Tabla 9. Descripciones textuales por fuente seleccionada	
ID de la fuente	Descripción textual
FS.1	Esta fuente solo hace mención de dos mecanismos de seguridad de ASP.NET Core, Autenticación y Autorización, el primero lo define como un proceso en el cual un usuario proporciona credenciales que son comparadas con las almacenadas en una base de datos, aplicación o recurso; para este mecanismo el autor no detalla escenarios, retos ni amenazas o vulnerabilidades asociadas a este, sin embargo, propone como una buena práctica la gestión de identidades, ya que estas son una forma segura de autenticarse en los servicios sin tener que poner las credenciales en código o variables de entorno, asimismo, también menciona que nunca se deben almacenar datos confidenciales en el código del proveedor o archivos de configuración de texto sin formato, en cambio se puede hacer uso de la herramienta de *Secret Manager*, otra cosa que menciona es no utilizar secretos de producción en entorno de desarrollo o pruebas, que se deben de especificar fuera del proyecto para así evitar que se envíen por error a un repositorio de código fuente. Por otro lado, describe que una mala práctica para la Autenticación es la concesión de credenciales de contraseña del propietario del recurso, ya que esta expone la contraseña del usuario al cliente, siendo esto un riesgo de seguridad importante y que solo debe utilizarse cuando no sea posible utilizar otros flujos de Autenticación. Finalmente, sobre la Autorización, menciona que es el proceso que continua después de la Autenticación, pues la autorización la define como el proceso que determina lo que un usuario está autorizado a hacer. Sin embargo, respecto a escenarios, amenazas y/o vulnerabilidades, buenas y malas prácticas, así como retos, no menciona nada para este mecanismo.
FS.2	En esta fuente, los autores solo hablan sobre la Autenticación, no lo describen, sin embargo, mencionan que una buena práctica para este mecanismo es el uso de *Secret Manager* la cual almacena datos sensibles durante el desarrollo, los cuales se almacenan en una ubicación separada del árbol de proyectos, y estos no se registran en el control del codigo fuente, ya que oculta los detalles de implementación, como dónde y cómo se almacenan los valores, asimismo, se menciona un escenario para el uso de secretos, donde una cadena de conexión de base de datos almacenada en *appsettings.json* no debe incluir la contraseña, si no que se almacene la contraseña como un secreto y se incluya en la cadena de conexión en tiempo de ejecución. Por otro lado, mencionan que una mala práctica es almacenar datos sensibles en codigo fuente o archivos de configuración, asi como que los secretos de producción no deben utilizarse para desarrollo o pruebas, ni deben desplegarse con la aplicación, tambien mencionan que no se debe escribir codigo que dependa de la ubicación o el formato de los datos guardados en *Secret Manager*, ya que dichos detalles de implementación pueden cambiar. Respecto a retos y amenazas o vulnerabilidades asociados al mecanismo, los autores no mencionan nada.
FS.3	Esta fuente aborda dos mecanismos de seguridad Autorización, la cual la define como el proceso que verifica si un usuario autenticado tiene los permisos necesarios para acceder a ciertos recursos o realizar operaciones específicas. Este mecanismo incluye herramientas como el atributo [*Authorize*], que restringe el acceso a usuarios autenticados, y las políticas de autorización, que permiten reglas detalladas para autorizar acciones específicas. El autor menciona como escenario a las aplicaciones financieras. Se menciona que las amenazas asociadas son *Cross-Site Scripting* (XSS) y *Cross-Site Request Forgery* (CSRF). Respecto a las buenas prácticas, se mencionan la implementación de protección contra CSRF mediante tokens anti-falsificación, Por otro lado, la Autenticación verifica la identidad del usuario mediante credenciales como nombre de usuario y contraseña, asignándole una identidad que se utiliza en posteriores comprobaciones de autorización. Se describen mecanismos como la autenticación basada en cookies, la autenticación basada en claims, que permite asociar información adicional (*claims*). También se menciona el uso de JSON Web Tokens (*JWT*) en la protección de *APIs*. El autor menciona escenarios como las aplicaciones de comercio electrónico y redes sociales, y que las amenazas asociadas son ataques de fuerza bruta, gestión de sesiones, almacenamiento de contraseñas y Cross-Site Scripting (XSS). Como buenas prácticas para este mecanismo son: aplicar la autenticación de dos factores, aplicar políticas de bloqueo de cuentas y limitación de velocidad, utilizar técnicas seguras de gestión de sesiones, almacenar contraseñas con algoritmos hash y *salting*. Para *Razor* Pages: aplicar el atributo [*Authorize*], usar reclamos o roles, usar *AuthorizeView*, proteger paginas mediante la comprobación de reclamaciones específicas. Buenas prácticas que aplican para ambos mecanismos: codificar entradas y validar datos antes de renderizarlos; realizar seguimiento de los fallos. Una mala práctica que se menciona para ambos mecanismos es las exposiciones de información sensible en los mensajes de error. Para retos no se menciona nada.
FS.4	En esta fuente, el autor solo habla sobre el mecanismo de Autorización, no da una descripcion sobre el mecanismo, pero menciona escenarios donde se utiliza dicho mecanismo, como los paneles de

	administración en los que solo los administradores tienen acceso total, servicios basados en suscripciones con distintos niveles de acceso, aplicaciones multiusuario en las que los usuarios de diferentes organizaciones tienen permisos distintos. El autor no habla sobre amenazas o vulnerabilidades asociadas al mecanismo, pero menciona un conjunto de buenas prácticas, como el utilizar políticas en lugar de roles directamente, ya que estas ofrecen mayor flexibilidad y hacen que la lógica de autorización sea más facil de mantener; asegurar los puntos finales de la API con autenticación y autorización, combinar la autorización basada en roles con mecanismos de autenticación adecuados como *JWT* para garantizar que la API este protegida contra accesos no autorizados; mantener separados los puntos finales sensibles, no mezclar acciones administrativas sensibles con acciones generales; otra practica que menciona es implementar registros y auditorias, que es importante revisar regularmente para no haya intentos de acceso no autorizado; por último, menciona que se actualicen periódicamente las funciones y los permisos, ya que a medida que las aplicaciones crecen, puede ser necesario actualizar las funciones de los usuarios y es importante tener un proceso para modificar las funciones y los permisos. El autor no menciona nada sobre mallas practicas o retos asociados al mecanismo.
FS.5	En esta fuente se habla de los mecanismos *Data Protection*, Autenticación y token anti-CSRF, el autor solo describe el primer mecanismo, el cual simplifica el cifrado para proteger datos confidenciales, gestiona las claves de cifrado de forma segura y maneja automáticamente la rotación de claves, de este mismo mecanismo se mencionan buenas prácticas como lo es cifrar los datos confidenciales tanto en reposo como en tránsito, asi como rotar periódicamente las claves de cifrado para mitigar riesgos de seguridad. Respecto a la autenticación, solo se habla sobre buenas prácticas las cuales son, aplicar políticas de contraseñas seguras, utilizar cookies exclusivas de HTTP para almacenar tokens de sesión, para que no se pueda acceder a través de JavaScript, asi como implementar tokens de sesión de corta duración con políticas de expiración seguras. Tambien se mencionan buenas prácticas para el uso de *JWT*, el autor habla sobre utilizar algoritmos de firma fuertes como lo es RS256 para proteger los tokens, tambien el establecer tiempos de expiración de tokens para minimizar el riesgo de robo de tokens y utilizar HTTPS para que las comunicaciones estén cifradas y evitar ataques de intermediario. Finalmente, sobre el mecanismo del token anti-CSRF, el autor no lo describe, sin embargo, si habla sobre la amenaza o vulnerabilidad asociada a este, el cual es la falsificación de solicitudes entre sitios (CSRF por sus siglas en ingles), el autor describe que esta vulnerabilidad explota la confianza que un sitio web tiene en el navegador d un usuario, engañando a los usuarios autenticados a que realicen solicitudes no deseadas, como cambiar contraseñas o realizar compras. No se menciona información sobre malas prácticas y retos de uso de los mecanismos.
FS.6	Esta fuente habla sobre el token anti-CSRF, los autores no proporcionan una descripcion como tal del mecanismo, sin embargo, si describen la amenaza/vulnerabilidad que tiene asociada, la cual es la falsificación de solicitudes entre sitios (CSRF por sus siglas en ingles), y lo definen como un ataque contra aplicaciones alojadas en la web, donde una aplicación web maliciosa puede influir en la interacción entre un navegador cliente y una aplicación web que confía en ese navegador, este ataque es posible debido a los tokens de autenticación que se envían automáticamente con cada solicitud a un sitio web, este ataque se aprovecha de la sesión previamente autenticada del usuario. Los autores describen el siguiente escenario donde sucede este ataque, donde un usuario se autentica en un sitio seguro y obtiene su cookie de autenticación valida, dado que dicho proceso fue correcto, pero le sale una alerta (que dice que has ganado un premio) que proviene de un sitio malicioso, el usuario da clic en dicha alerta y el sitio malicioso ahora es capaz de enviar peticiones en nombre del usuario autenticado y podrá realizar cualquier acción que dicho usuario este autorizado a hacer, sin embargo, tambien mencionan que no siempre se necesita de interacción por parte del usuario, que a veces solo con visitar el sitio malicioso es posible ejecutar scripts, enviar formularios con petición AJAX, ocultar formularios mediante CSS. Los autores mencionan buenas prácticas, las cuales son añadir explícitamente el token anti-falsificación a elementos <*form*> sin usar etiquetas, asi mismo, recomiendan el uso de *AutoValidateAntiforgery* Token para escenarios no API, ya que esto asegura que las peticiones POST estén protegidas por defecto y finalmente que los tokens deben ser refrescados despues de que el usuario es autenticado. En la fuente no se habla sobre malas prácticas ni retos en el uso del mecanismo.
FS.7	En esta fuente se habla sobre el mecanismo de protección de secretos, aunque el autor no proporciona una descripción o escenario del mecanismo, si menciona que la amenaza/vulnerabilidad asociada es el acceso no autorizado, y menciona una serie de buenas prácticas: realizar auditoria para identificar los secretos (para identificar la información que se va a proteger), evitar el *hardcoding* de los secretos (mantener los secretos fuera del código fuente), utilizar almacenes de claves seguros (garantiza que estén seguros y cifrados), implementar herramientas de escaneado de secretos (ayuda a evitar la exposición accidental), aprovechar las identidades gestionadas (apoyo de Azure para reducir la necesidad de

	gestionar los secretos manualmente), aplicar un control de acceso granular (seguir el principio del mínimo privilegio), rotar los secretos con regularidad (la rotación periódica reduce el riesgo de acceso no autorizado), supervisar y registra los accesos (realizar un seguimiento del acceso y el uso), implementar el aislamiento de red (configurar cortafuegos y grupos de seguridad de red para restringir el acceso), cifrar los secretos en reposo y en tránsito (esto proporciona una capa adicional de seguridad), tener una distribución segura de secretos (asegurarse de que se comparten de forma segura dentro y fuera de la organización, asi como tener procedimientos de recuperación de secretos). Por otro lado, el autor menciona que una mala práctica es incrustar secretos directamente en el código o archivos de configuración, ya que es un riesgo importante para la seguridad, porque si el código base se ve comprometido, los secretos también lo estarán. En la fuente no se habla nada sobre los retos asociados al uso de este mecanismo.
FS.8	En esta fuente se habla únicamente sobre el mecanismo de autenticación, de forma específica sobre el *JWT*, el autor menciona que estos se utilizan a menudo para autenticar a los usuarios después de que hayan proporcionado credenciales validas, donde el servidor verifica el *JWT* que se envía en cada solicitud desde el cliente para asegurarse de que el usuario es quien dice ser. En la fuente se habla de que a este mecanismo tiene asociada la amenaza/vulnerabilidad de ataques de escucha o de intermediario (MITM), aunque no menciona una descripcion de estas amenazas ni habla sobre escenarios, menciona buenas prácticas, menciona que cuando se utilicé *JWTs* hay que usar algoritmos criptográficos fuertes como HMAC , SHA256; se tiene que establecer el tiempo de caducidad de los tokens ya que estos no pueden ser validos indefinidamente, asi como debe de haber una opción para revocar los tokens si un usuario ya no está autorizado a acceder a la aplicación. Otra buena práctica que menciona es validar siempre la firma y la integridad del token antes de procesar las solicitudes; la transmisión segura del token es crítica para protegerlo contra la intercepción y finalmente recomienda cifrar los *JWT* si se transmite información sensible, ya que esto ayudaría a protegerse contra usuarios malintencionados que manipulen el token o recuperen datos de este. El autor no habla sobre malas prácticas o restos que plantea el uso del mecanismo.
FS.9	Esta fuente habla sobre *Secret Management*, el autor menciona que es un mecanismo para almacenar datos confidenciales fuera del código fuente del proyecto, evitando que los datos se compartan accidentalmente, así mismo menciona que es muy útil durante el desarrollo ya que permite a los desarrolladores mantener información confidencial fuera de su código base y archivos de configuración. Respecto a las preguntas de escenarios, amenazas/vulnerabilidad y buenas prácticas no las menciona el autor, sin embargo, menciona que una mala práctica es el *hardcoding* de información sensible directamente en el código fuente, el almacenar datos sensibles en el *appsettings.json* sin el cifrado o los controles de acceso adecuados, el pasar por alto la necesidad de canales de comunicación seguros transmitiendo datos sensibles a través de una conexión no cifrada. Otra mala práctica que menciona es el uso de las mismas credenciales en distintos entornos (desarrollo, pruebas y producción) ya que aumenta el riesgo de exposición y uso indebido, y finalmente el hecho de no aplicar controles de acceso y pistas de auditoria adecuados ya que no hacerlo complica los esfuerzos de supervisión y respuesta en materia de seguridad. Por otro lado, se hace mención sobre un reto en el uso del mecanismo, el cual es la edición de secretos de usuario en Visual Studio, a pesar de que este proporciona un enfoque integrado y fácil de usar para gestionar los secretos de usuario, requiere una configuración adicional y puede tener problemas de compatibilidad con versiones de .NET Core más antiguas o no compatibles.
FS.10	Esta fuente habla sobre tres mecanismos, el primero es el token *AntiForgery*, el autor describe que este garantiza que las solicitudes proceden de una aplicación original y no de sitios de terceros ya que, si falta un token o no coincide con el valor almacenado, el servidor rechazara la solicitud como maliciosa. Se menciona que la amenaza/vulnerabilidad asociada a este mecanismo es la falsificación de petición de sitio cruzado (CSRF), el cual se menciona que es donde un atacante engaña al usuario para que ejecute acciones no deseadas en una aplicación web, mientras esta autenticado. No menciona más información sobre este mecanismo. El segundo mecanismo del que habla es HTML *encoding*, el cual se menciona que codifica los caracteres potencialmente inseguros del contenido generado por el usuario antes de renderizarlo en la página. La fuente habla que está asociado con la vulnerabilidad de Cross-site scripting (XSS) el cual donde un atacante inyecta scripts maliciosos en sitios web de confianza con el objetivo de robar información sensible; el autor menciona las siguientes buenas prácticas: validación de entradas (especialmente las que se renderizan como HTML), crear políticas de seguridad de contenidos (CSP) para restringir al navegador la ejecución de scripts que no cumplan dicha política, y la validación de solicitudes (activar la función integrada en ASP.NET para bloquear cualquier solicitud potencialmente insegura). Para escenarios, malas prácticas y retos, no menciona nada. El tercer mecanismo es CORS, el autor lo define como una función de seguridad que permite a las aplicaciones web controlar que dominios externos

	pueden acceder a sus recursos, CORS ofrece un mecanismo para relajar las restricciones de origen cruzado; se menciona que está asociado con la amenaza/vulnerabilidad del acceso no autorizado a datos sensibles. Respecto a escenarios, buenas y malas prácticas, asi como retos, no se menciona nada en la fuente para este mecanismo.
FS.11	En esta fuente hablan únicamente de dos mecanismos de seguridad, la autenticación y autorización. Menciona más en detalle acerca de la autenticación la cual el autor lo define como el proceso de determinación de la identidad de un usuario. De igual manera, al tratarse de un mecanismo de seguridad que se suele acompañar con la autenticación, menciona una breve definición de la autorización, la cual la define como el proceso de determinar si un usuario tiene acceso a un recurso. En la fuente menciona dos ejemplos que puede asociarse a escenarios de cuando usar el mecanismo de seguridad de autenticación. El primero menciona que es cuando sea necesario autenticar un usuario, y el otro segundo es al dar respuesta cuando un usuario sin autenticación intenta acceder a recursos restringidos. Esta fuente no hace mención las amenazas y/o vulnerabilidades asociadas, ni buenas o malas prácticas; existe un apartado llamado *Challenge*, sin embargo, esto no se puede asociar como un reto para el mecanismo de autenticación debido a que *Challenge* se trata de una forma de redirigir las peticiones hechas por un usuario cuando no está autenticado o no cuenta con permisos a una página, cuando es el caso lo manda ya sea al inicio de sesión o alguna otra página que no requiera autenticación.
FS.12	En esta fuente se menciona cuatro mecanismos de seguridad, la autenticación, autorización, la aplicación del HTTPS, y el CORS. La autenticación la define el autor como el proceso de verificar la identidad de un usuario de un sistema. Por otro lado, la autorización se refiere como las acciones que un usuario o sistema tienen permitido realizar. En el de aplicar HTTPS, menciona que sirve para asegurarse que los datos transmitidos entre el cliente y la API vayan encriptados. En cuanto al CORS, menciona que son las políticas que determinan los orígenes que pueden acceder a la API. En cuanto a escenarios donde se utilicen estos mecanismos de seguridad no se menciona. Para las vulnerabilidades si se contesta, menciona sobre el acceso no autorizado el cual se puede asociar al mecanismo de autenticación, se define como un usuario malintencionado intenta acceder a datos confidenciales o realizar acciones no autorizadas. Otra amenaza es la de violación de datos y se puede asociar con autorización y protección de datos, se describe como la filtración de datos de alguien no autorizado, esto da a una exposición de información confidencial. Otra amenaza que menciona la fuente es sobre los ataques de denegación de servicios, trata de atacantes que saturan una API enviando una gran cantidad de peticiones, haciendo que esta no responda o se vuelva lenta, esta amenaza se puede asociar con la autenticación o autorización. Por último, menciona sobre la manipulación de datos y se puede asociar con el mecanismo de protección de datos, en esta amenaza un atacante puede interceptar los datos que se transmiten entre un cliente y la API y este puede modificarlos. Acerca de malas prácticas, buenas o retos no se menciona nada en esta fuente.
FS.13	El autor de esta fuente describe dos mecanismos de seguridad, autenticación el cual se refiere al proceso de identificar un usuario, es un proceso en el que se verifica la identidad de los usuarios que desean intentar acceder a una aplicación o a un sistema. Por otro lado, también habla de la autorización, se refiere al proceso de determinar si un usuario tiene acceso a un recurso, determina las acciones autenticadas que los usuarios pueden realizar dentro de la aplicación. Acerca de los escenarios no hay nada que menciona, pero si sobre las buenas prácticas. Menciona de usar una librería llamada *ASP.NET Core Identity*, sirve para gestionar la autorización y autenticación. Otra buena práctica es el de habilitar la autorización multifactor, ayuda a verificar la identidad del usuario a través de múltiples vías como SMS, correo y/o aplicaciones de autenticación. Menciona el uso de HTTPS para asegurar datos sensibles, hay que usarlo para cifrar los datos en tránsito entre el cliente y el servidor para evitar su intercepción y la manipulación. Para la protección de datos, menciona acerca del uso de almacenes para los secretos, la información confidencial como claves API, cadenas de conexión y entre otros debe resguardarse. Otra buena práctica es la de actualizar las dependencias, las dependencias deben actualizarse con los últimos parches de seguridad. De igual manera recomienda que los eventos de autenticación y autorización deben monitorearse. Hay que prevenir los ataques de inyección validando y satanizando las entradas del usuario usando consultas parametrizadas o *frameworks* como *Entity Framework Core* para evitar la inyección SQL o XSS. Por último, menciona el uso de autenticación externa usando *OAuth2* o *OpenID*, esto para su integración con *loggins* externos como Facebook, Google o Microsoft. En cuanto a malas prácticas, descripción detallada de vulnerabilidades o retos no se menciona nada.
FS.14	El autor de esta fuente habla acerca de los mecanismos de seguridad específicamente dos, la autenticación y autorización. La autenticación verifica la identidad de un usuario que intenta acceder a un sistema o aplicación, asegurándose que es quien dice ser; también menciona que involucra varios métodos como el de contraseña, autenticación por datos biométricos, así como la multifactor. En cuanto a la autorización, determina las acciones y los recursos a los que se permite acceder a un usuario en función de su

	identidad autenticada; esto implica definir roles, permisos y políticas de control de acceso. Esta fuente no responde a escenario y tampoco a vulnerabilidades o amenazas, pero sí a las buenas prácticas. Como primera práctica se puede asociar a la autorización, menciona el uso del Control de Acceso Basado en Roles, esto haciendo uso de la segregación de funciones, mantener los privilegios mínimos y la revisión continua del acceso; otra buena práctica es el del uso de HTTPS para encriptar los datos en tránsito; por último, menciona de la validación de entradas para prevenir ataques de inyección, el uso de contraseñas fuertes y algoritmos de *hashing* para asegurar y almacenar las credenciales del usuario, la actualización de dependencias para mitigar vulnerabilidades de seguridad y la implementación de registro y monitoreo para detectar y responder a accidentes de seguridad siendo proactivos, otra recomendación es el del uso de patrones de diseño para mejorar la escalabilidad, seguridad y mantenibilidad. En cuanto a las preguntas de malas prácticas o retos no se menciona nada.
FS.15	Para esta fuente, se mencionan dos mecanismos de seguridad de ASP.NET Core, la autorización y autenticación. La autorización el autor la define como algo que se usa para validar la identidad de un usuario; en cuanto a la autorización es usada para conceder o remover acceso a un recurso en específico en la aplicación en función de los privilegios de acceso del usuario. Para escenarios o contexto donde se haga uso los mecanismos de seguridad no se menciona nada al igual que en las amenazas o vulnerabilidades. Dentro de las buenas prácticas recalca 4; una técnica para la autenticación, la cual es la basada en tokens, se genera un único token para cada usuario que se autentica, otra manera es el de claves API para autenticar a los usuarios y que de igual manera son únicos. Ya sea las claves API o tokens estos se pasan en el encabezado de la solicitud de cada llamada que se hace a la API. De igual manera menciona la autenticación de dos factores; otra buena práctica es el de solo almacenar los datos que se necesiten durante el tiempo que se necesite, realizando una limpia periódica a los datos antiguos o innecesarios y el de proteger las contraseñas almacenadas usando *hashing*; otra buena práctica que se asocia con la autorización es el de implementar el principio de menor privilegio y solo proveer el acceso necesario; por último, menciona como buena práctica el uso de CORS para impedir el acceso no autorizado a la API desde otros dominios. Para las preguntas de malas prácticas o retos la fuente no menciona información de esto.
FS.16	En esta fuente el autor únicamente menciona dos mecanismos de seguridad para ASP.NET Core, la autorización y autenticación, pero se enfoca más en la autorización. El autor se refiere a la autorización como el proceso para determinar que tiene permitido un usuario realizar; para la autenticación el autor la define como un proceso para la verificación de la identidad de un usuario y de igual manera menciona que la autorización es un mecanismo separado e independiente de la autenticación. En cuanto a escenarios, se menciona un ejemplo de un posible uso que se le puede dar a este mecanismo, en este ejemplo lo menciona así: un usuario administrativo está autorizado a crear una biblioteca de documentos, añadir documentos, editar documentos y eliminarlos. Un usuario no administrativo que trabaje en la biblioteca sólo está autorizado a leer los documentos. Únicamente se responde en esta fuente para estas dos preguntas. En cuanto a amenazas o vulnerabilidades, buenas o malas prácticas que se asocian a los mecanismos de seguridad, o retos que se plantea en su uso, no se menciona nada al respecto.
FS.17	El autor de esta fuente menciona dos mecanismos de seguridad, la autenticación y autorización. La autenticación la define como el proceso de identificación del usuario. Responde a la pregunta "¿Quién eres?"; en cuanto a la autorización, menciona que es algo que viene después de la autenticación y responde a la pregunta "¿Qué se te permite hacer?". Para la pregunta de escenarios, se menciona cuatro escenarios donde se puede aplicar estos mecanismos de seguridad, el primer que menciona es sobre aquellos sitios de comercio electrónico, esto ya que es importante para la gestión de cuentas de usuario, para dar seguimiento a pedidos y dar gestión a funciones administrativas o de clientes; otro escenario es para sistemas de gestión de contenidos, el usar estos mecanismos de seguridad hace que los permisos estén limitados a determinados usuario como editores de contenido permitiendo que únicamente ellos puedan modificar o publicar contenidos; también menciona el escenario para aplicaciones empresariales, en donde es necesario la gestión de inicio de sesión de los usuarios o empleadores, el dar seguimiento a los permisos y la aplicación de las políticas de seguridad de las empresas; el último escenario que menciona el autor es para aquellas plataformas de medios sociales, es importante estos mecanismos ya que se requiere la gestión de los perfiles de los usuarios, gestionar los seguidores y dar asignación de las funciones en base a si es moderador o administrador para lograr gestionar las comunidades de las redes sociales. Acerca de las amenazas, buenas o malas prácticas o retos, el autor no menciona nada.
FS.18	Para esta fuente, el autor menciona varios mecanismos de seguridad. Autenticación, lo define como el proceso de verificación de la identidad de un usuario o sistema; la autenticación se define como las acciones que un usuario o sistema tienen permitido realizar; de igual manera, se menciona el uso de los

	JSON Web Token (JWT), esto es una manera de implementar la autenticación, el autor menciona que es un mecanismo para el aseguramiento de las *APIs* Web mediante la codificación de la información en un token que puede ser fácilmente validado; también menciona del *Cross Origin Resource Sharing* (CORS), son políticas que permiten determinar cuáles origines tienen permitido acceder a una API; también menciona sobre el HTTPS y el transporte seguro, el aplicar esto asegura que los datos transmitidos entre un cliente y la API vayan encriptados; la validación de entradas, menciona que la validación de entradas previene la inyección SQL y ataques XSS; por último, menciona como mecanismo el límite de tasa y las listas blancas de IP, esto para prevenir el abuso de la API y el restringir el número de solicitudes que un cliente puede hacer en un tiempo en específico. Acerca de escenarios no se menciona, pero si sobre las amenazas, una amenaza es el acceso no autorizado, los usuarios maliciosos intentarán acceder a datos sensitivos o realizar acciones no autorizadas; violación de datos, el acceso no autorizado puede hacer que los datos se filtren, resultado en una exposición de la información confidencial; ataques de denegación de servicio (DoS), los atacantes pueden saturar una API enviando un gran número de peticiones haciendo que esta no responda o se vuelva lenta; por último, menciona de la manipulación de datos, los datos transmitidos entre el cliente y la API pueden ser interceptados o modificados. Por último, buenas o malas prácticas, retos, no se menciona nada.
FS.19	En esta fuente se menciona del mecanismo de seguridad de autorización, el autor la define como un conjunto de políticas o mecanismos para proporcionar control sobre los recursos de la aplicación. Escenarios no se menciona nada, sin embargo, para las amenazas si y todo es centrado al control de acceso roto y se puede asociar con la autorización; una vulnerabilidad mencionada es el de no aplicar el principio de denegación por defecto, el acceso a la información o acciones restringidas no debe permitirse por defecto a nadie y debe habilitarse en función de las necesidades del usuario; también menciona como vulnerabilidad el de obtener acceso a recursos restringidos eludiendo el mecanismo de control de acceso de la aplicación mediante la manipulación de la URL; otra vulnerabilidad es cuando los atacantes modifican el identificador único de un usuario para acceder a los datos pertenecientes del usuario; otra vulnerabilidad es el de cambiar los privilegios que tiene un usuario asignado para que sea capaz de obtener los privilegios altos en la aplicación; explotación de vulnerabilidades de métodos de autenticación como los JSON Web Tokens (JWT) mediante la reproducción o manipulación de los tokens; otra vulnerabilidad es cuando orígenes no fiables o no autorizados acceden a la API debido a una mala configuración en CORS; forzar la navegación a las páginas autenticadas por usuarios no autenticados haciéndose pasar por alguien con la autorización y autenticación; los atacantes o usuarios maliciosos acceden a una API o una parte restringida de la aplicación por la falta de configuración del control de acceso. Como buena práctica, el autor menciona el uso de políticas para impedir a los usuarios acceder a recursos no permitidos. En cuanto a malas prácticas o retos no se mencionan.
FS.20	En esta fuente únicamente responde a las preguntas de amenazas y buenas prácticas. Para las amenazas primero menciona acerca del *Cross-Site Scripting* (XSS), esta amenaza permite a los atacantes inyectar scripts maliciosos en las páginas web; *Cross-Site Request Forgery* (CSRF), en esta amenaza se engaña a un usuario para que realice acciones en un sitio en el que está autenticado; inyección SQL, este ocurre cuando un atacante manipula las consultas SQL para obtener acceso a la base de datos. En cuanto a buenas prácticas, menciona el uso de HTTPS por defecto para encriptar los datos en tránsito entre el cliente y el servidor y del uso de HSTS para forzar el HTTPS; uso de *ASP.NET Core Identity* para implementar una autenticación y autorización segura, autenticación basada en roles, y para la API usar JWT y en los *endpoints* usar *Authorize* para asegurarlos; de igual manera validar y sanitizar las entradas y usar mecanismos como la codificación HTML que provee *Razors* para prevenir los ataques XSS; usar consultas parametrizadas o *Entity Framework* para prevenir las inyecciones SQL; utilizar *Razor* o habilitar la protección CSRF usando los tokens anti falsificación para prevenir los ataques CSRF; utilizar políticas de contraseñas fuertes; implementar la API de protección de datos (DPAPI) para encriptar los datos sensibles; asegurar las cookies para mitigar XSS y CSRF colocando los atributos *HttpOnly* y *SameSite* en las cookies; habilitar las políticas de seguridad de los contenidos en los encabezados; limitar el tamaño de solicitudes para mitigar los ataques DoS; implementar el límite de tasa; registrar las acciones sensibles; evitar usar los secretos en código; asegurar las *APIs*; monitorear y actualizar dependencias; asegurar los encabezados con encabezados como *Strict-Transport-Security, X-Content-Type-Options,* y *X-Frame-Options*; utilizar *OAuth2* y *OpenID* para la autenticación segura con terceros; validar las entradas y salidas en los modelos con anotaciones.
FS.21	En esta fuente menciona cuatro mecanismos de seguridad, en la autenticación menciona que verifica las identidades de los usuarios comparando con las credenciales, es decir, nombres de usuario y contraseñas con los datos almacenados; la autorización determina las acciones que los usuarios tienen permitido realizar dentro del servidor, base de datos o aplicación; la protección de datos usando la API de ASP.NET

	Core protege los datos confidenciales mediante la gestión y rotación de claves, garantizando la confidencialidad e integridad de los datos tanto en reposo como en tránsito; aplicar el HTTPS para asegurar las comunicaciones, la transmisión de los datos. Para los escenarios, esta fuente describe uno donde se pueden implementar los mecanismos de seguridad que provee ASP.NET Core, este es para la protección de los datos financieros ya que es importante para los sectores y organizaciones de estos servicios, donde se mantiene la confidencialidad, integridad y accesibilidad de la información sensible. De igual manera, el autor de esta fuente menciona las amenazas que ASP.NET Core mitiga mediante sus mecanismos de seguridad, sin embargo, solo las menciona y no las describe, estas amenazas son el Cross site Scripting (XSS), la inyección SQL, el *Cross-Site Request Forgery* (CSRF) y los ataques de redireccionamiento abierto. En cuanto a buenas prácticas, malas, o retos no se menciona nada en la fuente.
FS.22	En esta fuente se menciona buenas prácticas, una de ellas es aplicar técnicas de validación y desinfección de entradas para evitar ataques como el XSS, de igual manera menciona el uso de consultas parametrizadas y procedimientos almacenados para evitar la inyección SQL, también, recomienda evitar el uso de componentes y bibliotecas obsoletas o vulnerables; otra buena práctica que se asocia con la autenticación y autorización es el uso de mecanismos de autenticación seguros como la autenticación multifactor (MFA) así como las políticas de contraseñas seguras. El uso de protocolos seguros como *OAuth, OpenID Connect* para la autenticación y autorización y de igual manera emplear la autorización basada en roles; para la protección de datos, como buena práctica el autor menciona el cifrado de los datos confidenciales en reposo esto utilizando algoritmos de cifrado fuertes, contraseñas hash y hashes con sal para proteger las credenciales de los usuarios. De igual manera, recomienda emplear técnicas de almacenamiento seguro de datos, como el cifrado y control de acceso adecuado; también menciona que hay que proteger la comunicación usando protocolos seguros como HTTPS con protocolos SSL y el reforzar usando HSTS (HTTP *Strict Transport Security*) para reforzar la seguridad de la comunicación, también menciona actualizar periódicamente los certificados SSL y como única mala práctica el de evitar los certificados auto firmados; menciona la aplicación de codificación de la salida de los contenidos que el usuario genere y el de HTML generado dinámicamente para prevenir el XSS, agrega el uso de políticas de seguridad de contenidos (CSP) para restringir los scripts maliciosos e implementar la validación y el filtrado de entradas. En cuanto a malas prácticas a excepción de la que se agregó no se menciona más y retos tampoco se incluye en esta fuente.
FS.23	Esta fuente habla de CORS, función de seguridad que permite a las aplicaciones controlar los dominios externos que pueden acceder a sus recursos; ASP.NET Core ofrece cifrado y descifrado para proteger los datos confidenciales; HTTPS para asegurar la comunicación encriptando los datos transmitidos entre el servidor y el cliente; JWT para la autenticación y autorización. Para escenarios, la autorización basada en roles es ideal y simple en aplicaciones donde se tienen claros los roles y un limitado número de permisos; autorización basada en reclamos, ideal en aplicaciones complejas con múltiples tipos de usuarios y numerosos permisos y requisitos de control de acceso; para autenticación está la de Windows, ideal para aplicaciones en Intranet con un sistema basado en Windows; autenticación de formularios ideal para utilizar almacenamiento de credenciales personalizado/definido por el usuario; *OAuth, OpenID Connect*, SAML ideal para aplicaciones modernas que necesitan una gestión de identidades escalable. Amenazas está el CSRF, un atacante engaña a un usuario para que ejecute acciones no deseadas en una aplicación mientras está autenticado; XSS, un atacante inyecta scripts maliciosos en sitios web de confianza. El objetivo principal es robar información sensible; *Insecure Direct Object*, una aplicación expone referencias a objetos internos a los usuarios sin validar los derechos de acceso adecuados. Para buenas prácticas menciona, validación de entradas; codificación de la salida usando *Html.Encode*(); CSP; validación de solicitudes, bloquea contenido potencialmente inseguro; usar la API de protección de datos; cifrado simétricos y asimétrico; cadenas seguras usando *SecureString*; obtener certificados SSL/TSL; redirigir de HTTP a HTTPS, usar HSTS; cookies seguras; usar referencias indirectas a objetos como índices o mapeos para no revelar identificadores; validar permisos de entrada; encabezados seguros (*X-Content-Type-Options, X-Frame-Options, X-XSS-Protection*); prácticas de codificación seguras; validar campos de entradas basada en listas blancas; expresiones regulares; rechazar entradas inesperadas; validación del lado de cliente y servidor; utilizar *Razor* y su sintaxis para codificar la salida.
FS.24	Para esta fuente, el autor no menciona los mecanismos de seguridad, escenarios ni amenazas, se enfoca en hablar únicamente de buenas prácticas de seguridad de ASP.NET Core que se pueden asociar a los mecanismos de seguridad que provee el *framework*. Se puede asociar para la autorización la seguridad basada en roles, el autor menciona que esta función permite a los desarrolladores controlar el acceso a ciertas partes de la aplicación en función del rol que tenga el usuario, esto garantiza que sólo los usuarios autorizados puedan acceder a información sensible; usar comunicaciones seguras como con protocolos

	de HTTPS y SSL/TSL para proteger datos esenciales en tránsito entre el cliente y el servidor; usar la API de protección de datos, esta API provee mecanismos para la encriptación y desencriptación de datos sensibles, ayuda a proteger los datos almacenados en bases de datos o datos transmitidos a través de la red; también para la autenticación y autorización, se recomienda usar la librería de ASP.NET Core *Identity*, ya que ofrece mecanismos robustos para administrar la autenticación y autorización de los usuarios, incluye la autenticación multifactor (MFA). También menciona el uso de los protocolos de Auth y *OpenID Connect* para usar tokens seguros basados en autenticación. De igual manera, el control de acceso basado en roles (RBAC) para asegurar la aplicación concediendo a los usuarios autorización basado en su rol y así darles los permisos necesarios para realizar sus tareas; la validación de entradas es otra buena práctica, haciendo uso de consultas parametrizadas y procedimientos almacenados para mitigar la inyección SQL; usar el token contra falsificaciones para prevenir el *Cross Site Request Forgery*. En cuanto a malas prácticas o retos no se menciona nada en esta fuente.
FS.25	En esta fuente el autor no menciona ni explica un mecanismo de seguridad, pero si menciona un escenario, en algunos servicios de *backend* donde la conexión es de extremo público, no es necesario configurar la seguridad de la conexión en cada nodo, es decir, cuando se crea un proyecto usando las plantillas de Visual Studio ya trae por defecto la redirección de HTTPS y HSTS, sin embargo, en estos casos donde la conexión será pública no es necesario tenerlos habilitado. Amenazas no se mencionan, pero como buena práctica el autor recomienda que las aplicaciones web de ASP.NET Core en producción utilicen: middleware de redirección de HTTPS (*UseHttpsRedirection*) para redirigir las solicitudes HTTP a HTTPS, el uso de HSTS (*UseHsts*) para enviar cabeceras HTTP *Strict Transport Security Protocol* (HSTS) a los clientes; se recomienda utilizar redireccionamientos temporales en lugar de permanentes, también recomiendan HSTS solo en producción. Por otro lado, como malas prácticas HSTS no se recomienda en desarrollo porque la configuración de HSTS es altamente cacheable por los navegadores, es decir, cuando se inicie la aplicación web en el navegador lo recordará y empezará a usar HTTPS, sin embargo, si se tiene otra aplicación en desarrollo y se inicia, el navegador recordará el uso de HTTPS y empezará a intentar redirigir está a usar este protocolo y es muy posible que falle; también mencionan como mala práctica y no es recomendable el crear un certificado de desarrollo en un entorno que vaya a ser redistribuido, como una imagen de contenedor o una máquina virtual. Si lo hace, puede provocar la suplantación de identidad y la elevación de privilegios. Para retos no se menciona nada en esta fuente.
FS.26	Para esta fuente, se describe un único mecanismo de seguridad, el aplicar HTTPS, menciona que SSL es un protocolo que facilita la comunicación segura entre clientes y servidores a través de una red al permitir cifrar la comunicación, esto es posible asociarlo a HTTPS ya que SSL/TSL es el protocolo que utiliza. En cuanto a la pregunta de escenarios, no se responde nada. Dentro de las amenazas se mencionan tres, el *Cross-Site Request Forgery* (CSRF) este ataque consiste en engañar a un usuario para que realice actividades maliciosas mientras está conectado a una aplicación. Estos ataques se realizan más usualmente engañando a los usuarios con correos de phishing para atraerlos a sitios web maliciosos; Cross-Site Scripting (XSS), se refiere al acto de inyectar un script malicioso utilizando campos de entrada o formularios dentro de una aplicación web, esto con la intención de robar datos sensibles como credenciales de acceso; y la otra amenaza es la inyección SQL, se produce cuando un atacante inserta comandos SQL maliciosos dentro de las consultas SQL, esto provoca la exposición de información confidencial. La fuente menciona buenas prácticas, una es el uso de HSTS ya que esto evita los ataques de protocolo descendente y el secuestro de cookies asegurándose de que el servidor web se comunica mediante una conexión HTTPS y bloqueando todas las conexiones HTTP inseguras; igual recomienda el uso de los tokens anti-falsificación para mitigar los ataques CSRF, estos tokens funcionan enviando dos valores diferentes al servidor con cada POST. Uno de los valores se envía como una cookie del navegador, y el otro se envía como datos del formulario. A menos que el servidor reciba ambos valores, rechazará la solicitud. Para malas prácticas o retos el autor de la fuente no hace mención.
FS.27	En esta fuente, menciona HTTPS, mediante SSL se establece una conexión segura o cifrada entre el cliente y el servidor. Para amenazas, XSS, es la inyección de un script malicioso a través del campo de entrada/formulario de una página web con la intención de robar información confidencial; Inyección SQL, ataque en el que usuarios no autorizados inyectan código SQL malicioso que luego se ejecuta en una base de datos, permitiendo a los atacantes acceder a información confidencial almacenada en ella; CSRF, un atacante actúa como una fuente de confianza y envía datos falsificados a un sitio. El sitio procesa los datos falsificados porque cree que proceden de una fuente de confianza; XXE (XML *External Entity*), un analizador XML mal configurado procesa una entrada XML que contiene código XML malicioso. Este tipo de ataque puede causar un ataque de denegación de servicio mediante la inyección de entidades. Dentro de las buenas prácticas, se menciona el uso del atributo de expresiones regulares; el uso de objetos

	modales de expresiones regulares, se llaman a métodos estáticos de una clase *Regex*; codificación HTML, *Razor* codifica las entradas para no ejecutar scripts maliciosos; codificación de la URL, codificar los parámetros de las consultas en la URL; validar las entradas en el cliente y el servidor; usar procedimientos almacenados y consultas parametrizadas; usar Entity Framework Core u otro ORM; almacenar datos encriptados; usar token anti-falsificación agregando [*ValidateAntiForgeryToken*] en las peticiones POST; usar HSTS para rechazar las conexiones HTTP; usar *DtdProcessing* para procesar el XML si se usa *XmlTextReader* para evitar el ataque XML; auditar las autenticaciones. Como malas prácticas menciona no usar HSTS en una fase de desarrollo; no eliminar las cookies de autenticación después de un cierre de sesión exitoso; no almacenar datos confidenciales que incluyan código de base de datos o de aplicación en ningún sitio.
FS.28	En esta fuente menciona dos mecanismos de seguridad, el aplicar HTTPS garantiza que los datos entre el servidor y el cliente estén cifrados, lo que dificulta que los atacantes intercepten o manipulen los datos; *Anti-Forgery Tokens*, mitigan ataques CSRF asegurándose que las solicitudes que se envíen al servidor sean de una fuente legitima de usuario. En cuanto a escenarios no se habla de nada. En amenazas se menciona del ataque de *Cross-Site Request Forgery* (CSRF), explotan la confianza que una aplicación web tiene en el navegador de un usuario autenticado, permitiendo a los atacantes realizar acciones no autorizadas en nombre del usuario, un ejemplo de esto es que un usuario este autenticado en una página web, si este usuario visita una página maliciosa, el sitio malicioso puede enviar peticiones al sitio autenticado sin que el usuario lo note haciendo que se puedan cambiar configuraciones del usuario ya que se hace pasar por el usuario. Dentro de las buenas prácticas, se menciona una y es acerca de los archivos de configuración, estos suelen contener datos confidenciales como cadenas de conexión de la base de datos, es importante cifrar estos archivos para que sea más seguro y hace más difícil para que personas no autorizadas intenten acceder a su contenido. El uso de esta práctica es eficaz para proteger los datos confidenciales y detalles de configuración. Para malas prácticas o retos la fuente no menciona nada.
FS.29	En esta fuente únicamente se menciona un mecanismo de seguridad, el cual la administración de secretos, sin embargo, no se detalla en más una descripción o concepto de esto. En cuanto a escenarios, amenazas no se hace mención tampoco. Como buenas prácticas, la fuente menciona estas: considerar el tiempo de vida de los secretos, esto ya que los secretos se crean, revocan y caducan, y el no considerar alguno de estos pasos la seguridad los sistemas puede verse afectada; llevar un inventario de los secretos, esto se refiere a que es necesario saber la cantidad de secretos, donde están alojados y quien tiene acceso a ellos ya que si no se sabe solo hay una proliferación excesiva de secretos; permitir el acceso a solo clientes autorizados, es necesario aplicar el control de acceso para conceder el secreto en específico para que puedan lograr la tarea que requiere dicho secreto; auditoria de operaciones, las operaciones como el acceso a la rotación de secretos debe registrarse; cifrado de secretos, estos deben estar en reposo, por lo tanto mientras sea así deben estar cifrado y es necesario reducir el tiempo que no están en reposo para que no sean vulnerables y es por ello la importancia de SSL/TSL para cuando estos no están en reposo; garantizar la disponibilidad, tener los secretos listos cuando se necesiten. A nivel de red, evite los puntos únicos de fallo y garantice el acceso a todos los clientes que lo necesiten. Para malas prácticas, los secretos mal generados no proporcionan un nivel de protección adecuado. Y por último para retos, el autor de la fuente no menciona nada relacionado a esto.
FS.30	En esta fuente se habla sobre dos mecanismos, el token *AntiForgery*, el cual el autor describe que es un atributo HTML, y se debe establecer su valor como true para que se genere dicho token anti-falsificación y que además es necesario usar el atributo [*ValidateAntiForgeryToken*] para poder comprobar que se generó un token válido. El autor menciona que una amenaza/vulnerabilidad asociada a este mecanismo es la falsificación de peticiones en sitios cruzados (CSRF por sus siglas en ingles), y describe que este se trata de cuando un atacante actúa como una fuente de confianza y envía datos falsificados a un sitio, el sitio los procesa porque cree que proceden de una fuente de confianza. Sin embargo, esta fuente no proporciona más información que conteste sobre retos, escenarios, malas y buenas prácticas para este mecanismo. El segundo mecanismo del que habla es sobre HTTPS *Enforcement* (aplicación de HTTPS), la cual la define como una política de seguridad web que protege una aplicación web de ataques de protocolo de degradación y secuestro de cookies, la cual obliga a usar conexiones HTTPS y rechaza las conexiones HTTP. No se mencionan escenarios o amenazas asociadas a este mecanismo, pero si habla sobre algunas buenas prácticas, menciona que para configurar este se debe definir *MaxAge*, el *Transport-Security*, si se desea que la cabecera *Strict-Transport-Security* esté disponible también para los subdominios se debe incluir *IncludeSubDomains*, asimismo se puede añadir el *Preload* ya que añade soporte de precarga a la cabecera *Strict-Transport-Security*, y por último el poner *ExcludedHosts* que es una lista de nombres de host que no añadirán la cabecera HSTS. Finalmente menciona que una mala práctica es usar el middleware HSTS en entorno de desarrollo, ya que los navegadores almacenan en cache el encabezado

FS.31	En esta fuente se habla de CORS, los autores describen que es un estándar del W3C que permite a un servidor relajar la política del mismo origen, que permite a un servidor permitir explícitamente algunas peticiones de origen cruzado y rechazar otras. También se mencionan buenas prácticas para el uso de este mecanismo, como el que debe de invocarse en el orden correcto (*UseCors* debe invocarse antes de *UseResponseCaching*), el habilitar *Cors* con el atributo [*EnableCors*] y aplicar una política con nombre solo a aquello endpoints que requieran CORS proporciona un control más preciso, por ello no se debe definir una política predeterminada y no se debe usar enrutamiento de endpoints. Respecto a las malas prácticas se menciona que no se debe usar un middleware y el atributo [*EnableCors*] en una misma aplicación, usar *AllowAnyOrigin* y *AllowCredentials* es una configuración insegura y puede dar lugar a falsificación de peticiones entre sitios, y finalmente menciona que permitir credenciales de origen cruzado es un riesgo de seguridad, ya que asi un sitio web de otro dominio puede enviar las credenciales de un usuario que ha iniciado sesión a la aplicación en nombre del usuario sin su consentimiento. Para las preguntas sobre los retos, escenarios y amenazas/vulnerabilidades asociadas al mecanismo, en la fuente no se menciona nada al respecto.
FS.32	En esta fuente solo se habla sobre CORS, el cual el autor lo describe como un mecanismo de seguridad que permite solicitar recursos restringidos de una página web desde un dominio distintos del que ha servido la página. El autor menciona que la amenaza/vulnerabilidad asociada a este mecanismo es la falsificación de peticiones entre sitios (CSRF por sus siglas en ingles). Asimismo, en la fuente se menciona que como buena práctica es importante situar la llamada a *UseCors* correctamente en el proceso, antes del middleware que gestiona las peticiones, como la autenticación o los *endpoints*, para garantizar así que la política CORS se aplica correctamente. Por otro lado, el autor menciona que dentro de los retos en el uso del mecanismo, es la dificultad de depuración, ya que los errores relaciones con CORS pueden no ser notificados por los navegadores, lo que dificulta la identificación del problema, también menciona que implementar CORS puede ser complicado en escenarios donde la aplicación necesita soportar múltiples orígenes con diferentes niveles de permiso, por ello recomienda que en estos casos es importante definir políticas CORS específicas para cada origen o grupo de orígenes, de forma que se asegure que cada uno de estos tiene los permisos adecuados para acceder a los recursos necesarios, sin embargo, un problema de una configuración demasiado permisiva es que puede exponer la aplicación a riesgos de seguridad. El autor no menciona nada sobre malas prácticas o escenarios donde se utilice este mecanismo de seguridad.
FS.33	En esta fuente se habla sobre CORS, la definen como un mecanismo que permite solicitar recursos restringidos en una página web desde otro dominio del que procede el recurso, o, en otras palabras, que es una comunicación entre dominios URL diferentes. En la fuente se menciona que la amenaza/vulnerabilidad asociada es el Cross-site scripting (XSS), el autor define que una buena práctica es el configurar las cabeceras adecuadamente, como usar «*Access-Control-Allow-Origin*» y «*Access-Control-Allow-Headers*» para no exponer accidentalmente información sensible sobre la aplicación o usuarios a través de las peticiones, asimismo, menciona que se utilicen solicitudes de verificación previa, ya que estas permiten al servidor determinar si puede responder adecuadamente antes de realizar la petición, es asi que si una solicitud de verificación previa falla, el navegador no enviara la solicitud real, asi se evita que usuarios malintencionados o *bots* realicen peticiones entre dominios no autorizados que podrían abusar del sitio web. Por otro lado, se menciona sobre malas prácticas el utilizar solicitudes de origen comodín (*wildcars*), ya que estas permiten especificar que cualquier origen puede realizar una petición, lo que genera violaciones a las políticas de seguridad de contenidos (CSP por sus siglas en inglés) por parte de sitios web maliciosos que intentes realizar peticiones entre dominios. Respecto a escenarios y retos asociados al uso del mecanismo en la fuente no se habla al respecto.
FS.34	Se habla sobre CORS en esta fuente, el autor describe que es una característica de seguridad implementada por los navegadores web para evitar que las páginas web realicen peticiones a un dominio diferente del que sirvió la página web. En la fuente no se mencionan escenarios donde se utilice este mecanismo, sin embargo, si menciona que una amenaza/vulnerabilidad asociada a este mecanismo es el acceso no autorizado, pero no lo describe más allá. El autor describe que una buena práctica es usar los orígenes comodín, ya que, en lugar de especificar un único origen, puede utilizar *Builder.AllowAnyOrigin*() para permitir peticiones de cualquier origen, pero, también hace hincapié que es algo que se debe hacer con cuidado porque este enfoque puede plantear riesgos de seguridad. Asimismo, menciona que si en la aplicación la API y el *frontend* están en dominios diferentes y es necesario enviar credenciales (como por ejemplo cookies o tokens de sesión), se pueden añadir *AllowCredentials*() a la política CORS y que se puede personalizar aún más las políticas de CORS de forma que se permitan métodos específicos, cabeceras y más. Respecto a escenarios donde se utilice CORS, malas prácticas y retos asociados al uso del mecanismo, el autor no menciona nada para responder a esas preguntas.

FS.35	En esta fuente se habla sobre el codificador HTML (HTML *encoder*), de acuerdo con los autores, la codificación de la salida garantiza que los datos devueltos por la API se limpian correctamente para que el navegador del usuario no pueda ejecutarlos como código. Los autores no mencionan algún escenario, pero si hablan sobre la vulnerabilidad que está asociada a este mecanismo, el Cross-site scripting (XSS), la cual la definen como una vulnerabilidad de seguridad que permite a un ciber atacante colocar scripts del lado del cliente en páginas web, y cuando los usuarios cargan paginas afectadas, se ejecutan los scripts del ciber atacante, lo que permite robar cookies, tokens de sesión, o redirigir el navegador a otra página. Asimismo, se menciona que una buena práctica es la validación, ya que puede ser una herramienta útil para limitar los ataques de XSS, antes de poner datos no fiables dentro de un elemento HTML, asegurarse de que esta codificado con HTML; evitar que el código quede expuesto a XSS basado en DOM con propiedades apropiadas como *node.textContent* o *node.InnerText*. Por otro lado, los autores mencionan que dentro de las malas prácticas que se asocian es el poner datos no confiables en la entrada HTML, no usar la clase *HtmlString* en combinación con la entrada no fiable, no se deben concatenar entradas no fiables en JavaScript para crear elementos DOM ni utilizar *document.write*() en contenido generado dinámicamente, no utilizar datos no fiables como parte de una ruta URL y no confiar únicamente en la validación, codificar siempre la entrada no fiable antes de la salida. Respecto a la pregunta de retos asociados al uso del mecanismo no se menciona nada en la fuente.
FS.36	En esta fuente solo se habla sobre el codificador HTML (HTML *encoder*), el autor lo describe como un codificador del contenido generado por el usuario antes de renderizarlo en HTML para prevenir ataques XSS, se menciona un escenario sencillo donde se considere una aplicación web sencilla, con una sección de comentarios donde los usuarios pueden publicar mensajes, sin la validación y desinfección adecuada, la aplicación es susceptible a sufrir ataques XSS. Esta vulnerabilidad, el autor describe que se produce cuando los atacantes inyectan scripts maliciosos en páginas web visitadas por otros usuarios, dichos scripts explotan vulnerabilidades en el manejo de las entradas del usuario, provocando accesos no autorizados, robo de datos o manipulación. Por ello dentro de las buenas prácticas que menciona el autor, es la validación y desinfección de entradas, menciona que es una actividad que se debe de realizar de forma rigurosa ya que es una forma de eliminar las vulnerabilidades XSS, la implementación de políticas de seguridad de contenido, configurando encabezados CSP para restringir la ejecución de scripts de fuentes no autorizadas, y el uso de tokens anti-falsificación para protegerse contra los ataques CSRF. En cuanto a las preguntas de malas prácticas y retos asociados al uso del mecanismo de seguridad, el autor no habla al respecto.
FS.37	En esta fuente se habla sobre el codificador HTML (HTML *encoder*), de forma más específica sobre los CSP, los cuales desactivan la ejecución de scripts en línea y solo permite aquellos scripts procedentes de fuentes de confianza, el autor menciona que la vulnerabilidad asociada a este mecanismo es el Cross-site scripting, el cual define que es aquel en que un atacante inyecta scripts maliciosos en un sitio web de confianza, lo que provoca el robo de datos y el secuestro de la sesión. Asimismo, el autor describe que pueden pasar ataques de inyección de datos (JavaScript malicioso), *clickjacking* (usa *iframe* invisibles), vulnerabilidades de contenido mixto (una aplicación HTTPS carga recursos a través de HTTP), ataque *Man-in-the-Middle* (intercepción en la comunicación entre usuario y navegador). El autor menciona que una buena práctica es establecer una sólida política CSP y mejorarla usando directivas *report-uri*, ya que esta directiva especificara el lugar al que el navegador debe enviar los informes en el momento de la infracción. Asimismo, el autor menciona unos retos que implica el usar CSP, y es que este mecanismo puede bloquear recursos legítimos como scripts, fuentes, estilos o imágenes si se cargan desde servicios de terceros (se puede evitar creando una lista blanca de fuentes), también CSP puede bloquear JavaScript y CSS en línea de forma predeterminada (la solución es no permitir '*unsafe-inline*') e incluso puede bloquear inadvertidamente el contenido dinámico cargado desde JavaScript y romper la funcionalidad, igualmente menciona que el proceso de entender y depurar CSP es una actividad que consume mucho tiempo y mantener una política CSP rigurosa se vuelve abrumador a medida que la aplicación crece y comienza a integrar bibliotecas de terceros. Finalmente menciona que no todos los navegadores soportan la aplicación de CSP, respecto a malas prácticas o escenarios, el autor no habla al respecto.
FS.38	En esta fuente solo se habla sobre el codificador HTML (HTML *encoder*), la cual el autor la define como una forma de codificar la entrada de variables para evitar la ejecución de scripts. La vulnerabilidad que está asociada a este mecanismo son las secuencias de comando en sitios cruzados (XSS por sus siglas en ingles), en la fuente se menciona que es aquella en la que los atacantes pueden insertar scripts maliciosos, a menudo escritos en JavaScript, en sitios en línea utilizando un fallo de seguridad, esto da a los hackers la capacidad de alterar contenido de la página o aprovecharse de las cookies o tokens de sesión para obtener credenciales de inicio de sesión. El autor menciona que como buenas prácticas para este mecanismo y evitar XSS, se deben validar las entradas del usuario utilizando expresiones regulares para

	asegurarse de que cumplen los criterios específicos, evitar utilizar texto sin formato en las cadenas de consulta; en su lugar utilizar cadenas de consultas codificadas, utilizar métodos estáticos de clase *Regex* para validar eficazmente las entradas del usuario. También menciona que hay que utilizar consultas parametrizadas y procedimientos almacenados para detener ataques de inyección SQL y hay que evitar componentes y bibliotecas obsoletas o de riesgo, finalmente, menciona que es esencial mantener actualizado el *framework* ASP.NET Core y sus dependencias con los últimos parches. En la fuente el autor no habla respecto a las preguntas de escenarios, malas prácticas y retos asociados al uso del mecanismo.