que **las sesiones son un *token* transmitido entre el cliente y el servidor** que identifica de forma única a un usuario en un momento en el tiempo, de ahí que sea muy importante su generación y gestión correcta. En el contexto de esta guía, lo mejor es usar el mecanismo de gestión de sesiones del *framework* de desarrollo que estés empleando, y tener la última versión estable para asegurarse de que dicho mecanismo no tenga vulnerabilidades conocidas.

Debemos también recordar que las sesiones deben regenerarse en cada autenticación, re-autenticación u otro evento que cambie el nivel de privilegio del usuario, de acuerdo con el esquema de manejo general de la Figura 47.

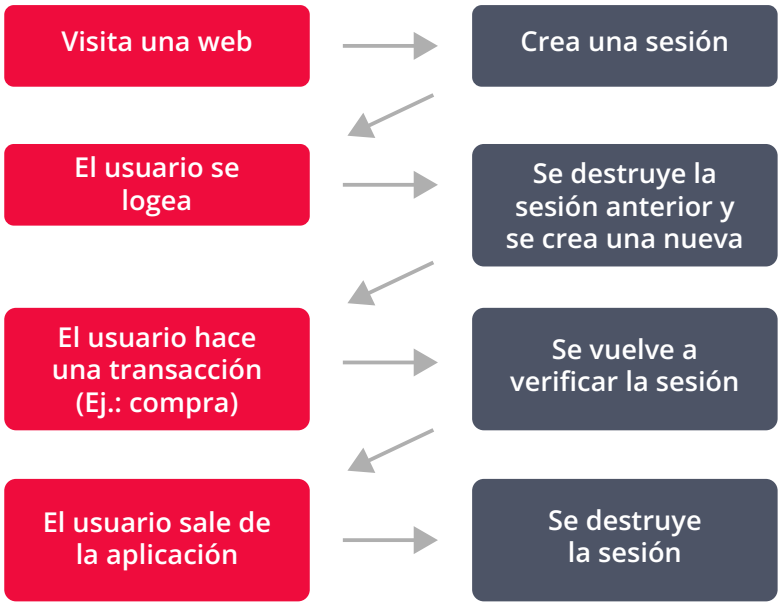


Figura 47. Secuencia general de manejo de sesiones de una aplicación

Puedes consultar más particularidades del manejo de sesiones en **[64]**.

5.7. Implementación adecuada de los *logs* de la aplicación

TODOS los errores de la aplicación deben ser capturados y manejados y **no se debe dejar NINGÚN error sin tratar**. Hacer un `catch global` (en el `main`) es bueno como último recurso para capturar errores inesperados y tratarlos de manera adecuada (**nunca debe dejarse vacío**).

En este punto, debes asegurarte de cumplir el requisito RS1.2, por el que la información interna del error (trazas de pila...) **NUNCA** debe revelarse al usuario, y que **los errores deben mostrar la menor información posible a los usuarios**, **NUNCA** versiones de productos, tecnologías...

> ***Por ejemplo, y como ya hemos dicho, nunca debes decir si se ha introducido mal un usuario o una clave, sino dar un error idéntico en ambos casos (previene la enumeración).***

Recuerda que los errores como el de la ***Figura 48*** son preferibles a cualquier otro más informativo para un usuario de nuestro *software*.

Figura 48. Error indefinido reportado correctamente al usuario

Los **eventos relevantes de seguridad** (de *login*, control de acceso, validación...) deben lanzar una alerta, en cumplimiento del requisito RS8.2. **Ejemplos** de eventos que lanzan alertas son:

- Usuario bloqueado tras varios intentos fallidos.
- Validaciones de permisos fallidas al acceder a un recurso crítico.
- Intentos de acceso a recursos para los que no se tiene permiso o pertenecen a otro usuario.
- Exceso de errores en la validación de entradas que el usuario esté introduciendo (podría ser un indicio de un proceso de *fuzzing*).

Idealmente, los *logs* de la aplicación se deberían mandar a un IDS/IPS o un SIEM que los analice (RS5.8), pero en nuestro contexto puede que no tengamos aún recursos para disponer de estos sistemas de vigilancia. No obstante, escribir desde el principio los *logs* en un formato compatible con un futuro elemento de esta clase es una **buena forma de plantear un crecimiento** en este sentido.

Al fin y al cabo, puestos a elegir un formato de log, ¿por qué no elegir uno que nos dé más opciones futuras?

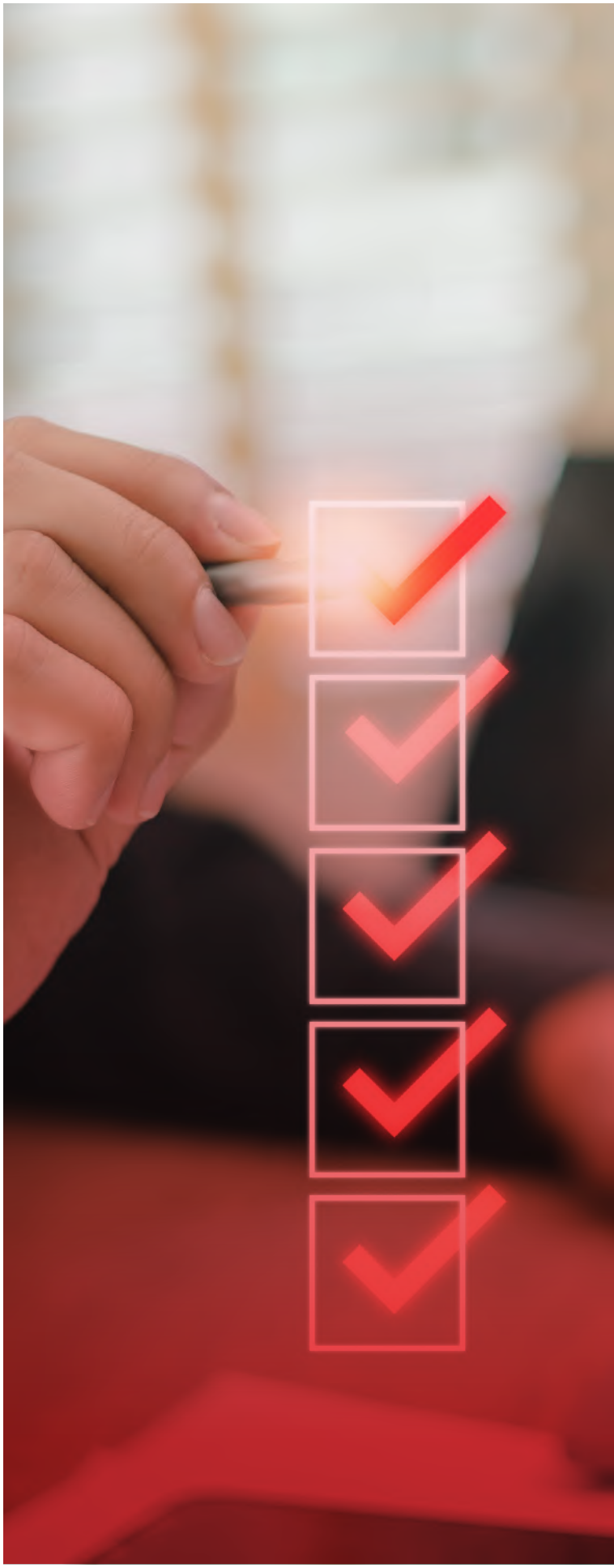
Una forma "barata" de gestionar las alertas, especialmente en fases iniciales donde el volumen de estas sea bajo, es **mandarlas por email** a una cuenta que tengamos específica para estos fines. No obstante, recuerda que el email no es un medio seguro y, salvo que usemos un proveedor para ello que lo sea (Ej.: *ProtonMail*), no deberíamos mandar en esas alertas información privada, de la misma forma que no la escribimos en los *logs*.

Cuando el error incluye información que viene de otros sistemas, **se debe codificar el contenido**, eliminar los caracteres no imprimibles y limitar el tamaño de la información que se registra para evitar problemas sobrevenidos de inyección de código. En general hay que cumplir estas **reglas** sobre la información que contienen los *logs*:

- **Recuerda:** Los *logs* no deben contener información privada o personalmente identificable.
- El **tipo de evento** que ha ocurrido (de seguridad / otro).
- **Cuándo** ha ocurrido (*timestamp*).
- **Dónde** ha ocurrido, tanto a nivel de dirección como de sistema (sistema, IP, URL, subdominio...).
- El **resultado** del evento.
- Si es posible, **identificar** al usuario u origen asociado al evento.

Otros aspectos que considerar usando y protegiendo los *logs* son:

- Deben **protegerse de accesos no autorizados, y hacer *backup*** de los mismos como parte de la estrategia de *backup* de la *start-up* que describimos anteriormente.
- Deben **guardarse en un mismo sitio centralizado** (***Figura 37***) y formato compatible con el SIEM (idealmente).
- Los accesos a los *logs* deben también **registrarse y monitorizarse**.
- Deben **guardarse cifrados y en un servidor seguro** (idealmente).
- Debe ser **accesibles** por los equipos de respuesta a incidentes.
- Deben **borrarse periódicamente**, siguiendo la misma política de la empresa para información sensible y, por supuesto, las normas legales de nuestro país.



5.8. *Checklist* de validación de prácticas de creación de código seguro

Como puedes comprobar, en toda esta sección hemos hecho un **repaso** de las prácticas de codificación más convenientes a la hora de crear la aplicación. Si necesitas una guía general que te sirva como base para establecer tus normas detalladas de codificación segura siguiendo los principios de esta guía y que puedas usar en forma de *checklist*, puedes consultar **[65]**.

Para lenguajes específicos existen también guías similares, como esta de HTML5, la de la ***Figura 49*** para ***PHP***, las de la ***Figura 50*** y la ***Figura 51*** para ***Node .js*** o la de la ***Figura 52*** para ***Angular***.

PHP Application Security Checklist

BASIC

- ☐ Strong passwords are used.
- ☐ Passwords stored safely.
- ☐ register_globals is disabled.
- ☐ Magic quotes is disabled.
- ☐ display_errors is disabled.
- ☐ Server(s) are physically secure.

INPUT

- ☐ Input from $_GET, $_POST, $_COOKIE, and $_REQUEST is considered tainted.
- ☐ Understood that only some values in $_SERVER and $_ENV are untainted.
- ☐ $_SERVER['PHP_SELF'] is escaped where used.
- ☐ Input data is validated.
- ☐ \0 (null) is discarded in input.
- ☐ Length of input is bounded.
- ☐ Email addresses are validated.
- ☐ Application is aware of small, very large, zero, and negative numbers. Sci. notation too.
- ☐ Application checks for invisible, look-alike, and combinining characters.
- ☐ Unicode control characters stripped out when required.
- ☐ Outputted data is sanitized.
- ☐ User-inputted HTML is santized with HTMLPurifier.
- ☐ User-inputted CSS is sanitized using a white-list.
  - ☐ Abusable properties (position, margin, etc.) are handled.
  - ☐ CSS escape sequences are handled.
  - ☐ JavaScript in CSS is discarded (expressions, behaviors, bindings).
- ☐ URLs are sanitized and unknown and unwanted protocols are disallowed.
- ☐ Embedded plugins are restricted from executing JS.
- ☐ Embedded plugin files (Flash movies) are embedded in a manner so that only the intended plugin is loaded.
- ☐ The application uses a safe encoding.
  - ☐ An encoding is specified using a HTTP header.
  - ☐ Inputted data is verified to be valid for your selected encoding if using an unsafe encoding.

FILE UPLOADS

- ☐ Application verifies file type.
  - ☐ User-provided mime type value is ignored.
  - ☐ Application analyzes the content of files to determine their type.
  - ☐ It is understood that a perfectly valid file can still contain arbitrary data.
- ☐ Application checks the file size of uploaded files.
  - ☐ MAX_FILE_SIZE is not depended upon.
  - ☐ File uploads cannot "overtake" available space.
- ☐ Content is checked for malicious content.
  - ☐ Application uses a malware scanner (if req.).
  - ☐ Uploaded HTML files are displayed securely.
- ☐ Uploaded files are not moved to a web-accessible directory.
- ☐ Extensive path checks are used when serving files.
- ☐ Uploaded files are not served with include().
- ☐ Uploaded files are served as an attachment using the Content-Disposition header.
- ☐ Application sends the `X-Content-Type-Options: nosniff` header.
- ☐ Files are not served as "application/octet-stream", "application/unknown", or "plain/text" unless necessary.

DATABASE

- ☐ Data inserted into the database is properly escaped or parameterized/prepared statements are used.
  - ☐ addslashes() is not used.
- ☐ Application does not have more privileges to the database than necessary.
- ☐ Remote connections to the database are disabled if they are unnecessary.

SERVING FILES

- ☐ User input is not directly used in a pathname.
  - ☐ Directory traversal is prevented.
  - ☐ Null (\0) in paths filtered.
  - ☐ Application is aware of ":" PHP streams are filtered.
- ☐ Access to files is not restricted by hiding the files.
- ☐ Remote files not included with include().

AUTHENTICATION

- ☐ Bad password throttling.
  - ☐ CAPTCHA is used.
- ☐ SSL used to prevent MITM.
- ☐ Passwords are not stored in a cookie.
- ☐ Passwords are hashed.
  - ☐ Per-user salts are used.
  - ☐ crypt() is used with sufficient number of rounds.
  - ☐ MD5 is not used.
- ☐ Users are warned about obvious password recovery questions.
- ☐ Account recovery forms do not reveal email existence.
- ☐ Pages that send emails are throttled.

SESSIONS

- ☐ Sessions only use cookies. (session.use_only_cookies)
- ☐ On logout, session data is destroyed.
- ☐ Session is recreated on authorization level change.
- ☐ Sites on the same server use different session storage dirs.

3RD-PARTIES

- ☐ CSRF issues are prevented with tokens/keys.
  - ☐ Referrers are not relied upon.
  - ☐ Pages that perform actions use POST.
  - ☐ Important pages (logout, etc.) are protected.
- ☐ Your pages are not written in a way (i.e. JSON, JS-like) where they can be included and read on a remote website successfully.
- ☐ Aware that Flash can bypass referrer checks to load images and sound files.
- ☐ The following things will not reveal significant information if included remotely:
  - ☐ Images.
  - ☐ Pages that take a longer time to load.
  - ☐ CSS files.
  - ☐ Existence or ordering of frames.
  - ☐ Existence of a JS variable.
  - ☐ Detected visit of a URL.
- ☐ Inclusion of your website in an inline frame with JS disabled does not reveal a threat.
- ☐ Application uses frame bursting code and sends the `X-Frame-Options` header.

MISCELLANEOUS

- ☐ A cryptographically secure PRNG is used for secret randomly-generated IDs (activation links, secret IDs, etc.).
  - ☐ Suhosin is installed or you are not using rand() or mt_rand() for this.
- ☐ Anything that consumes a lot of resources should be throttled and limited.
  - ☐ Pages that use 3rd-party APIs are throttled.
- ☐ You did not create your own encryption algorithm.
- ☐ Arguments to external programs (i.e. exec()) are validated.
- ☐ Generic internal and external redirect pages are secured.
- ☐ Precautions taken against the source code of your PHP pages being shown due to misconfiguration.
- ☐ Configuration and critical files are not in a web-accessible directory.

SHARED HOSTING

- ☐ Using a secure shared host where users cannot access the files of other users.
- ☐ Aware that fellow shared hosting users:
  - ☐ Can, if on the same IP address, issue requests against your site with XMLHttpRequest in IE6.
  - ☐ Can access your website from 127.0.0.1 or ::1.
  - ☐ Can host a server on the same IP address.
  - ☐ Are not "remote" as far as your DB is concerned.
- ☐ Session & file upload directories are not shared.

Figura 49. Checklist para validar si un código es seguro en PHP

Spring Boot Security Cheat Sheet

Use HTTPS in Production

To use HTTPS in your Spring Boot app, extend WebSecurityConfigurerAdapter and require a secure connection (Note: this forces HTTPS in development also):

```
@Configuration
public class WebSecurityConfig extends
                WebSecurityConfigurerAdapter {
 @Override
 protected void configure(HttpSecurity http)
                        throws Exception {
   http.requiresChannel().requiresSecure();
 }
}
```

Enable CSRF Protection

Spring Security enables CSRF support by default. If you use a JavaScript framework, configure the CookieCsrfTokenRepository so cookies are not HTTP-only.

```
@EnableWebSecurity
public class WebSecurityConfig extends
                WebSecurityConfigurerAdapter {
 @Override
 protected void configure(HttpSecurity http)
                        throws Exception {
 http.csrf()
    .csrfTokenRepository(CookieCsrfTokenRepository
                .withHttpOnlyFalse());
 }
}
```

Use OpenID Connect

OpenID Connect (OIDC) provides user information via an ID token in addition to an access token. Query the /userinfo endpoint for additional user information.

Use Password Hashing

Don't store passwords in plain text. Spring Security doesn't allow plain text passwords by default. PasswordEncoder is the main interface for password hashing in Spring Security:

```
public interface PasswordEncoder {
  String encode(String rawPasswd);
  boolean matches(String rawPasswd, String encPasswd);
}
```

Test Your Dependencies

Ensure your application does not use dependencies with known vulnerabilities. Use a tool like Snyk to:

- Test your app dependencies for known vulnerabilities.
- Automatically Fix issues that exist.
- Continuously Monitor for new vulns.

Use a Content Security Policy

Enable to avoid XSS attacks.

Spring Security provides a number of security headers by default, but not CSP. Enable in your Spring Boot app as follows:

```
@EnableWebSecurity
public class WebSecurityConfig extends
                WebSecurityConfigurerAdapter {
 @Override
 protected void configure(HttpSecurity http)
                        throws Exception {
 http.headers().contentSecurityPolicy("script-src
'self' https://trustedscripts.example.com; object-src
https://trustedplugins.example.com; report-uri /csp-
report-endpoint/");
 }
}
```

Use the Latest Releases

start.spring.io site automatically uses the latest version of libraries for new apps.

For existing apps, when upgrades aren't possible, consider patches from a security vendor, like Snyk.

Store Secrets Securely

Store secrets in Vault by HashiCorp or Spring Vault. Extract secrets from the Spring Vault using annotations.

```
@Value("${password}")
String password;
```

Pen Test Your App

The OWASP ZAP security tool is a proxy that performs penetration testing. It runs in Spider and Active Scan modes to identify and map all hyperlinks in your app, and automatically test your selected targets against a list of potential vulnerabilities.

Have Your Security Team do a Code Review

Code reviews are essential. Ensure all your code changes urdergo:

- A security team code review.
- Automatic testing on every pull request using Snyk

incibe_
INSTITUTO NACIONAL DE CIBERSEGURIDAD

Figura 50. Checklist para validar si un código es seguro en Node.js (I)

Cheat Sheet: 10 npm Security Best Practices

Avoid publishing secrets to the npm registry

1. Run npm publish --dry-run to review the package before publishing
2. Put sensitive files in .gitignore
3. Use the files property in package.json to whitelist files and directories

Enforce lockfile

Freeze lockfile and ensure the npm CLI installs per lockfile only, without changing it. In CI and build environments favor:

1. `$ npm ci`
2. `$ yarn install --frozen-lockfile`

Minimize attack surface—ignore run-scripts

Malicious packages take advantage of key lifecycle events when an npm install runs arbitrary commands. To minimize this attack surface:

1. Assess a project's health status and credibility before installing a package
2. Disable run-scripts during install such as:

```
$ npm install <package> --ignore-scripts
```

Assess npm project health

Review a project for outdated dependencies, and assess environment health with CLI commands:

```
$ npm doctor
$ npm outdated
```

Scan and monitor for vulnerabilities in open source dependencies

Don't let vulnerabilities in your project dependencies reduce the security of your application. Make sure to:

1. Connect Snyk to GitHub or other SCMs for optimal CI/CD 1 integration with your projects
2. Run snyk test to scan a new project from the CLI.
3. Run snyk monitor to track and open PRs to automatically fix security vulnerabilities in open source dependencies.

Use a local npm proxy

A local private registry such as Verdaccio will give you an extra layer of security, enabling you:

1. Full control of lightweight private package hosting
2. To cache packages and avoid being affected by network and external incidents

Easily spin up verdaccio using docker:

```
$ docker run verdaccio/verdaccio
```

Responsible disclosure

Publicly disclosed security vulnerabilities without prior warning and proper coordination pose a potentially serious threat.

We are happy to collaborate on responsible security disclosures for the npm community:

1. Report a security issue via the vulnerability disclosure form
2. Email us at security@snyk.io

Responsible disclosure

Publicly disclosed security vulnerabilities without prior warning and proper coordination pose a potentially serious threat.

We are happy to collaborate on responsible security disclosures for the npm community:

1. Report a security issue via the vulnerability disclosure form
2. Email us at security@snyk.io

Enable 2FA

Enable two-factor authentication on npm with

```
$ npm profile enable-2fa auth-and-writes
```

Use npm author tokens

Make use of restricted tokens for querying npm packages and functionalities from CI by creating a read-only and IPv4 address range restricted token:

```
$ npm token create --read-only —cidr=192.0.2.0/24
```

Understand typosquatting risks

Typos in package installation can be deadly.

1. Be mindful when copy-pasting package install instructions to the terminal and verify authenticity.
2. Opt to have a logged-out npm user in your developer environment
3. Favor npm install with --ignore-scripts

incibe_
INSTITUTO NACIONAL DE CIBERSEGURIDAD

Figura 51. Checklist para validar si un código es seguro en Node.js (II)

Pragmatic Web Security
Security for developers

Security Cheat Sheet
Version 2020.001

Angular and the OWASP top 10

The OWASP top 10 is one of the most influential security documents of all time. But how do these top 10 vulnerabilities resonate in a frontend JavaScript application?

This cheat sheet offers practical advice on handling the most relevant OWASP top 10 vulnerabilities in Angular applications.

DISCLAIMER This is an opinionated interpretation of the OWASP top 10 (2017), applied to frontend Angular applications. Many backend-related issues apply to the API-side of an Angular application (e.g., SQL injection), but are out of scope for this cheat sheet. Hence, they are omitted.

1 Using *dependencies* with known vulnerabilities

OWASP #9

- ☐ Plan for a periodical release schedule
- ⚙ Use `npm audit` to scan for known vulnerabilities
- ☐ Setup automated dependency checking to receive alerts
  Github offers automatic dependency checking as a free service
- ☐ Integrate dependency checking into your build pipeline

2 Broken authentication

OWASP #2

From an Angular perspective, the most important aspect of broken authentication is maintaining state after authentication. Many alternatives exist, each with their specific security considerations.

- ⚙ Decide if a stateless backend is a requirement
  Server-side state is more secure, and works well in most cases

Server-side session state

- ☐ Use long and random session identifiers with high entropy
  OWASP has a great cheat sheet offering practical advice [1]

Client-side session state

- ☐ Use signatures to protect the integrity of the session state
- ⚙ Adopt the proper signature scheme for your deployment
  HMAC-based signatures only work within a single application
  Public/private key signatures work well in distributed scenarios
- ☐ Verify the integrity of inbound state data on the backend
  Explicitly avoid the use of "decode-only" functions in libraries
- ☐ Setup key management / key rotation for your signing keys
- ☐ Ensure you can handle session expiration and revocation

Cookie-based session state transport

- ☐ Enable the proper cookie security properties
  *Set the **HttpOnly** and **Secure** cookie attributes*
  *Add the **__Secure** or **__Host-** prefix on the cookie name*
- ☐ Protect the backend against Cross-Site Request Forgery
  *Same-origin APIs should use a **double submit cookie***
  *Cross-Origin APIs should force the use of CORS preflights by only accepting a non-form-based content type (e.g. **application/json**)*

Authorization header-based session state transport

- ☐ Only send the authorization header to pre-approved hosts
  *Many custom interceptors send the header to **every** host*

[1] https://bit.ly/2U8kJWc

3 Cross-Site Scripting

OWASP #7

Preventing html/script injection in Angular

- ☐ Use interpolation with `{{}}` to automatically apply escaping
- ☐ Use safe property binding such as [href], [src], [style.color]
- ☐ Use binding to [innerHTML] to safely insert HTML data
- ☐ Do not use `bypassSecurityTrust*()` on untrusted data

Preventing code injection outside of Angular

- ☐ Avoid direct DOM manipulation
  *E.g. through **ElementRef** or other client-side libraries*
- ☐ Do not combine Angular with server-side dynamic pages
- ☐ Use Ahead-Of-Time compilation (AOT)

4 Broken access control

OWASP #5

Authorization checks

- ☐ Implement proper authorization checks on API endpoints
  Check if the user is authenticated
  Check if the user is allowed to access the specific resources
- ☐ Do not rely on client-side authorization checks for security

Cross-Origin Resource Sharing (CORS)

- ☐ Prevent unauthorized cross-origin access with a strict policy
- ☐ Avoid accepting the ***null*** origin in your policy
- ☐ Avoid blindly reflecting back the value of the origin header
- ☐ Avoid custom CORS implementations
  Origin-matching code is error-prone, so prefer the use of libraries

5 Sensitive data exposure

OWASP #3

Data in transit

- ☐ Serve everything over HTTPS
- ☐ Ensure that all traffic is sent to the HTTPS endpoint
  Redirect HTTP to HTTPS on endpoints dealing with page loads
  Disable HTTP on endpoints that only provide an API
- ☐ Enable ***Strict Transport Security*** on all HTTPS endpoints

Data at rest in the browser

- ☐ Encrypt sensitive data before persisting it in the browser
- ☐ Encrypt sensitive data in JWTs using JSON Web Encryption

Figura 52. Checklist para validar si un código es seguro en Angular

Cada *framework* / tecnología tiene sus funcionalidades mejoradas de seguridad y podemos encontrar cheatsheets de seguridad en el desarrollo asociados a muchos de ellos aquí. Ejemplos:

.NET Framework

REST

(Hay muchas más disponibles para cualquier tecnología o lenguaje que puedas usar)

> ***Finalmente, ten en cuenta que si la aplicación está alojada en la nube el panorama de amenazas cambia, y es algo que debemos tener en cuenta antes de su despliegue [66].***

5.9. Aspectos básicos de usabilidad

Es importante tener presente que, si las medidas de seguridad de una aplicación la hacen difícil de usar, **los usuarios tenderán a tratar de evitarlas**... o irse a la competencia. Es necesario hacer un esfuerzo en hacer la seguridad mínimamente usable. Este es un tema complejo que excede los objetivos de esta guía por ser algo que requiere bastantes recursos, pero aquí tienes algunos **materiales sobre diseño de UI desde la perspectiva de seguridad** por si quieres implementarlos a medida que el *software* evolucione.

No obstante, sí que hay dos cosas mínimas a implementar en nuestro contexto de una *start-up*:

- Cumplir estrictamente el requisito de **permitir copiar y pegar en los campos donde se introducen las claves**, para facilitar el uso de gestores de contraseñas que puedan tener nuestros usuarios.
- **No diseñar programas para "engañar" a los usuarios y que hagan cosas que no quien hacer:** Comprar productos, suscribirse a un servicio... (Ej.: artículos de más en carrito, falsos "X personas están viendo esto" ...). No solo algunas de ellas pueden comprometer la seguridad de los usuarios, sino que muchas de ellas **no son éticas** y, en caso de ser descubiertos, dañarán nuestra reputación y negocio.

Esto último puede resumirse en **no incorporar "*deceptive patterns*" a la aplicación** (consciente o inconscientemente). Se basan en que los usuarios no leen la página completa y/o asumen que se va a comportar de una forma dada para provocar que hagan una acción que nos beneficie.

> ***Dicho de otra forma: las páginas se diseñan para engañar; parece que dicen una cosa, pero realmente dicen otra.***

Como creadores de un producto, debemos conocerlos para **evitar reproducirlos**. Está claro que como *start-up* querremos aumentar las ventas de nuestros productos, pero esta no es una forma ética ni conveniente de hacerlo.

> ***Si se detecta y viraliza que los estamos empleando, puede causarnos una crisis reputacional que afecte a nuestro negocio y a nuestros ingresos.***

Una lista de ***"deceptive patterns"*** podemos encontrarla en **[67]** . La ***Figura 53*** muestra algunos.

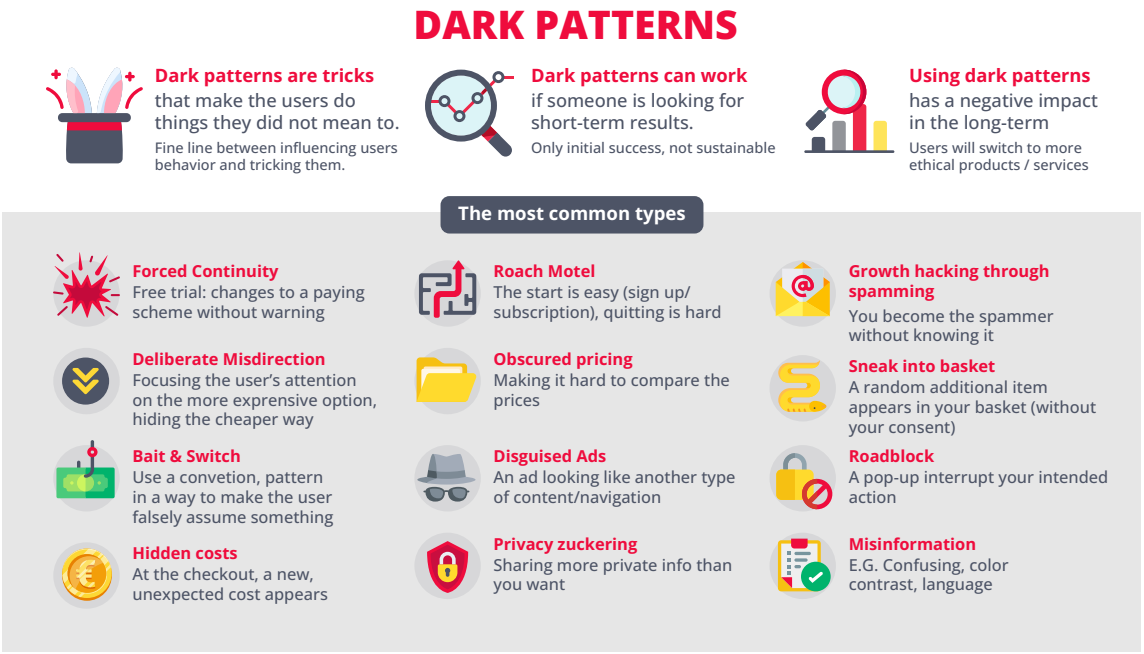


Figura 53. Ejemplos típicos de "patrones engañosos (llamados "patrones oscuros" por algunos autores)

5.10. Codificando centrados en seguridad: Atacando los errores más comunes

Una de las cosas que debemos tener en cuenta a la hora de desarrollar el código fuente de nuestro *software* es que no todos los errores de seguridad son igual de comunes, es decir, que **la probabilidad de cometer un error mientras desarrollamos la comunicación es mayor con unos tipos de errores que con otros**.

La asociación OWASP mantiene una clasificación de qué errores son más frecuentes en aplicaciones web en su índice OWASP Top 10 **[68]**. Este índice se actualiza periódicamente, y en la ***Figura 54*** puedes ver la última edición disponible en el momento de escribir esta guía (noviembre 2023).

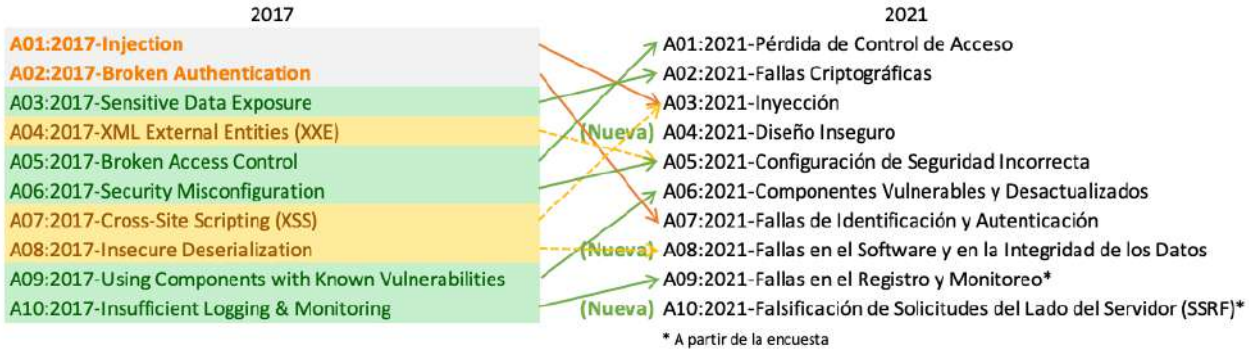


Figura 54. Top 10 de problemas de seguridad de una aplicación web

Estos errores cambian también con el tipo de aplicación diseñada, y por ejemplo para APIs son diferentes, como muestra el índice Top 10 para APIs de OWASP (***Figura 55***).

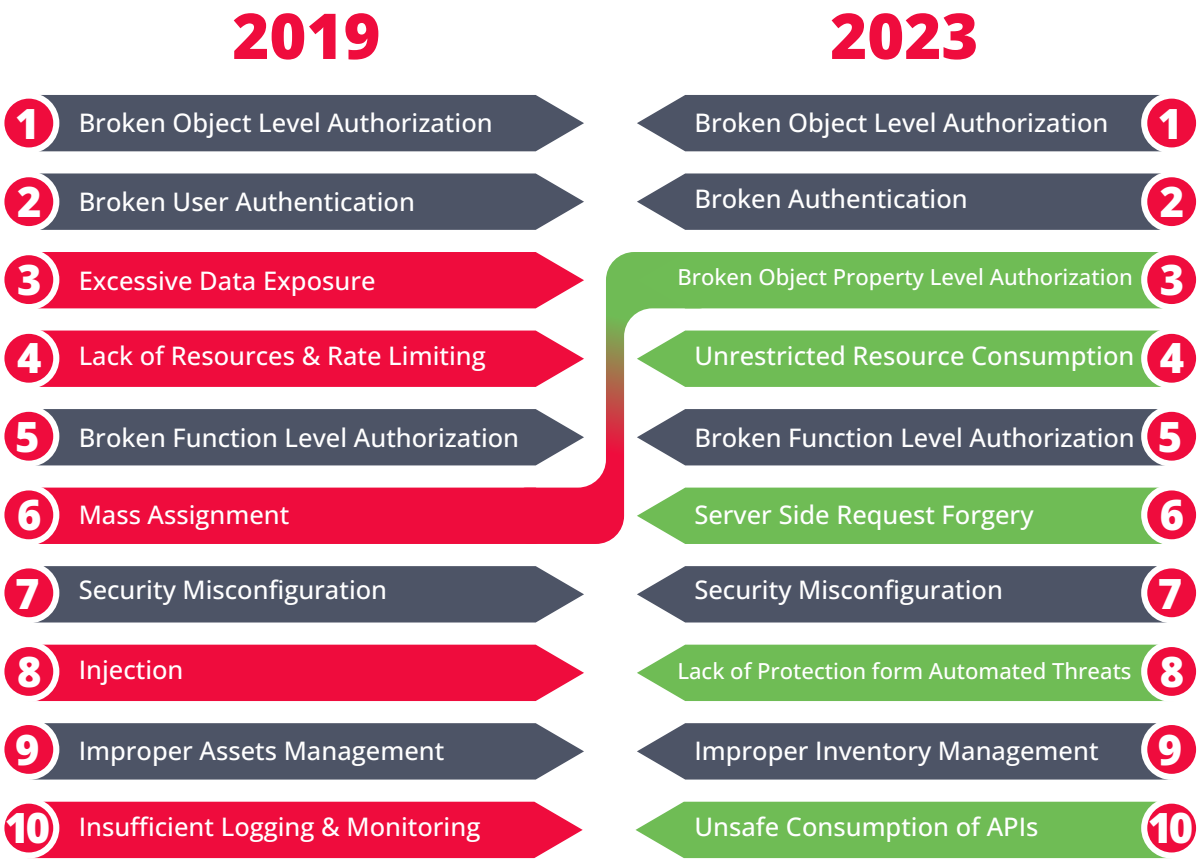


Figura 55. Top 10 de problemas de seguridad en un API

Conocer este aspecto es de importancia por ejemplo en procesos de ***code review*** (RS5.7), de forma que podamos saber qué errores ir a analizar primero, dándole así importancia a las cosas que tienen mayor probabilidad de fallo.

6. Integración rápida de herramientas de AST (*Application Security Testing*) en tu proceso de producción de *software*

6.1. Usar herramientas SCA

Ya sea integradas en *GitHub* o en el *software* de gestión del pipeline de desarrollo, conviene usar de la forma más automatizada posible este tipo de herramientas para gestionar el uso de componentes ***open source*** de tu aplicación. Estas herramientas escanean automáticamente ese código, incluyendo artefactos como contenedores y registros, con los siguientes **objetivos**:

- **Inventario:** Recopilan los componentes *open source* de la aplicación (dependencias directas e indirectas). Así es posible saber todo lo que se está usando, lo que es clave para controlar su seguridad.
- **Cumplimiento de licencias:** Dan información sobre cada uno de los componentes identificados, incluyendo si el tipo de licencia es compatible con la política de la organización y la atribución de autoría.
- **Vulnerabilidades de seguridad:** Identifica vulnerabilidades conocidas en estos componentes (**CVE**), que es su principal función y la que más nos interesa en este contexto.

> ***Informan si la aplicación llama a código vulnerable, sugieren mitigaciones, gestionan actualizaciones...y también avisan de sus vulnerabilidades antes de hacer un pull request y correr el riesgo de desplegarlos en este estado. Ten en cuenta que una vulnerabilidad en un componente de terceros es también una vulnerabilidad que tenemos nosotros (RS5.5), por lo que no debemos consentir este tipo de incidencias.***

Ejemplos de estas herramientas puedes encontrarlos aquí. Como **acciones adicionales** puedes además establecer lo siguiente:

- **Úsalas en tus repositorios de código de aplicaciones** regularmente, especialmente cuando ya esté desplegada y en fase de mantenimiento. Las vulnerabilidades pueden aparecer cuando menos te lo esperas.
- **Intégralas en tu *pipeline* CI/CD** y detén el despliegue si se encuentra alguna vulnerabilidad.

6.2. Herramientas SAST

De nuevo, ya sea integradas en *GitHub* o en el *software* de gestión del *pipeline* de desarrollo. Prueban estructuras internas o código de la aplicación, pero no su funcionalidad (son **pruebas de caja blanca**). Las principales ventajas es que soportan muchos lenguajes: *PHP*, *Java* y *.Net*, C, C++, C#... y descubren vulnerabilidades complejas durante las **1ªs etapas de desarrollo** que se pueden resolver rápidamente, lo cual es compatible con la filosofía *pushing left* vista al principio de esta guía. Habitualmente detallan los problemas encontrados, incluidas las **líneas de código**.

Los principales inconvenientes en nuestro contexto de *start-up* es que suelen ser **inexactas**: dan muchos falsos positivos y negativos que tenemos que invertir tiempo y recursos en investigar. Tampoco pueden probar la aplicación en el **entorno real** (no detectan vulnerabilidades en la lógica de la aplicación o configuraciones inseguras). La ***Figura 56*** muestra las herramientas **SAST** y las **SCA**.

Figura 56. Diferencias entre una herramienta SAST y una SCA

Ejemplos de estas herramientas puedes verlos aquí. La ***Figura 57*** muestra una muy popular (**SonarQube**, **[69]**) en funcionamiento.

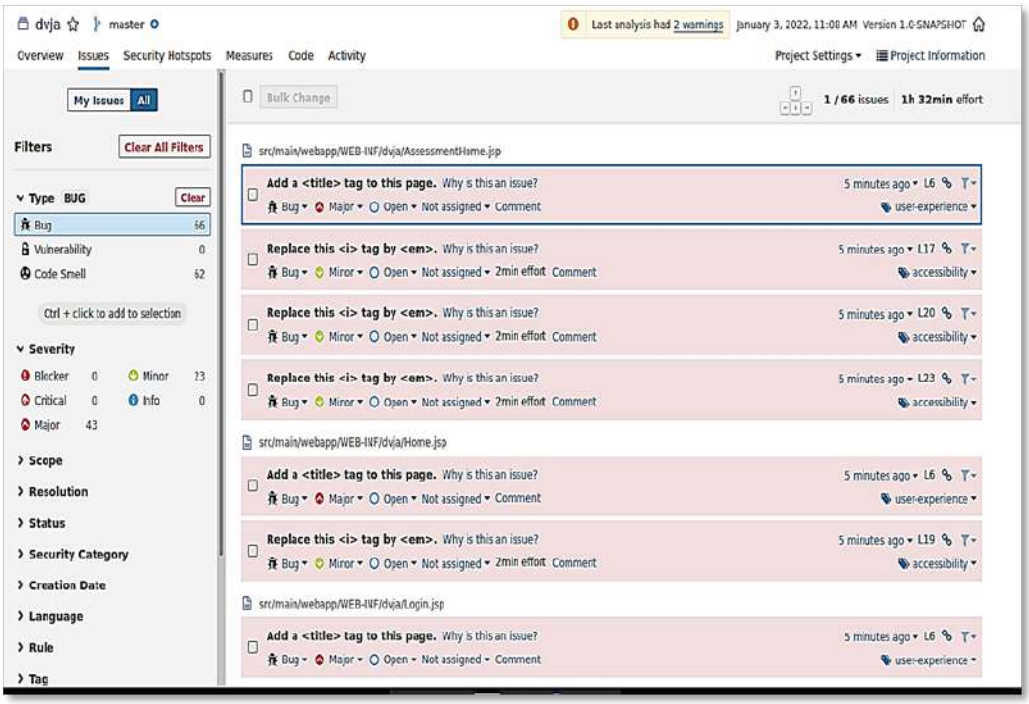


Figura 57. Herramienta SAST Sonarqube

En nuestro contexto de *start-up*, ***GitHub*** **te da la capacidad de usar las funciones SCA y SAST muy fácilmente y de manera rápida**, si bien solo si el repositorio es público por el momento. ***GitHub*** integra escáneres SCA y SAST como parte de sus ***GitHub Actions*** entre otras muchas funcionalidades.

> ***En caso de que tu repositorio sea privado, puedes usar SonarQube e integrarle un plugin con una funcionalidad SCA, o bien usar las herramientas equivalentes en un GitLab que tengas en tu red.***

Uno de los escáneres de esta clase gratuitos más populares es ***CodeQL***, y para usarlo puedes seguir las instrucciones de la ***Figura 58***. Puedes consultar otras ***GitHub Actions*** disponibles para encontrar más herramientas de ***Application Security Testing*** que se ajusten a tus desarrollos, y mejorar tu seguridad de forma automatizada a un coste mínimo.

Figura 58. Escaneo SAST y SCA en GitHub

Siguiendo los cuatro pasos de la ***Figura 58*** ya hemos incorporado características SCA (***Dependabot***) y SAST (***Code scanning***) básicas en nuestro código (***Figura 59***).

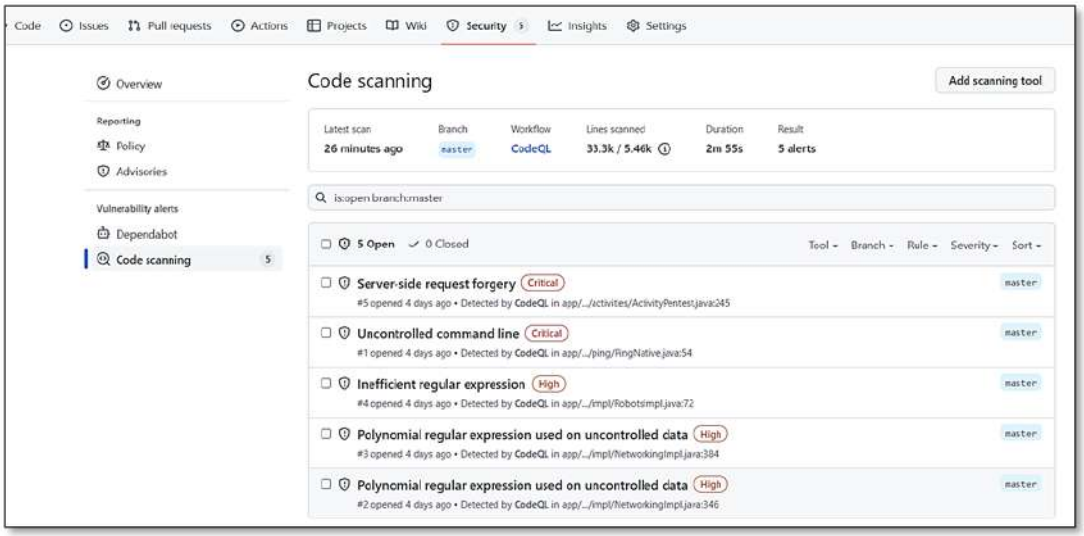


Figura 59. Informe SAST y SCA en GitHub

Las opciones de ***Code security and analysis*** activan SCA y SAST automáticamente, pero también ***secret scans***, que permiten mitigar automáticamente el problema que hablábamos antes de la presencia de secretos en el código por accidente (***Figura 60***).

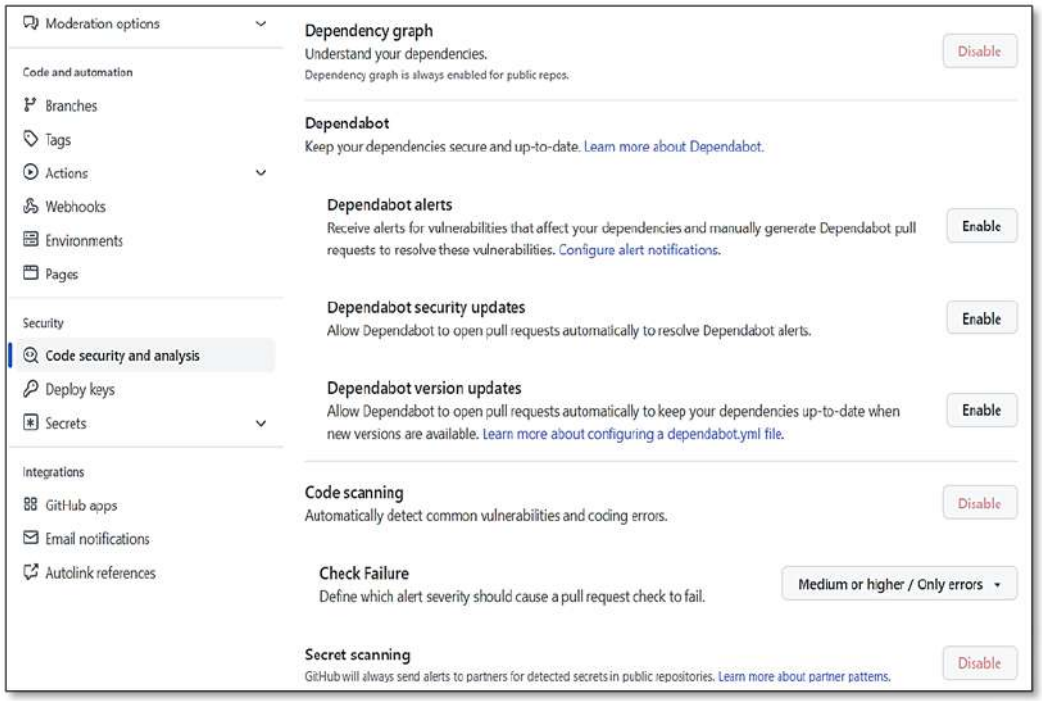


Figura 60. Opciones de escaneo de código en GitHub

6.3. Herramientas DAST

Estas herramientas son un **método de prueba de caja negra** que introduce errores en las entradas para probar la seguridad de las rutas de ejecución que estos provocan. No requieren código fuente ni binarios; **analizan la aplicación en ejecución** (***Figura 61***).

Las principales **ventajas** son que, a diferencia de las SAST, **permite detectar problemas en ejecución** (Errores de autenticación y configuración de red, problemas tras *login*...), hay menos falsos positivos y son compatibles con todos los lenguajes y/o *frameworks*, ya que lo que prueban es la aplicación funcionando.

No obstante, como principales **inconvenientes** podemos mencionar que no dan información sobre las causas de las vulnerabilidades, **no son adecuadas en las 1ªs etapas** (necesitan la aplicación en ejecución) y no simulan una parte de los posibles ataques. Una que puede incorporarse de forma gratuita al desarrollo de la aplicación es **ZAP [70]**. Si no sabes cómo funciona ZAP, el **INCIBE** tiene una formación gratuita disponible sobre ella **[71]**.

ZAP analiza todas las URL de tu web que puede alcanzar. Hay formas de automatizar el proceso y de saltarse también tu autenticación para que pueda alcanzar toda la web. No obstante, pueden ser complejas según como sea la autenticación de tu aplicación. Un método más rápido en el contexto de esta guía seguramente sea hacer un escaneo manual: navegar tú normalmente por tu software y dejar que la herramienta haga sus pruebas con el navegador especial que lanza para ello.

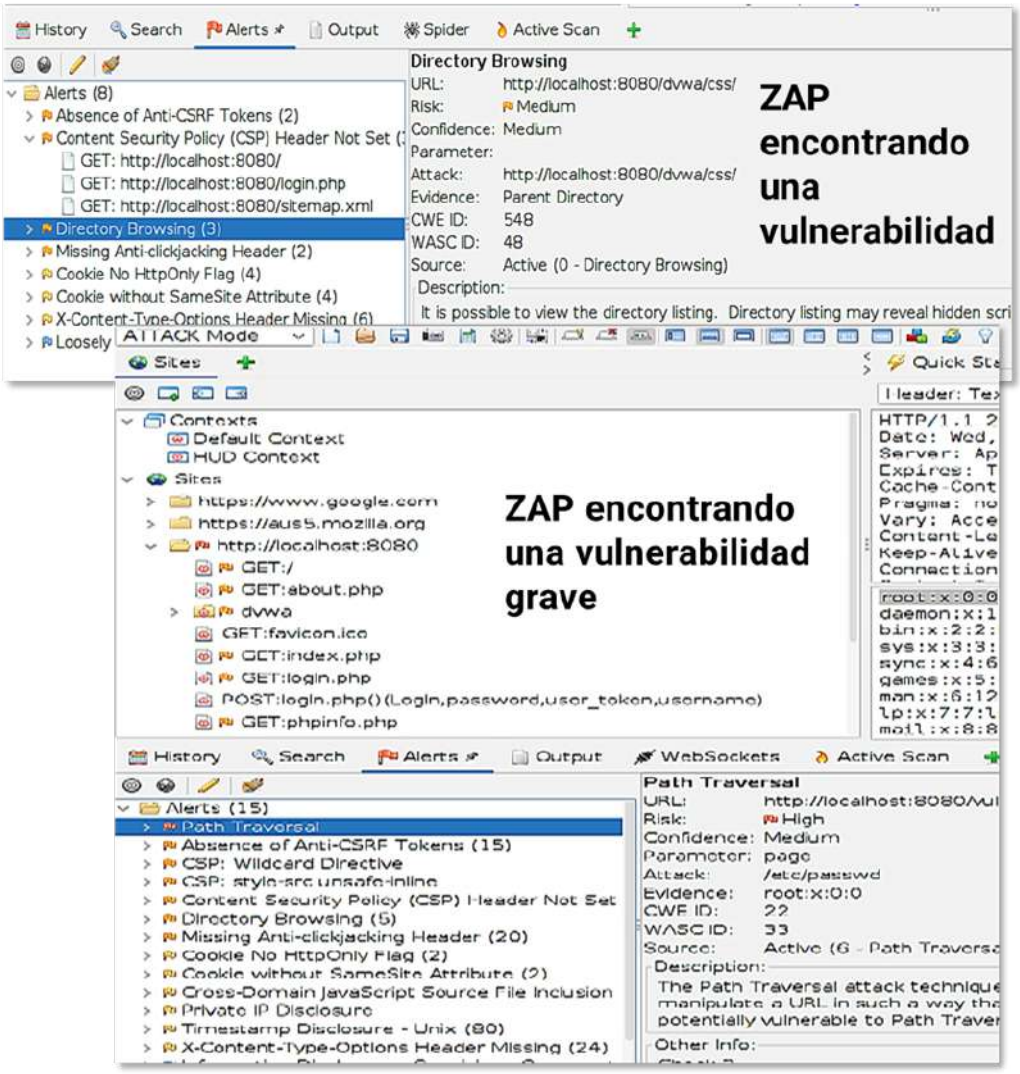


Figura 61. ZAP como herramienta DAST

La ***Figura 62*** de muestra las diferencias entre este tipo de herramientas y las anteriores. Además, podemos ver un **listado** de estas herramientas aquí.

SAST vs DAST

Static application security testing (SAST) and dynamic application security testing (DAST) are both methods of testing for security vulnerabilities, but they're used very differently.

Here are some key differences between the two:

WHITE BOX SECURITY TESTING:
- The tester has access to the underlying framework, design, and implementation.
- The application is tested from the inside out.
- This type of testing represents the developer approach.

BLACK BOX SECURITY TESTING:
- The tester has no knowledge of the technologies or frameworks that the application is built on.
- The application is tested from the outside in.
- This type of testing represents the hacker approach.

REQUIRES SOURCE CODE:
- SAST doesn't require a deployed application.
- It analyzes the source code or binary without executing the application.

REQUIRES A RUNNING APPLICATION:
- DAST doesn't require source code or binaries.
- It analyze by executing the application.

FINDS VULNERABILITIES EARLIER IN THE SDLC:
- The scan can be executed as soon as code is deemed feature-complete.

FINDS VULNERABILITIES TOWARD THE END OF THE SDLC:
- Vulnerabilities can be discovered after the development cycle is complete.

LESS EXPENSIVE TO FIX VULNERABILITIES:
- Since vulnerabilities are found earlier in the SDLC, it's easier and faster to remediate them.
- Findings can often be fixed before the code enters the QA cycle.

MORE EXPENSIVE TO FIX VULNERABILITIES:
- Since vulnerabilities are found toward the end of the SDLC, remediation often gets pushed into the next cycle.
- Critical vulnerabilities may be fixed as an emergency release.

CAN'T DISCOVER RUN-TIME AND ENVIRONMENT-RELATED ISSUES:
- Since the tool scans static code, it can't discover run-time vulnerabilities.

CAN'T DISCOVER RUN-TIME AND ENVIRONMENT-RELATED ISSUES:
- Since the tool uses dynamic analysis on an application, it is able to find run-time vulnerabilities.

TYPICALLY SUPPORTS ALL KINDS OF SOFTWARE:
- Examples include web applications, web services, and thick clients.

TYPICALLY SCANS ONLY APPS LIKE WEB APPLICATIONS AND WEB SERVICE:
- DAST is not useful for other types of software.

SAST and DAST techniques complement each other.

Both need to be carried out for comprehensive testing.

Figura 62. Diferencias entre escaneos SAST y DAST (synopsis.com)

7. Más allá: salir a producción con unas mínimas garantías de seguridad

Una vez tengamos disponible nuestro MVP, o sus posteriores evoluciones a medida que el negocio de la *start-up* vaya avanzando y prosperando, no podemos poner en producción nuestra aplicación sin por lo menos **cumplir unos mínimos** que garantizarán tener una seguridad suficiente para parar ataques habituales y muy conocidos. La unión de estas técnicas y todo lo visto de la seguridad en el desarrollo harán nuestra aplicación mucho más resistente a ataques.

En esta sección final de la guía veremos por tanto **cuáles son estos mínimos de seguridad a cumplir** en las salidas a producción de la aplicación.

7.1. Escaneo periódico de vulnerabilidades

Como hemos dicho en el RS5.6, las **vulnerabilidades** de los productos que hemos usado para la creación de la aplicación pueden ir apareciendo con el tiempo. Por otro lado, aunque la configuración inicial de la infraestructura que hayamos creado sea correcta, puede ser que a medida que la aplicación va evolucionando se cambie algún aspecto que deje el sistema vulnerable, o bien alguno de los servicios que tenemos corriendo la infraestructura tenga también vulnerabilidades nuevas descubiertas. Por ello **el sistema en producción debe someterse a una vigilancia periódica para detectar estos casos**.

Si el producto *software* de tu *start-up* es una web, hay una forma manual de analizarlo que, si bien es más lento, te permite ir mucho más al detalle. El **INCIBE** dispone gratuitamente de esta formación en el ***Programa: Fundamentos del análisis de sitios web*** **[72]**. No obstante, esto puede ser un proceso lento, que no encaje bien con lo que necesitamos en el contexto de una *start-up*. ***¿Hay una forma rápida de probar si un sistema tiene vulnerabilidades en un momento dado?*** La hay, aunque no es perfecta: **las herramientas automáticas de descubrimiento de vulnerabilidades**. Estas nos permiten obtener un esquema claro de las vulnerabilidades más evidentes que un sistema puede tener, ahorrándonos tiempo.

> ***NO sustituyen el trabajo de una auditoria de seguridad bien hecha (sino que la complementan) y no debe confundirse nunca como un pentesting o un Red Team.***

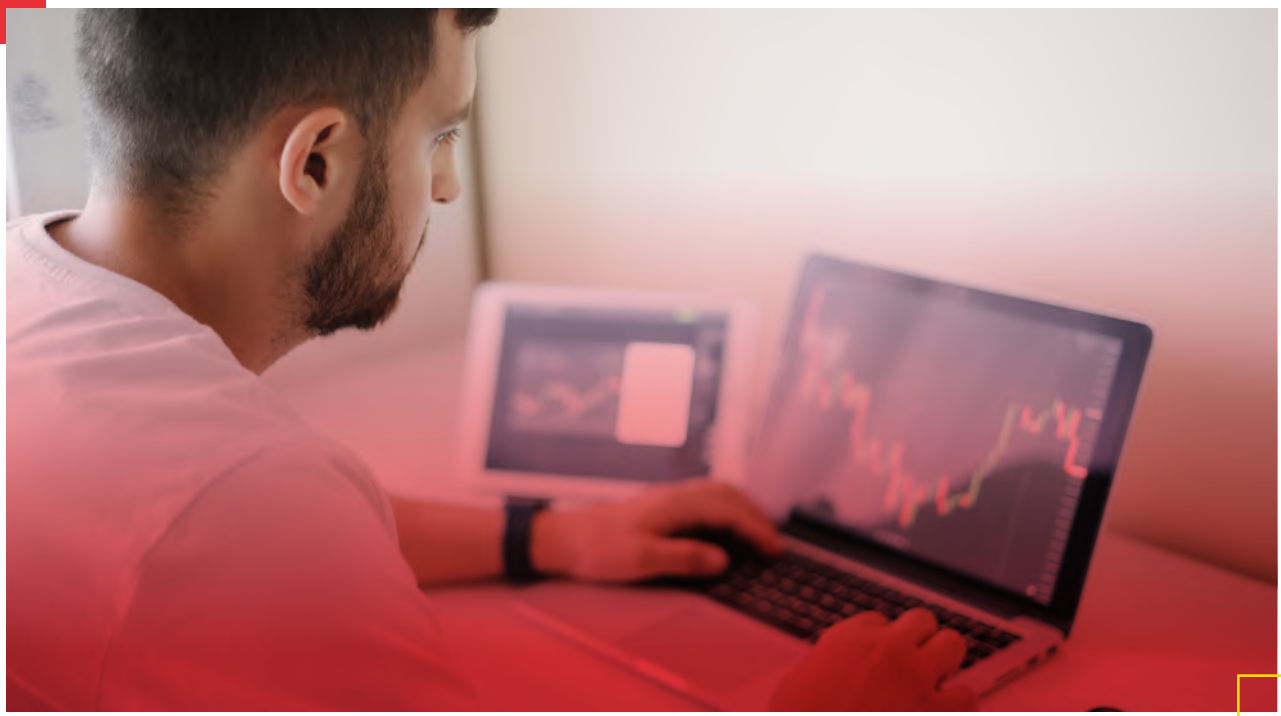
En el contexto de esta guía es una ayuda muy importante que podemos obtener a un coste muy bajo, puesto que hay herramientas de esta clase gratuitas (solo debemos tener un lugar adecuado para instalarlas). Por tanto, son **programas profesionales y muy potentes** que descubren automáticamente vulnerabilidades en cualquier máquina (*Windows, Linux, Mac OS*, servidor web o no, ...), nos hacen **ahorrar mucho tiempo** a la hora de descubrir vulnerabilidades y podemos combinar los resultados de varias para aumentar la fiabilidad.

Si se solucionan todos los problemas detectados por estas herramientas, la máquina objetivo será significativamente menos vulnerable. No obstante, ten en cuenta que **cada herramienta genera una cantidad sustancial de tráfico de red** y puede consumir gran cantidad de recursos del objetivo. Más información.

Ten mucho cuidado si las lanzas sobre tu propio sistema en producción (NUNCA las uses contra sistemas externos sin autorización del propietario).

En esta sección vamos a hablar de dos de estas herramientas de carácter gratuito que puedes usar. **Si tu producto crece y necesitas algo más potente, algunas otras son**:

invicti Acunetix FORTRA Probely RAPID7



7.1.1. *Nessus*

Una de las herramientas gratuitas más usadas de este tipo, **permite el análisis en profundidad de un PC** realizando una auditoría detallada de sus servicios y cubriendo gran cantidad de vectores potenciales de problemas de seguridad (analiza configuraciones, parches, aplicaciones web, redes, sistemas, datos y aplicaciones...).

Requiere **4+ GB RAM** (solo la aplicación) y detecta más de 100.000 vulnerabilidades potenciales, teniendo además un gran nº de escaneos y funcionalidades relacionadas distintas (***Figura 63***). La versión gratuita (***Nessus Home Feed***) no detecta tantas vulnerabilidades como la de pago y está limitada a escanear **16 IP**. Tutorial.

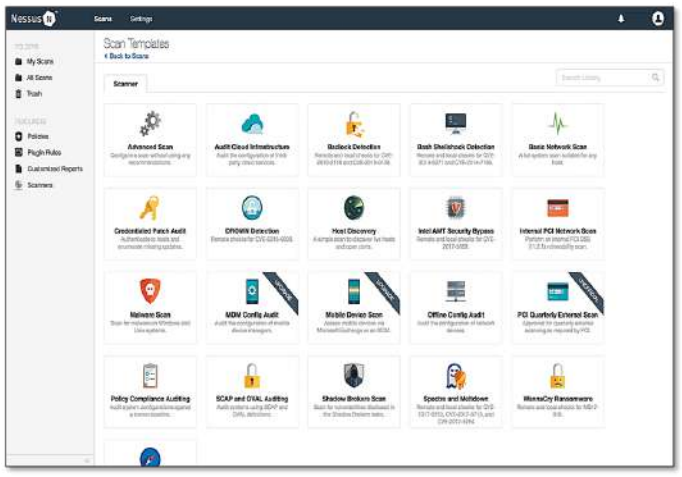


Figura 63. Pantalla de inicio de Nessus

7.1.2. OpenVAS

Es parte de la familia de productos ***Greenbone Security Manager*** (GSM) y permite análisis en detalle de servidores para comprobar más de 53.000 vulnerabilidades (***Figura 64***), actualizándose diariamente. También requiere 4+ Gb RAM solo para la aplicación. En estos enlaces podemos ver un **tutorial de uso**.

El resultado de estas herramientas en un **informe completo de vulnerabilidades encontradas**, una explicación de por qué son vulnerables, su gravedad y posibles soluciones o mitigaciones. La ***Figura 65*** muestra el aspecto de un informe de esta clase creado por ***OpenVAS***.

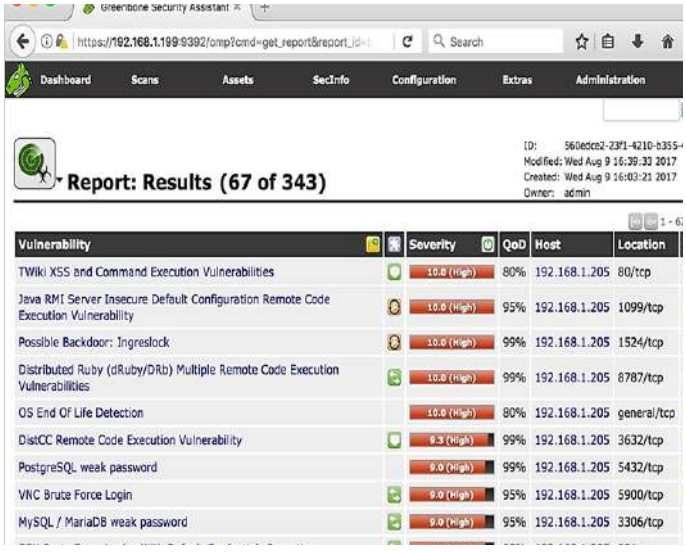


Figura 64. Resultado de un escaneo con OpenVAS

Medium (CVSS: 4.3)
NVT: jQuery < 1.9.0 XSS Vulnerability

Product detection result
`cpe:/a:jquery:jquery:1.7.1`
Detected by `jQuery Detection` (OID: 1.3.6.1.4.1.25623.1.0.141622)

Summary
jQuery before 1.9.0 is vulnerable to Cross-site Scripting (XSS) attacks. The jQuery(strInput) function does not differentiate selectors from HTML in a reliable fashion. In vulnerable versions, jQuery determined whether the input was HTML by looking for the character anywhere in the string, giving attackers more flexibility when attempting to construct a malicious payload. In fixed versions, jQuery only deems the input to be HTML if it explicitly starts with the '<' character, limiting exploitability only to attackers who can control the beginning of a string, which is far less common.

Vulnerability Detection Result
Installed version: `1.7.1` / Fixed version: `1.9.0`

Solution · Solution type
VendorFix / Update to version 1.9.0 or later.

Affected Software/OS
jQuery prior to version 1.9.0.

Vulnerability Detection Method
Checks if a vulnerable version is present on the target host.
Details: `jQuery 1.9.0 XSS Vulnerability`
OID.1.3.6.1.4.1.25623.1.0.141636
Version used: `$Revision: 12183 $`

Product Detection Result
Product: `cpe:/a:jquery:jquery: 1.7.1`
Method: `jQuery Detection`
OID: 1.3.6.1.4.1.25623.1.0.141622)

References
CVE: CVE-2012-6708
Other: URL: https://bugs.jquery.com/ticket/11290

Figura 65. Informe de resultados de OpenVAS

7.2. Protección en nube

7.2.1. El efecto positivo de contar con una CDN

Existe una serie de empresas, llamadas **CDNs (*Content Delivery Networks*)** que ofrecen una serie de servicios que, además de mejorar el tráfico que llega a nuestras webs, también permiten **contratar una serie de mecanismos de seguridad** que, si bien podemos instalar nosotros mismos, hacerlo bien es difícil si no tenemos los conocimientos técnicos adecuados.

Además de eso, protege nuestras webs **contra ataques de denegación de servicio**, donde alguien de forma aleatoria (o plenamente consciente como parte de un posible chantaje) envía una cantidad muy alta de tráfico a nuestra web con la intención de que sea incapaz de dar servicios a los clientes legítimos. En la imagen de la infraestructura que vimos anteriormente (***Figura 37***) apareció una de ellas, CloudFlare. Sobre **medidas de prevención de ataques DoS** el **INCIBE** tiene un artículo **[73]** que puede venirte muy bien si no estás familiarizado con estos conceptos.

Una de las cosas que muchos de estos servicios hacen por nosotros es **proteger nuestras webs con un tipo de herramienta denominada *Web Application Firewall* (WAF)**. Se trata de una herramienta que se pone "delante" de nuestras páginas web para interceptar el tráfico que le llega. Entonces se comporta como si fuera una especie de *antimalware* con el contenido de dicho tráfico: lo examina e identifica si dentro del mismo puede haber algo que se pueda considerar malicioso, o un ataque contra alguna de nuestras webs **parándolo en el acto si es así**.

En otras palabras, **esta herramienta es capaz de parar automáticamente ataques antes de que lleguen a nuestro *software*** y, por tanto, si es vulnerable a los mismos nunca recibirá ese tráfico malicioso y no sufrirá el ataque. Con esta explicación puedes hacerte una idea de la importancia que tiene contar con uno de ellos. La ***Figura 66*** muestra el esquema típico de funcionamiento, y hablaremos más de él aquí.

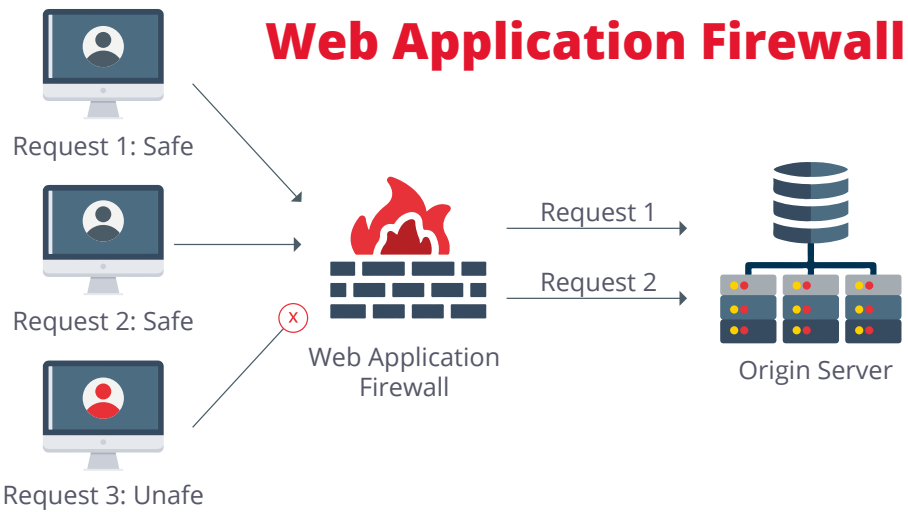
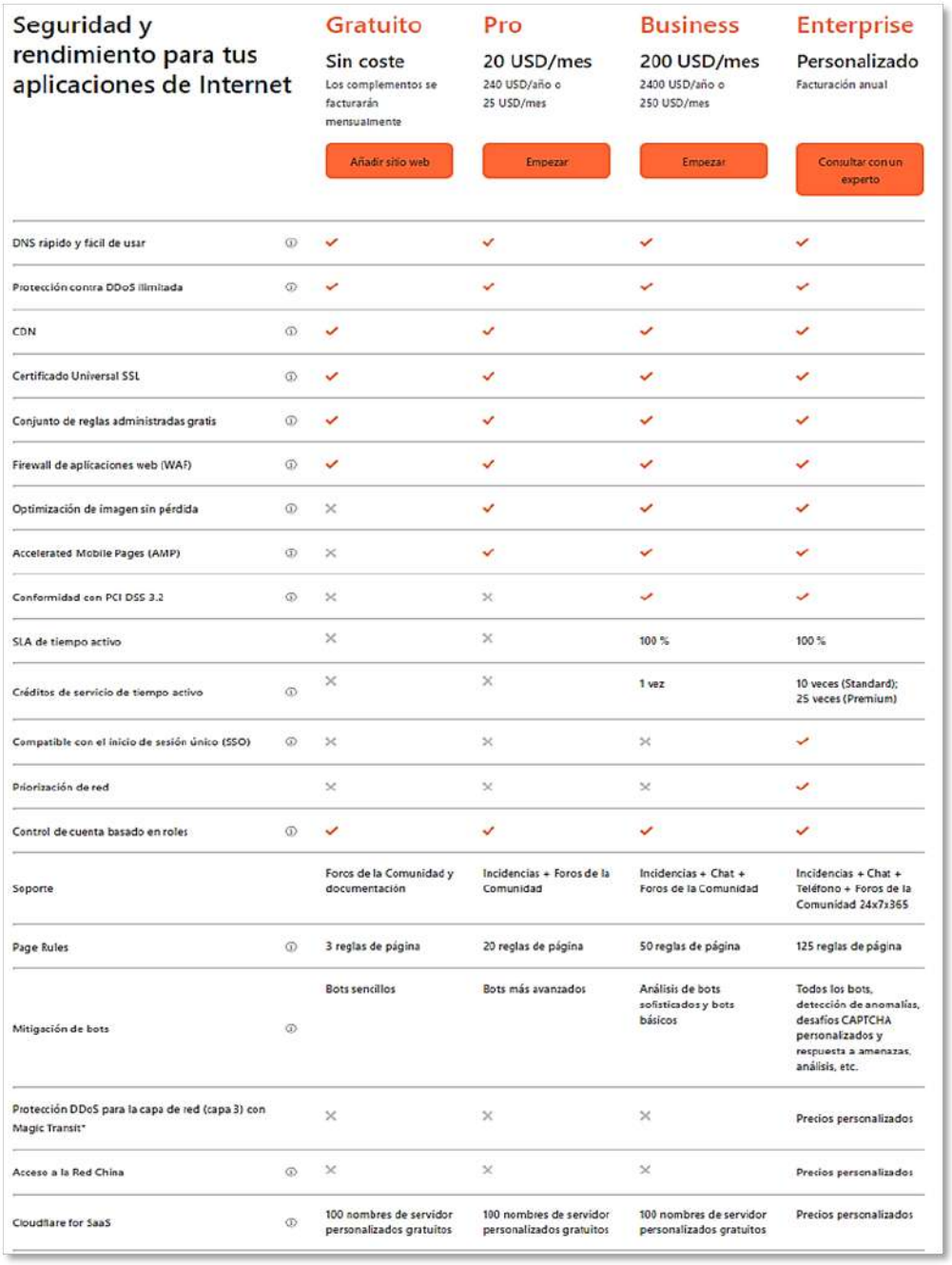


Figura 66. Esquema típico de un WAF

Al igual que las herramientas *antimalware*, un **WAF no es infalible**, pero sí que es un complemento imprescindible hoy en día en la web, ya que nos libra de muchos posibles problemas y dificulta a los atacantes entrar en nuestra web. CDNs como ***CloudFlare*** o ***TransparentEdge*** pueden ofrecer ese servicio a cambio de un precio mensual (que suele ser reducido para usos básicos) o incluso gratuito, y que es suficiente para dar más seguridad a webs que no aguanten un volumen de tráfico muy alto.

Contratar una protección (con WAF incluido) básica con *CloudFlare* es **gratuito** (***Figura 67***). Una protección tipo *Pro* para el contexto de una *start-up* también es algo muy asequible si el servicio gratuito se nos queda corto.



Seguridad y rendimiento para tus aplicaciones de Internet	Gratuito Sin coste Los complementos se facturarán mensualmente Añadir sitio web	Pro 20 USD/mes 240 USD/año o 25 USD/mes Empezar	Business 200 USD/mes 2400 USD/año o 250 USD/mes Empezar	Enterprise Personalizado Facturación anual Consultar con un experto
DNS rápido y fácil de usar	✓	✓	✓	✓
Protección contra DDoS ilimitada	✓	✓	✓	✓
CDN	✓	✓	✓	✓
Certificado Universal SSL	✓	✓	✓	✓
Conjunto de reglas administradas gratis	✓	✓	✓	✓
Firewall de aplicaciones web (WAF)	✓	✓	✓	✓
Optimización de imagen sin pérdida	×	✓	✓	✓
Accelerated Mobile Pages (AMP)	×	✓	✓	✓
Conformidad con PCI DSS 3.2	×	×	✓	✓
SLA de tiempo activo	×	×	100 %	100 %
Créditos de servicio de tiempo activo	×	×	1 vez	10 veces (Standard); 25 veces (Premium)
Compatible con el inicio de sesión único (SSO)	×	×	×	✓
Priorización de red	×	×	×	✓
Control de cuenta basado en roles	✓	✓	✓	✓
Soporte	Foros de la Comunidad y documentación	Incidencias + Foros de la Comunidad	Incidencias + Chat + Foros de la Comunidad	Incidencias + Chat + Teléfono + Foros de la Comunidad 24x7x365
Page Rules	3 reglas de página	20 reglas de página	50 reglas de página	125 reglas de página
Mitigación de bots	Bots sencillos	Bots más avanzados	Análisis de bots sofisticados y bots básicos	Todos los bots, detección de anomalías, desafíos CAPTCHA personalizados y respuesta a amenazas, análisis, etc.
Protección DDoS para la capa de red (capa 3) con Magic Transit*	×	×	×	Precios personalizados
Acceso a la Red China	×	×	×	Precios personalizados
Cloudflare for SaaS	100 nombres de servidor personalizados gratuitos	100 nombres de servidor personalizados gratuitos	100 nombres de servidor personalizados gratuitos	Precios personalizados

Figura 67. Tabla de precios y servicios de CloudFlare

7.2.2. *API Gateway*

Un *API Gateway* es un gestor que acepta todas las llamadas a un API y luego actúa como un *proxy* inverso (visto posteriormente), recuperando recursos de las aplicaciones del *back-end* de la aplicación en nombre de la aplicación cliente. Maneja tareas relacionadas con los servicios de API, como autenticación de usuarios, limitación de velocidad y monitorización.

> ***Un API Gateway es como una especie de "puerta de acceso", detrás de la cual están los servicios back-end a los que las aplicaciones del cliente quieren acceder. El cliente, también conocido como el consumidor de API, realiza su solicitud en esa "puerta", que verifica su identidad si se requiere para ello (normalmente sí en este contexto) y luego espera a que se entreguen los datos solicitados.***

Un ***API Gateway*** desacopla la interfaz del cliente de la implementación de *back-end* de la aplicación y hace que sea más fácil trabajar con el mismo, adaptándose muy bien a la infraestructura segura vista anteriormente. Además, **permite implementar las opciones de seguridad** que sean necesarias (autenticación, validación...) de forma centralizada (que va en consonancia con los requisitos vistos de gestión de peticiones para evitar saturación, por ejemplo). Esto permite luchar mejor contra los errores típicos a la hora de desarrollar un API **[74]**.

La ***Figura 68*** muestra la estructura de uno de estos elementos, en este caso proporcionado por *Amazon*. Si desplegamos en nube, puede ser interesante estudiar si nuestro proveedor nos proporciona un servicio de esta clase a un precio que podamos asumir en nuestra *start-up*.

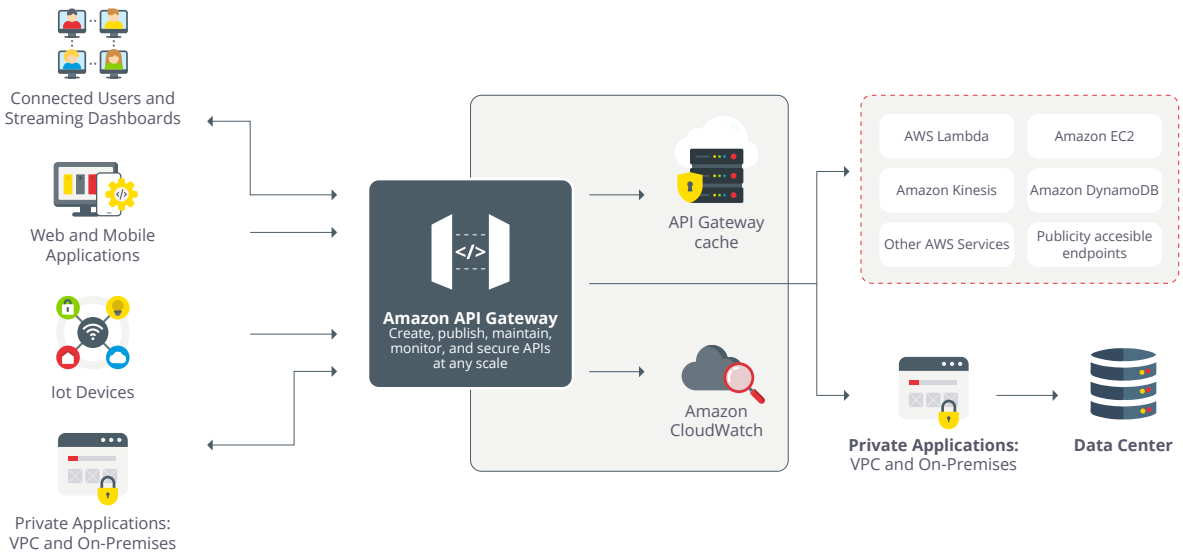


Figura 68. Estructura típica de un API Gateway

7.3. Protección *on-premises*

7.3.1. WAFs

Aunque ya hemos hablado de ellos como parte de los servicios que ofrecen CDNs como *CloudFlare*, este *software* se encuentra también en forma de **módulos** de los principales servidores web, que protegen automáticamente cualquier aplicación web que alojen (***Figura 69***) contra una amplia gama de ataques a aplicaciones. Su labor es muy importante, hasta el punto de que **se desaconseja desplegar la aplicación sin uno**, ya sea tuyo o bien contratado como vimos antes. Si no estás familiarizado con este concepto, el **INCIBE** tiene un artículo **[75]** que te explica que son.

Debemos tener en cuenta estas **consideraciones**:

- Incorpora una capa de protección adicional para cualquier aplicación web (*defense in depth*).
- `mod_security` es probablemente el WAF más conocido.
- Necesitan reglas adaptadas a cada aplicación; las más conocidas son CSR (*Core Security Rules*).
- Se puede instalar uno gratuitamente en nuestra infraestructura, o bien recurrir a los servicios de *CloudFlare* que hemos mencionado anteriormente.

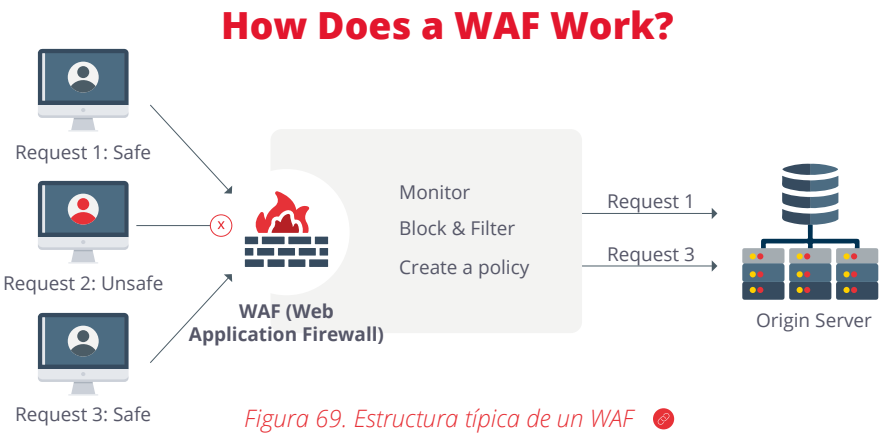


Figura 69. Estructura típica de un WAF

Una vez activo puede realizar las siguientes **operaciones** automáticamente, entre otras:

- **Protección HTTP:** Detecta infracciones de protocolo.
- **Listas negras de IPs en tiempo real:** Usando servidores de reputación IP de terceros.
- **Detectar *malware* web, *bots*, *crawlers*, escáneres:** API Google Safe Browsing y otras integraciones.
- **Detección DoS y otros ataques web comunes:** XSS, Inyección SQL, CSRF...
- **Identificación de datos confidenciales y prevenir el servir código fuente accidentalmente** (Ej. tarjetas de crédito).
- **Detecta y oculta los mensajes de error y enmascara la identidad del servidor.**

7.3.2. *Proxies* inversos

Un *proxy* inverso es un tipo de servidor *proxy* que recupera recursos en nombre de un cliente externo desde uno o más servidores internos. En una red, un *proxy* inverso básico se sitúa entre un grupo de servidores y los clientes que quieran utilizarlos, tal y como se mostró en la ***Figura 37***. Un cliente es cualquier *hardware* o *software* que pueda enviar solicitudes a un servidor.

El ***proxy* inverso** transmite todas las solicitudes de los clientes a los servidores y también entrega todas las respuestas y servicios procedentes de los servidores de vuelta a los clientes. Desde el punto de vista del cliente, **parece que todo procede del mismo lugar** (***Figura 70***).

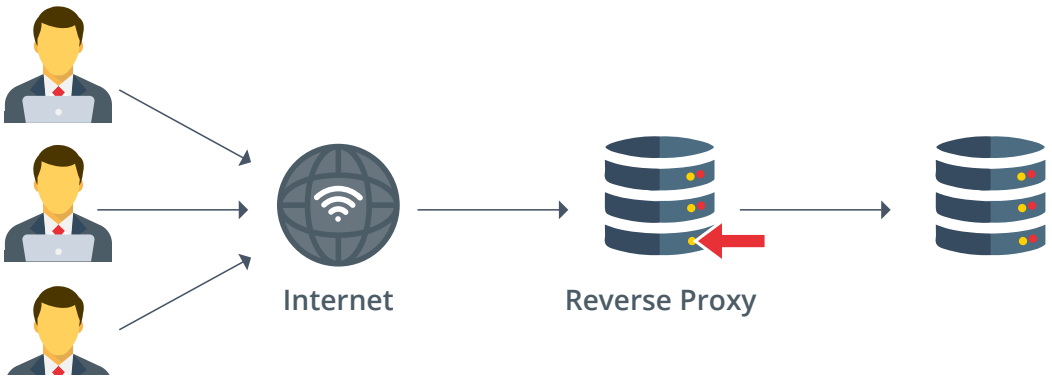


Figura 70. Esquema típico de un reverse proxy

Como vimos en la figura de la infraestructura, un *proxy* inverso se suele situar frente a un servidor web y recibir todas las solicitudes antes de que lleguen al servidor de origen. Suelen estar instalados en una **red privada o una zona desmilitarizada** (DMZ), estar protegidos por un *firewall*, e incorporar otras medidas de seguridad que protejan a todos los servicios que dependen de él, como el **WAF** de la sección anterior.

De esta manera, **representan la única conexión entre Internet y una red privada**, y es un punto muy propicio para centralizar todas las funciones de seguridad, como hemos dicho en los requisitos. Otros propósitos, como el **equilibrio de carga y la optimización** quedan un poco fuera del contexto de una *start-up* que estamos manejando, pero contar con uno puede ayudarnos a implementarlos en el futuro a medida que la empresa vaya creciendo y teniendo más recursos.

7.3.3. Vigilancia con productos gratuitos: *XDR/SIEM Wazuh*

Wazuh **es un *software* de tipo XDR / SIEM gratuito y de código abierto** que puede poner a tu alcance capacidades de prevención, detección y respuesta a amenazas como las que se mencionaron en la red de administración de la infraestructura de la ***Figura 37***.

Consiste en agentes de seguridad que se instalan en los endpoints de la infraestructura a monitorizar (máquinas virtuales, contenedores, on-premises, en nube...), y un servidor de gestión, que recopila y analiza los datos recopilados por esos agentes, lanzando alertas cuando detecta eventos de seguridad (***Figura 71***).

> ***Wazuh se integra con Elastic Stack para permitir búsquedas y visualizaciones avanzadas de los datos recopilados y las alertas lanzadas tras su análisis.***

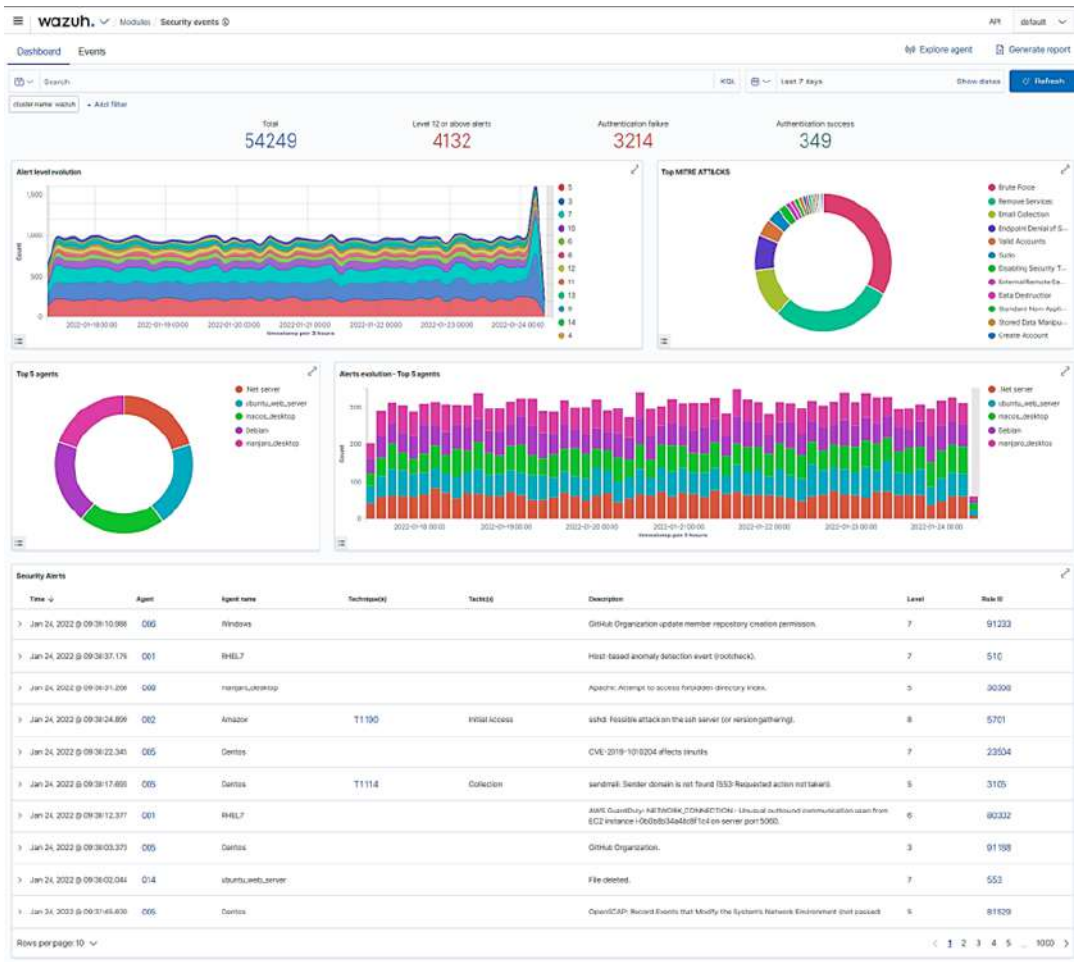


Figura 71. Incidentes detectados por Wazuh

Las **capacidades de *Wazuh*** son muchas, y enumeraremos las principales a continuación:

- **Detección de intrusos (IDS):** Los agentes escanean los sistemas monitoreados en busca de malware y anomalías sospechosas de distintos tipos.
- **Análisis de datos de los *logs*:** Los agentes leen los registros del SO y de las aplicaciones, y los reenvían de forma segura a un administrador central para su análisis y almacenamiento basados en reglas. Las reglas detectan errores de la aplicación o del sistema, configuraciones incorrectas, actividades maliciosas intentadas y/o exitosas, las violaciones de las políticas y otros problemas.

> ***En este escenario sí que puede merecer la pena hacer el esfuerzo de que los logs de nuestro software se escriban en un formato y localización compatibles con Wazuh, para aprovecharnos de sus capacidades de monitorización.***

- **Supervisión de la integridad de los archivos:** Supervisa el sistema de archivos de los *endpoints* monitorizados, identificando cambios en el contenido, permisos, propiedad y atributos de archivos que considera de interés.
- **Detección de vulnerabilidades:** Los agentes hacen inventario de *software* y envían esta información al servidor, donde se correlaciona con bases de datos de CVE para identificar vulnerabilidades conocidas (***Figura 72***), de la misma forma que ya hemos visto en esta guía.

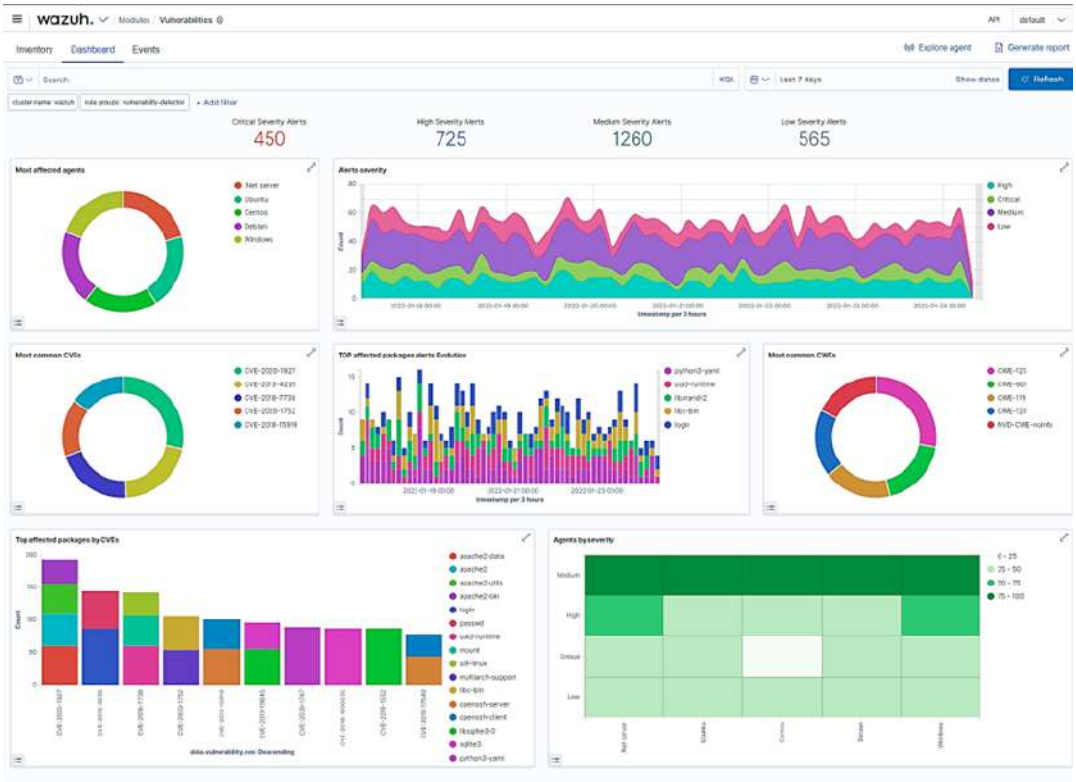


Figura 72. Vulnerabilidades detectadas por Wazuh

- **Evaluación de la configuración:** Supervisa la configuración de los *endpoints* monitorizados y sus aplicaciones para asegurarse de que cumplen con sus políticas de seguridad, estándares y/o guías de *hardening*.

> ***Los agentes hacen análisis periódicos para detectar aplicaciones que se sabe que son vulnerables, que no tienen parches o que están configuradas de forma insegura.***

- **Respuesta a incidentes:** Proporciona respuestas activas listas para usar que sirvan de contramedidas contra amenazas activas, como bloquear el acceso a un sistema desde las fuentes de amenazas cuando se cumplen ciertos criterios. También ayuda a la identificación de **IoC (Indicios de Compromiso)**.
- **Cumplimiento normativo:** Proporciona algunos de los controles de seguridad necesarios para cumplir con los estándares y regulaciones de la industria, para ayudar a nuestra *start-up* si quisiera cumplir con ellos a medida que fuera creciendo.
- **Seguridad en la nube:** Ayuda a monitorizar la infraestructura de la nube a nivel de API, utilizando módulos de integración que son capaces de extraer datos de seguridad de proveedores de nube como *Amazon AWS, Azure* o *Google Cloud*.

> ***Esto es de mucha utilidad si la infraestructura de nuestro software está total o parcialmente alojada en entornos de nube.***

- **Seguridad de contenedores:** Proporciona información de seguridad en los *endpoints* y contenedores ***Docker*** donde se aloja el *software*, supervisando su comportamiento y detectando amenazas, vulnerabilidades y anomalías.

Por tanto, podemos ver claramente como las capacidades de ***Wazuh*** podrían superar ampliamente el tiempo y recursos de los que disponemos inicialmente en nuestra start-up para desplegar un sistema de monitorización así. Pero, sin embargo, puede ser muy útil si se integra su uso de una forma creciente:

- **Simplificar su instalación sobre la infraestructura gracias a la automatización:** Existen proyectos asociados a ***Wazuh***, como ***Wazuh-Ansible***, que simplifican mucho su instalación, especialmente en escenarios de un solo nodo o pocos nodos, más afines a la situación inicial de una *start-up*. No obstante, para una instalación simple en una red de pocas máquinas, puede ser más rápido usar **wazuh-docker**.
- **Usar de manera creciente sus capacidades a medida que dispongamos de más recursos para exportarlas:** Se puede empezar por lo menos por la capacidad de detectar eventos de seguridad sobre las máquinas analizadas y de hacer escaneos de vulnerabilidades, para luego pasar a explotar más sus capacidades a medida que nuestra infraestructura y recursos vayan creciendo.

8. Referencias

[1] **INCIBE, «Privacidad, identidad digital y reputación online,» 2024. [En línea]. Available:** https://www.incibe.es/ciudadania/tematicas/privacidad

[2] **INCIBE, «Guía nacional de notificación y gestión de ciberincidentes,» 21 2 2020. [En línea]. Available:** https://www.incibe.es/sites/default/files/contenidos/guias/doc/guia_nacional_notificacion_gestion_ciberincidentes.pdf

[3] **INCIBE, «Validación inadecuada de los datos de entrada en productos de Festo Didactic SE,» 18 10 2023. [En línea]. Available:** https://www.incibe.es/incibe-cert/alerta-temprana/avisos-sci/validacion-inadecuada-de-los-datos-de-entrada-en-productos-de-festo-didactic-se

[4] **OWASP, «Input Validation Cheat Sheet,» 25 10 2023. [En línea]. Available:** https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

[5] **OWASP, «OWASP Validation Regex Repository,» 25 10 2023. [En línea]. Available:** https://owasp.org/www-community/OWASP_Validation_Regex_Repository

[6] **Microsoft Learn, «Prevención de ataques de redireccionamiento abierto en ASP.NET Core,» 8 7 2023. [En línea]. Available:** https://learn.microsoft.com/es-es/aspnet/core/security/preventing-open-redirects?view=aspnetcore-7.0.

[7] **«Rust,» 25 10 2023. [En línea]. Available:** https://www.rust-lang.org/

[8] **J. Pastor, «La Casa Blanca no quiere C ni C++, pero hay un problema: estos lenguajes son el pilar de SSOO como Windows, Linux o macOS,» 1 3 2024. [En línea]. Available:** https://www.xataka.com/seguridad/casa-blanca-no-quiere-c-c-hay-problema-estos-lenguajes-pilar-ssoo-como-windows-linux-macos

[9] **INCIBE, «Trojan Source: vulnerabilidades en Unicode que afectan a compiladores,» 11 4 2021. [En línea]. Available:** https://www.incibe.es/incibe-cert/alerta-temprana/avisos/trojan-source-vulnerabilidades-unicode-afectan-compiladores

[10] **PortSwigger, «Burp Proxy,» 25 10 2023. [En línea]. Available:** https://portswigger.net/burp/documentation/desktop/tools/proxy

[11] **K. Abuhakmeh, «Server-side validation, client-side feel,» 27 4 2022. [En línea]. Available:** https://www.jetbrains.com/dotnet/guide/tutorials/htmx-aspnetcore/serverside-validation-clientside-feel/

[12] **Let's Encrypt, «Let's Encrypt es una Autoridad de Certificación gratuita, automatizada, y abierta,» 26 10 2023. [En línea]. Available:** https://letsencrypt.org/es/

[13] **Mozilla, «SSL Configuration Generator,» 26 10 2023. [En línea]. Available:** https://ssl-config.mozilla.org/

[14] **Qualys SSL Labs, «SSL Server Test,» 26 10 2023. [En línea]. Available:** https://www.ssllabs.com/ssltest/

[15] **Seguridad y Firewall , «Un nuevo error de PHP Composer podría permitir ataques generalizados de la cadena de suministro,» 26 10 2023. [En línea]. Available:** https://www.seguridadyfirewall.cl/2021/04/un-nuevo-error-de-php-composer-podria.html?m=1

[16] **INCIBE, «Alerta temprana. Avisos,» 16 4 2024. [En línea]. Available:** https://www.incibe.es/incibe-cert/alerta-temprana/avisos

[17] **INCIBE, «Inyección de código malicioso en la librería XZ Utils,» 1 4 2024. [En línea]. Available:** https://www.incibe.es/incibe-cert/alerta-temprana/avisos/inyeccion-de-codigo-malicioso-en-la-libreria-xz-utils

[18] **Refactoring Guru, «Patrones de diseño,» 28 10 2023. [En línea]. Available:** https://refactoring.guru/es/design-patterns

[19] **Refactoring Guru, «Refactoring,» 26 10 2023. [En línea]. Available:** https://refactoring.guru/es/refactoring

[20] **Grupo Ático 34, «Política de cookies: texto de cookies, plantilla y ejemplos,» 29 10 2023. [En línea]. Available:** https://protecciondatos-lopd.com/empresas/politica-de-cookies/

[21] **K. M. Stine, R. L. Kissel, W. C. Barker, A. Lee, J. Fahlsing y J. Gulick, *«Guide for Mapping Types of Information and Information Systems to Security Categories»* 2008**

[22] **Gobierno de España, «Real Decreto 311/2022, de 3 de mayo, por el que se regula el Esquema Nacional de Seguridad,» 4 5 2022. [En línea]. Available:** https://www.boe.es/buscar/act.php?id=BOE-A-2022-7191

[23] **Consejo de la Unión Europea, «Protección de datos en la UE,» 26 10 2023. [En línea]. Available:** https://www.consilium.europa.eu/es/policies/data-protection/

[24] **INCIBE, «Medidas para proteger la información: defiende el principal activo de tu empresa,» 21 11 2023. [En línea]. Available:** https://www.incibe.es/empresas/blog/medidas-para-proteger-la-informacion-defiende-el-principal-activo-de-tu-empresa

[25] **INCIBE, «Política de Protección de Datos Personales,» 4 2023. [En línea]. Available:** https://www.incibe.es/incibe/proteccion-datos-personales

[26] **OWASP, «SQL Injection,» 26 10 2023. [En línea]. Available:** https://owasp.org/www-community/attacks/SQL_Injection

[27] **J. M. Roviralta Puente, «Ataques de inyección SQL, una amenaza para tu web,» INCIBE, 26 10 2021. [En línea]. Available:** https://www.incibe.es/empresas/blog/ataques-inyeccion-sql-amenaza-tu-web

[28] **OWASP, «SQL Injection Prevention Cheat Sheet,» 26 10 2023. [En línea]. Available:** https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[29] **R. Gilbert, «Information exposure through query strings in url,» 26 10 2023. [En línea]. Available:** https://owasp.org/www-community/vulnerabilities/Information_exposure_through_query_strings_in_url

[30] **OWASP, «Cross Site Scripting (XSS),» 27 10 2023. [En línea]. Available:** https://owasp.org/www-community/attacks/xss/

[31] **OWASP, «OWASP Secure Headers Project,» 27 10 2023. [En línea]. Available:** https://owasp.org/www-project-secure-headers/index.html#configuration-proposal

[32] **OWASP, «HTTP Security Response Headers Cheat Sheet,» 27 10 2023. [En línea]. Available:** https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Headers_Cheat_Sheet.html

[33] **OWASP, «Content Security Policy,» 27 10 2023. [En línea]. Available:** https://owasp.org/www-community/controls/Content_Security_Policy

[34] **OWASP, «Content Security Policy Cheat Sheet,» 27 10 2023. [En línea]. Available:** https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html

[35] **Mozilla Developer Network, «Permissions-Policy,» 27 10 2023. [En línea]. Available:** https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Permissions-Policy

[36] **OWASP, «HTML5 Security Cheat Sheet,» 27 10 2023. [En línea]. Available:** https://cheatsheetseries.owasp.org/cheatsheets/HTML5_Security_Cheat_Sheet.html

[37] **K. Koishigawa, «Everything You Need to Know About Cookies for Web Development,» 3 2 2021. [En línea]. Available:** https://www.freecodecamp.org/news/everything-you-need-to-know-about-cookies-for-web-development/

[38] **Invicti, «Security Cookies: White Paper,» 2019**

[39] **INCIBE, «Qué son las cookies y cómo mostrarlas en un sitio web,» 20 7 2021. [En línea]. Available:** https://www.incibe.es/empresas/blog/son-las-cookies-y-mostrarlas-sitio-web

[40] **INCIBE, «Política de cookies,» 2024. [En línea]. Available:** https://www.incibe.es/incibe/politica-cookies

[41] **OWASP, «OWASP Code Review Guide,» 7 2017. [En línea]. Available:** https://owasp.org/www-project-code-review-guide/

[42] **INCIBE, «Gestores de contraseñas: ¿cómo funcionan?,» 27 1 2021. [En línea]. Available:** https://www.incibe.es/ciudadania/blog/gestores-de-contrasenas-como-funcionan

[43] **INCIBE, «Gestores de contraseñas,» 2024. [En línea]. Available:** https://www.incibe.es/node/499093

[44] **INCIBE, «Gestión de contraseñas seguras,» 2024. [En línea]. Available:** https://www.incibe.es/ciudadania/tematicas/contrasenas-seguras

[45] **INCIBE, «El factor de autenticación doble y múltiple,» 27 2 2019. [En línea]. Available:** https://www.incibe.es/ciudadania/blog/el-factor-de-autenticacion-doble-y-multiple

[46] **INCIBE, «Mecanismos básicos de control de acceso,» 27 11 2014. [En línea]. Available:** https://www.incibe.es/incibe-cert/blog/control-acceso

[47] **CloudFlare, «¿Qué es el control de acceso basado en roles (RBAC)?,» 27 10 2023. [En línea]. Available:** https://www.cloudflare.com/es-es/learning/access-management/role-based-access-control-rbac/

[48] **OWASP, «OWASP Software Component Verification Standard,» 27 10 2023. [En línea]. Available:** https://owasp.org/www-project-software-component-verification-standard/

[49] **OWASP, «File Upload Cheat Sheet,» 27 10 2023. [En línea]. Available:** https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html

[50] **INCIBE, «Políticas de seguridad para la pyme,» 2024. [En línea]. Available:** https://www.incibe.es/empresas/herramientas/politicas

[51] **C. Chipeta, «What is Data Leak Detection Software?,» 9 8 2023. [En línea]. Available:** https://www.upguard.com/blog/what-is-data-leak-detection-software

[52] **Have I Been Pwnd?, «Have I Been Pwnd?,» 28 10 2023. [En línea]. Available:** https://haveibeenpwned.com/

[53] **Flare, «Preventing and Detecting Data Leaks: The Complete Guide,» 28 4 2023. [En línea]. Available:** https://flare.io/learn/resources/blog/data-leakage-prevention/

[54] **OWASP, «Authorization Cheat Sheet,» 28 10 2023. [En línea]. Available:** https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html

[55] **OWASP, «Cross Site Request Forgery (CSRF),» 28 10 2023. [En línea]. Available:** https://owasp.org/www-community/attacks/csrf

[56] **Google, «reCAPTCHA protects your website from fraud and abuse without creating friction,» 28 10 2023. [En línea]. Available:** https://www.google.com/recaptcha/about/

[57] **T. Serafim, «JavaScript obfuscator,» 28 10 2023. [En línea]. Available:** https://github.com/javascript-obfuscator/javascript-obfuscator

[58] **INCIBE, «https://www.incibe.es/incibe/jornadas-incibe-espacios-ciberseguridad/estudiantes/programa-programacion-segura-web,» 2024. [En línea]. Available:** https://www.incibe.es/incibe/jornadas-incibe-espacios-ciberseguridad/estudiantes/programa-programacion-segura-web

[59] **M. Howard y D. Leblanc, Writing Secure Code, Second Edition, Microsoft Press, 2003.**

[60] **ipsec.pl, «Input validation of free-form Unicode text in Python,» 13 6 2017. [En línea]. Available:** https://ipsec.pl/input-validation-of-free-form-unicode-text-in-python.html

[61] **OWASP, «Cross Site Scripting Prevention Cheat Sheet,» 28 10 2023. [En línea]. Available:** https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.md

[62] **OWASP, «OWASP Java Encoder,» 28 10 2023. [En línea]. Available:** https://owasp.org/www-project-java-encoder/

[63] **OWASP, «Java HTML Sanitizer,» 28 10 2023. [En línea]. Available:** https://github.com/owasp/java-html-sanitizer

[64] **OWASP, «Session Management Cheat Sheet,» 28 10 2023. [En línea]. Available:** https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

[65] **J. M. Redondo, «Secure Coding Checklist Template,» 10 2021. [En línea]. Available:** https://www.researchgate.net/publication/355167455_Secure_Coding_Checklist_Template

[66] **Cloud Security Alliance, «Top Threats to Cloud Computing: Egregious Eleven,» 8 9 2019. [En línea]. Available:** https://cloudsecurityalliance.org/press-releases/2019/08/09/csa-releases-new-research-top-threats-to-cloud-computing-egregious-eleven/

[67] **deceptive.design, «Types of deceptive pattern,» 28 10 2023. [En línea]. Available:** https://www.deceptive.design/types

[68] **INCIBE, «OWASP publica el Top 10 - 2021 de riesgos de seguridad en aplicaciones web,» 15 9 2021. [En línea]. Available:** https://www.incibe.es/incibe-cert/publicaciones/bitacora-de-seguridad/owasp-publica-el-top-10-2021-riesgos-seguridad-aplicaciones

[69] **Sonar, «clean code for teams and enterprises with {SonarQube},» 28 10 2023. [En línea]. Available:** https://www.sonarsource.com/products/sonarqube/

[70] **Zap Dev Team, «Zed Attack Proxy,» 28 10 2023. [En línea]. Available:** https://www.zaproxy.org/

[71] **INCIBE, «Uso de OWASP-ZAP,» 2024. [En línea]. Available:** https://www.incibe.es/incibe-cert/seminarios-web/uso-owasp-zap

[72] **INCIBE, « Programa: Fundamentos del análisis de sitios web,» 2024. [En línea]. Available:** https://www.incibe.es/incibe/jornadas-incibe-espacios-ciberseguridad/estudiantes/programa-fundamentos-analisis-web

[73] **A. Fernandez Castrillo, «Medidas de protección frente ataques de denegación de servicio (DoS),» INCIBE, 26 1 2018. [En línea]. Available:** https://www.incibe.es/incibe-cert/blog/medidas-proteccion-frente-ataques-denegacion-servicio-dos

[74] **OWASP, «OWASP Top 10 API Security Risks – 2023,» 28 10 2023. [En línea]. Available:** https://owasp.org/API-Security/editions/2023/en/0x11-t10/

[75] **INCIBE, «WAF: cortafuegos que evitan incendios en tu web,» 7 6 2018. [En línea]. Available:** https://www.incibe.es/empresas/blog/waf-cortafuegos-evitan-incendios-tu-web

Financiado por la Unión Europea
NextGenerationEU

GOBIERNO DE ESPAÑA
MINISTERIO PARA LA TRANSFORMACIÓN DIGITAL Y DE LA FUNCIÓN PÚBLICA
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL

Plan de Recuperación, Transformación y Resiliencia

incibe_
INSTITUTO NACIONAL DE CIBERSEGURIDAD

incibe emprende