Reporte de Vulnerabilidad

Archivo: IPeliculaRepositorio.cs

Código Analizado:

```
?using ApiPeliculas.Modelos;

namespace ApiPeliculas.Repositorio.IRepositorio
{
    public interface IPeliculaRepositorio
    {
        //V1
        //ICollection<Pelicula> GetPeliculas();

        //V2
        ICollection<Pelicula> GetPeliculas(int pageNumber, int pageSize);
        int GetTotalPeliculas();
        ICollection<Pelicula> GetPeliculasEnCategoria(int catId);
        IEnumerable<Pelicula> BuscarPelicula(string nombre);
        Pelicula GetPelicula(int peliculaId);
        bool ExistePelicula(int id);
        bool ExistePelicula(string nombre);

        bool CrearPelicula(Pelicula pelicula);
        bool ActualizarPelicula(Pelicula pelicula);
        bool BorrarPelicula(Pelicula pelicula);
        bool Guardar();
    }
}
```

Análisis: ```html
Vulnerabilidades de Seguridad Detectadas

Análisis de Seguridad

El código proporcionado corresponde a una interfaz, y como tal, no implementa ninguna lógica. Por lo tanto, no presenta vulnerabilidades de seguridad directamente en este archivo. Las vulnerabilidades aparecerán en la clase que *implementa* esta interfaz. Sin embargo, podemos identificar areas donde la implementación podría introducir riesgos.

- **Posible Inyección SQL (en la implementación):** Métodos como `BuscarPelicula(string nombre)`, si la implementación usa Direct SQL queries (ej: concatenación de strings para construir la query), pueden ser susceptibles a inyección SQL.
  Línea aproximada donde puede estar el problema: En la *implementación* de `BuscarPelicula(string nombre)`, dentro del repositorio que implemente la interfaz `IPeliculaRepositorio` al generar la consulta a la base de datos.
  Cómo mitigarla: Usar ORM (Entity Framework Core) o stored procedures con parámetros. Sanitizar la entrada `nombre` es insuficiente y no recomendable.
- **Ataques de Denegación de Servicio (DoS) (en la implementación):** Métodos como `GetPeliculas(int pageNumber, int pageSize)` y `BuscarPelicula(string nombre)`, si no se implementan correctamente, pueden consumir demasiados recursos del servidor si `pageSize` es muy grande o `nombre` es una búsqueda muy amplia.
  Línea aproximada donde puede estar el problema: En la *implementación* de `GetPeliculas` y `BuscarPelicula`.
  Cómo mitigarla: Establecer límites razonables para `pageSize`, implementar paginación eficiente y optimizar las consultas de búsqueda. Considere usar `AsNoTracking()` en Entity Framework Core para mejorar el rendimiento de las consultas de solo lectura.
- **Exposición de información sensible (en la implementación):** La interfaz devuelve objetos `Pelicula` directamente. Es crucial que la implementación asegure que solo los datos necesarios se exponen y que la lógica de serialización no revele información interna sensible.

Línea aproximada donde puede estar el problema: En el código que llama a los métodos de la interfaz y serializa el resultado para enviarlo al cliente.
Cómo mitigarla: Usar DTOs (Data Transfer Objects) para transferir solo los datos necesarios. Implementar lógica de autorización adecuada para asegurar que solo los usuarios autorizados puedan acceder a ciertas propiedades de `Pelicula`.

- **Falta de validación de entrada (en la implementación):** Los métodos que toman IDs (`GetPelicula(int peliculaId)`, `GetPeliculasEnCategoria(int catId)`, etc.) deben validar que los IDs sean positivos y que existan en la base de datos antes de realizar operaciones.
  Línea aproximada donde puede estar el problema: Dentro de los métodos mencionados al inicio de su implementación.
  Cómo mitigarla: Agregar validaciones al principio de cada método. Retornar `NotFound` o `BadRequest` cuando el id es inválido.

Métricas de Calidad del Código

Análisis de Calidad

Si bien esta es una interfaz, podemos comentar sobre las métricas de calidad en relación a la implementación y cómo la interfaz afecta estas métricas.

- **Acoplamiento:** La interfaz reduce el acoplamiento entre el código que la usa y la implementación concreta del repositorio. Esto facilita el testing y la sustitución de implementaciones.
  Cómo mejorar: Asegurarse de que la interfaz sea lo más pequeña y cohesiva posible, exponiendo solo la funcionalidad necesaria.
- **Legibilidad:** La interfaz es relativamente legible. Los nombres de los métodos son claros y descriptivos.
  Cómo mejorar: Asegurarse de que los nombres de los parámetros sean significativos. Considerar el uso de comentarios XML para documentar la interfaz.
- **Complejidad:** La interfaz en sí misma no introduce complejidad. Sin embargo, la implementación puede ser compleja dependiendo de cómo se implementen las operaciones de base de datos y la lógica de negocio.
  Cómo mejorar: Usar patrones de diseño como el Repository Pattern y el Unit of Work Pattern para simplificar la implementación. Considerar el uso de CQRS (Command Query Responsibility Segregation) para separar las operaciones de lectura y escritura.
- **Duplicación:** La interfaz no introduce duplicación. Sin embargo, es importante evitar la duplicación en la implementación.
  Cómo mejorar: Extraer lógica común a métodos privados o clases auxiliares. Usar generics para reducir la duplicación de código en métodos que operan en diferentes tipos de entidades.

Solución Propuesta

Mejoras Sugeridas

Si bien la interfaz es correcta, las recomendaciones se orientan a la implementación del repositorio y al código que la utiliza.

1. **Implementar validaciones robustas:** Validar todas las entradas del usuario en la capa de la API y en la capa del repositorio. Usar Data Annotations y Fluent Validation. Retornar códigos de error HTTP adecuados (BadRequest, NotFound, etc.).
2. **Usar un ORM (Entity Framework Core recomendado):** Evitar la construcción manual de consultas SQL. Entity Framework Core ayuda a prevenir inyecciones SQL y simplifica la interacción con la base de datos.
3. **Implementar Paginación y Filtrado:** Controlar los parámetros `pageNumber` y `pageSize`. Implementar filtrado para las búsquedas. Considerar usar `Skip()` y `Take()` en Entity Framework Core para la paginación.
4. **Utilizar DTOs:** Crear DTOs para transferir datos entre la capa de la API y la capa del repositorio. Esto permite controlar qué datos se exponen al cliente y desacopla la API de la estructura de la base de datos.
5. **Manejo de errores centralizado:** Implementar un middleware para capturar excepciones no controladas y convertirlas en respuestas HTTP adecuadas. Usar logging para registrar errores.

6. **Implementar un sistema de autorización:** Controlar el acceso a los recursos en función de los roles y permisos del usuario. Usar ASP.NET Core Identity o un sistema de autorización personalizado.
7. **Sanitización (como defensa adicional):** Aunque no reemplaza el uso de un ORM, considerar sanitizar entradas de texto antes de usarlas en consultas (especialmente si, por alguna razón, no se puede evitar el uso de SQL directo).
8. **Asegurar la atomicidad de las operaciones:** Utilizar transacciones de base de datos para garantizar que las operaciones que modifican datos se completen de forma atómica. Usar `TransactionScope` o las transacciones integradas en Entity Framework Core.

```
```