API de Controle

Vulnerabilidad	Descripcion
Autorización de nivel de objeto vulnerable	Las API tienden a exponer puntos finales (end-points) que manejan identificadores de objetos, creando un problema de Control de Acceso y amplieando la superficie de ataque. En cada función que accede a datos utilizando una entrada del usuario, se deben considerarse agregar verificaciones adicionales de autorización a nivel de objeto. Ejemplo: /api/shops/{shopID}/create
Autenticación de usuario vulnerable	los mecanismos de autenticación se implementan incorrectamente, lo que permite a los atacantes comprometer los tokens de autenticación o explotar fallas de implementación para asumir las identidades de otros usuarios de manera temporal o permanente. La incapacidad del sistema para identificar al cliente/usuario compromete la seguridad de la API en general. Ejemplo: /api/system/verification-codes/{smsToken} (con un token de 4 dígitos) Los end-point y los flujos de autenticación son activos y deben protegerse. Los formularios de "Olvidé mi contraseña / restablecer contraseña" debe tratarse de la misma manera que los mecanismos de autenticación más sensibles.

Exposición excesiva de datos	Al esperar implementaciones genéricas, los desarrolladores tienden a exponer todas las propiedades de los objetos sin tener en cuenta su sensibilidad individual, confiando en que los clientes realicen el filtrado de datos antes de mostrarlo al usuario. Ejemplo: /api/articles/{articleId}/comments/{commentId} (expone todos los comentarios y los datos personales del autor del mismo) Las API devuelven, por diseño, datos confidenciales al cliente. Estos datos
Fallas en la restricción y limitación de recurs	Muy a menudo, las API no imponen ninguna restricción sobre el tamaño o la cantidad de recursos que puede solicitar el cliente. Esto no solo puede afectar el rendimiento del servidor API, lo que lleva a la denegación de servicio (DoS), sino que también deja la puerta abierta a fallas de autenticación a través de fuerza bruta.
Autorización vulnerable a nivel de función	Las políticas complejas de control de acceso con diferentes jerarquías, grupos y roles, y una separación poco clara entre las funciones administrativas y regulares, tienden a conducir a fallas de autorización. Al explotar estos problemas, los atacantes obtienen acceso a los recursos y/o funciones administrativas de otros usuarios. Ejemplo: GET /api/admin/v1/users/all (muestra todos los usuarios de la aplicación) La mejor manera de encontrar problemas de autorización a nivel de función es realizar un análisis profundo del mecanismo de autorización, teniendo en cuenta la jerarquía de usuarios, los diferentes roles o grupos en la aplicación.

Asignación masiva	La vinculación de datos proporcionados por el cliente (por ejemplo, JSON) a modelos de datos, sin un filtrado de propiedades adecuado basado en una lista blanca, generalmente conduce a una asignación masiva de datos. Adivinar las propiedades de los objetos, explorar otros end-point de la API, leer documentación o proporcionar
Configuración incorrecta de seguridad	La configuración incorrecta de seguridad suele ser el resultado de configuraciones predeterminadas no seguras, configuraciones incompletas o ad-hoc, almacenamiento abieto en la nube, encabezados HTTP mal configurados, métodos HTTP innecesarios, uso compartido de recursos de origen cruzado permisivo (CORS) y mensajes de error detallados que contienen información confidencial.
Inyección	Las inyecciones SQL, NoSQL, comandos, etc., ocurren cuando se envían datos no confiables a un intérprete como parte de un comando o consulta. Los datos maliciosos del atacante pueden engañar al intérprete para que ejecute comandos no deseados o acceda a los datos sin la autorización adecuada.

Gestión de activos inapropiada	Las API tienden a exponer más end-points que las aplicaciones web tradicionales, lo que hace que la documentación adecuada y actualizada sea muy importante. Los hosts adecuados y el inventario de versiones de API implementadas también juegan un papel importante para mitigar problemas como versiones de API obsoletas y puntos finales de depuración expuestos.
Registro y monitoreo insuficiente	El registro y el monitoreo insuficientes, junto con la integración faltante o ineficaz de la respuesta a incidentes, permite facilitar los ataques, mantener la persistencia, controlar y manipular más sistemas, extraer o destruir datos sin control, etc. La mayoría de los estudios de incumplimiento demuestran que el tiempo para detectar un incumplimiento es superior a 200 días, generalmente detectado por partes externas en lugar de procesos internos o monitoreo.

s de Seguridad

Control Seguridad a nivel Web
•Implementar un mecanismo de autorización adecuado que se base en las políticas y la jerarquía del usuario. •En cada función que utiliza una entrada del cliente para acceder a un registro de la base de datos, se debe usar un mecanismo de autorización para verificar si el usuario conectado tiene acceso para realizar la acción solicitada en el registro. •Se debería usar valores aleatorios e impredecibles como GUID para las ID de los registros. •Se deben escribir pruebas para evaluar el mecanismo de autorización.
•Asegurarse de conocer todos los flujos posibles para autenticarse en la API (mobile, web, enlaces profundos que implementan autenticación con un solo clic, etc.) • Asegurarse de comprender qué y cómo se usan. Por ejemplo, OAuth no es autenticación, y tampoco lo son las API keys. •Generación de tokens, almacenamiento de contraseñas. Usar los estándares. •Los puntos finales de recuperación de credenciales y "olvido de contraseña" deben tratarse como end-points de inicio de sesión en términos de fuerza bruta, limitación de velocidad y protecciones de bloqueo. •Donde sea posible, implementar la autenticación multifactor (MFA). •Implementar mecanismos anti-fuerza bruta para mitigar el relleno de credenciales automático, ataques de diccionario y ataques de fuerza bruta en los end-points de autenticación. Este mecanismo debería ser más estricto que el mecanismo de limitación de velocidad regular en la API. •Implementar un mecanismo de bloqueo o recuperación de cuenta para evitar la fuerza bruta contra usuarios específicos. Implemente verificaciones de contraseña débil y un CAPTCHA. •Las API keys no deben usarse para la autenticación de usuarios, sino para la autenticación de la aplicación o el proyecto del cliente.

•Nunca confiar en el cliente para filtrar datos confidenciales. •Revisar las respuestas de la API para asegurarse de que solo contengan los datos necesarios. •Los ingenieros de backend siempre deben preguntarse "¿quién es el consumidor de los datos?" antes de exponer un nuevo end-point API. •Evitar el uso de métodos genéricos como to_json() y to_string(). En cambio, se debe seleccionar las propiedades específicas que realmente se quieran devolver al cliente. •Clasificar la información confidencial y de identificación personal (PII) y revisar todas las llamadas API que devuelven dicha información para ver si estas respuestas plantean un problema de seguridad o confidencialidad. •Implementar un mecanismo de validación de respuesta basado en esquemas como una capa adicional de seguridad. Como parte de este mecanismo, se debe definir y aplicar los datos devueltos por todos los métodos API, incluidos los errores.
•Docker facilita la limitación de memoria, CPU, número de reinicios, descriptores de archivos y procesos. Es recomendable su utilización. •Implementar un límite sobre la frecuencia con que un cliente puede llamar a la API dentro de un marco de tiempo definido. •Notificar al cliente cuando se excede el límite. proporcionando el número máximo y la hora a la que se restablecerá el límite. •Validar adecuadamente (del lado del servidor) el formato de la consulta y los parámetros del cuerpo de la solicitud, específicamente el relacionado al número de registros que se devolverán en cada respuesta. •Definir y aplicar un tamaño máximo de datos en todos los parámetros entrantes y payloads, como la longitud máxima de las cadenas y el número máximo de elementos en las matrices y los objetos.
•La aplicación debe tener un módulo de autorización consistente y fácil de analizar que se invoque desde todas las funciones de negocio. Con frecuencia, dicha protección es proporcionada por uno o más componentes externos al código de la aplicación. •Los mecanismos de aplicación deberían denegar cualquier tipo de acceso por defecto, requiriendo concesiones explícitas a roles específicos para acceder a cada función. •Revisar los end-points y su autorización a nivel de función, teniendo en cuenta la lógica empresarial de la aplicación y la jerarquía de grupos. •Asegurar que todos los controladores administrativos implemente comprobaciones de autorización basadas en los grupos y el rol del usuario. •Asegurar que las funciones administrativas implementen comprobaciones de autorización basadas en el grupo y el rol del usuario.

•Evitar utilizar funciones que enlacen automáticamente la entrada de un cliente en variables u objetos internos. •Incluir en una lista blanca solo las propiedades que el cliente pueda y deba actualizar. •Utilizar las funciones integradas para incluir en la lista negra las propiedades a las que los clientes no deberían acceder. •Si corresponde, definir explícitamente y forzar esquemas en datos de entrada.
•Se debe desarrollar un proceso de fortalecimiento (hardening) repetible que permita la implementación rápida y fácil de un entorno adecuadamente protegido. •Revisar y actualizar configuraciones en toda la pila de API. La revisión debe incluir: archivos de orquestación, componentes de API y servicios en la nube (por ejemplo, permisos S3). •Un canal de comunicación seguro para todos los activos estáticos (por ejemplo, imágenes y CSS). •Incluir un proceso automatizado para evaluar continuamente la efectividad de la configuración y las configuraciones en todos los entornos. •Evitar que se envíen excepciones y otra información valiosa a los atacantes y, si corresponde, definir y aplicar esquemas de respuestas de error. •Asegurar de que solo los verbos HTTP especificados puedan acceder a la API. Todos los demás verbos HTTP deben estar deshabilitados (por ejemplo, HEAD). •Las API que esperan acceder desde clientes basados en el navegador (por ejemplo, front-end de WebApp) deben implementar una política adecuada de Cross-Origin Resource Sharing (CORS).
•La prevención de la inyección requiere mantener los datos separados de los comandos y consultas. •Realizar la validación de datos utilizando una biblioteca única, confiable y mantenida activamente. •Validar, filtrar y desinfectar todos los datos proporcionados por el cliente u otros datos provenientes de sistemas integrados. •Los caracteres especiales se deben escapar utilizando la sintaxis específica para el intérprete de destino. •Utilizar una API segura que proporcione una interfaz parametrizada. •Siempre limitar el número de registros devueltos para evitar la divulgación masiva en caso de inyección. •Validar los datos entrantes utilizando filtros suficientes para permitir solo valores válidos para cada parámetro de entrada. •Definir tipos de datos y patrones estrictos para todos los parámetros de tipos de cadena.

•Hacer un inventario de todos los hosts de la API y documentar los aspectos importantes de cada uno de ellos, centrándose en el entorno de la API (por ejemplo, producción, test, desarrollo), quién debe tener acceso de red al host (por ejemplo, público, interno, socios) y versiones de la API •Hacer un inventario de los servicios integrados y documentar aspectos importantes como su papel en el sistema, qué datos se intercambian (flujo de datos) y su sensibilidad. •Documentar todos los aspectos de la API, como autenticación, errores, redireccionamientos, limitación de velocidad, política de intercambio de recursos de origen cruzado (CORS) y puntos finales, incluidos sus parámetros, solicitudes y respuestas. •Generar documentación automáticamente mediante la adopción de estándares abiertos. Incluir la compilación de documentación en los procesos de CI/CD. •Hacer que la documentación de la API esté disponible para aquellos autorizados a usar la API. •Usar medidas de protección externas como los firewalls de seguridad de API para todas las versiones expuestas de la API, no solo para la versión de producción actual. •Evitar utilizar datos de producción con implementaciones de API que no sean de producción. Si esto es inevitable, estos puntos finales deberían recibir el mismo tratamiento de seguridad que los de producción. •Cuando las versiones más recientes de las API incluyan mejoras de seguridad, realizar un análisis de riesgos para tomar la decisión de las acciones de mitigación requeridas para la versión anterior: por ejemplo, si es posible respaldar las mejoras sin romper la compatibilidad hacia atrás de la API o si se necesita sacar la versión anterior rápidamente y obligar a todos los clientes a pasar a la última versión.
•Registrar todos los intentos fallidos de autenticación, acceso denegado y errores de validación de entrada. •Los registros deben escribirse usando un formato adecuado para ser consumido por una solución de administración de registros, y deben incluir suficientes detalles para identificar al actor malicioso. •Los registros deben manejarse como datos confidenciales y su integridad debe garantizarse en reposo y tránsito. •Configurar un sistema de monitoreo para monitorear continuamente la infraestructura, la red y el funcionamiento de la API. •Utilizar un Security Information and Event Management (SIEM) para administrar registros de todos los componentes de la pila API y los hosts. •Configurar paneles y alertas personalizados, lo que permite detectar actividades sospechosas y responder más rápido a los incidentes.