Reporte de Vulnerabilidad

Archivo: ICategoriaRepositorio.cs

Código Analizado:

```
?using ApiPeliculas.Modelos;

namespace ApiPeliculas.Repositorio.IRepositorio
{
    public interface ICategoriaRepositorio
    {
        ICollection<Categoria> GetCategorias();
        Categoria GetCategoria(int CategoriaId);
        bool ExisteCategoria(int id);
        bool ExisteCategoria(string nombre);

        bool CrearCategoria(Categoria categoria);
        bool ActualizarCategoria(Categoria categoria);
        bool BorrarCategoria(Categoria categoria);
        bool Guardar();
    }
}
```

Análisis: ```html
Análisis de Seguridad y Calidad del Código

El código proporcionado define una interfaz para un repositorio de categorías. A simple vista, no presenta vulnerabilidades de seguridad directas en el código *mostrado*, ya que solo define la interfaz. Las vulnerabilidades probablemente residirán en la implementación de esta interfaz, en la clase que la implementa, o en el código que usa esta interfaz.

Posibles Vulnerabilidades (en la Implementación)

- **Inyección SQL:** Si el repositorio implementa consultas SQL directamente, especialmente en los métodos `ExisteCategoria(string nombre)`, `GetCategoria(int CategoriaId)`, `CrearCategoria(Categoria categoria)`, `ActualizarCategoria(Categoria categoria)` y `BorrarCategoria(Categoria categoria)`, y no sanitiza la entrada, es susceptible a inyección SQL.
  Línea Aproximada: Dentro de la implementación de los métodos mencionados.
  Mitigación: Utilizar Entity Framework Core o Dapper con parámetros, o cualquier ORM que escape automáticamente las entradas. Nunca concatenar directamente cadenas en consultas SQL.
  Ejemplo (Incorrecto): `"SELECT * FROM Categorias WHERE Nombre = '" + nombre + "'"`
  Ejemplo (Correcto - usando EF Core): `_context.Categorias.FirstOrDefault(c => c.Nombre == nombre);`
- **Ataques de Denegación de Servicio (DoS):** Si `GetCategorias()` devuelve una lista muy grande sin paginación o limitación, podría causar problemas de rendimiento y potencialmente un DoS.
  Línea Aproximada: En la implementación de `GetCategorias()`.
  Mitigación: Implementar paginación o limitar el número de resultados devueltos.
- **Condiciones de Carrera:** Si múltiples usuarios acceden concurrentemente a los métodos `CrearCategoria`, `ActualizarCategoria` o `BorrarCategoria` sin una gestión adecuada de la concurrencia, podrían ocurrir problemas de integridad de datos.
  Línea Aproximada: En la implementación de los métodos mencionados.
  Mitigación: Utilizar transacciones y mecanismos de bloqueo (optimistas u pesimistas) para garantizar la consistencia de los datos.
- **Validación Inadecuada de Datos:** Si la clase `Categoria` no tiene validaciones adecuadas (por ejemplo, usando atributos `[Required]`, `[MaxLength]`, `[Range]`) y estos no se aplican antes de guardar en la base de datos, se pueden guardar datos inválidos o inconsistentes.

Línea Aproximada: Dentro de la clase `Categoria` y antes de llamar a `Guardar()` en la implementación del repositorio.
Mitigación: Utilizar Data Annotations en el modelo `Categoria` y asegurarse de que se validan los datos antes de persistirlos.

Métricas de Calidad del Código (de la Interfaz)

- **Acoplamiento:** El acoplamiento es bajo, ya que la interfaz solo depende del modelo `Categoria` y de colecciones. Esto es bueno para la mantenibilidad.
- **Cohesión:** La cohesión es alta, ya que la interfaz define operaciones relacionadas con la gestión de categorías.
- **Legibilidad:** La interfaz es legible y fácil de entender gracias a los nombres descriptivos de los métodos.
- **Complejidad Ciclomática:** La complejidad ciclomática es baja, ya que la interfaz solo define métodos sin lógica compleja. La complejidad se trasladará a la implementación del repositorio.
- **Duplicación:** No hay duplicación en el código mostrado. Sin embargo, es crucial evitar la duplicación en la *implementación* del repositorio.

Solución Propuesta

Para abordar las posibles vulnerabilidades y mejorar la calidad del código (en la implementación), se recomienda lo siguiente:

- **Utilizar un ORM (Entity Framework Core):** Evita la necesidad de escribir consultas SQL directamente, reduciendo el riesgo de inyección SQL.
- **Implementar validación de datos:** Utilizar Data Annotations en el modelo `Categoria` y validar los datos antes de guardarlos.
- **Implementar paginación:** En `GetCategorias()`, utilizar paginación para evitar la carga excesiva de datos.
- **Manejar la concurrencia:** Utilizar transacciones y mecanismos de bloqueo para garantizar la consistencia de los datos en operaciones concurrentes.
- **Implementar logging:** Registrar eventos importantes (errores, accesos, modificaciones) para facilitar la auditoría y el diagnóstico de problemas.
- **Escribir pruebas unitarias:** Crear pruebas unitarias para verificar la correcta implementación del repositorio y detectar regresiones.
- **Considerar el patrón Repository con Unit of Work:** Para asegurar que las operaciones se realizan en una única transacción y reducir el acoplamiento al contexto de datos. La interfaz mostrada ya es el patrón Repository. Implementar Unit of Work.

```
```