Vulnerability Report

Archivo: IPeliculaRepositorio.cs

Code Analyzed:

```
?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();
    }
}
```

Analysis: ```html
Potential Security Vulnerabilities

SQL Injection (Potential)

Type: SQL Injection.
Approximate Location: Likely in the implementation of `BuscarPelicula(string nombre)`, `ExistePelicula(string nombre)`, and possibly within `GetPeliculasEnCategoria(int catId)`, `GetPelicula(int peliculaId)`, `ExistePelicula(int id)`, `ActualizarPelicula(Pelicula pelicula)`, and `BorrarPelicula(Pelicula pelicula)` depending on the underlying data access implementation (e.g., Entity Framework, raw SQL queries). The vulnerability arises if the input strings (`nombre`) or integer IDs (`catId`, `peliculaId`, `id`) are directly concatenated into SQL queries without proper sanitization or parameterization.
Description: If the `nombre` parameter in `BuscarPelicula` or `ExistePelicula` is not properly sanitized, an attacker could inject malicious SQL code into the query, potentially allowing them to read, modify, or delete data. Similarly, integer IDs used to query or update data without validation can lead to issues if, for example, the database library/ORM configuration allows insecure direct object reference (IDOR) type attacks by users spoofing an ID they don't own.
Mitigation:

- **Use Parameterized Queries or an ORM:** The most effective mitigation is to use parameterized queries or an Object-Relational Mapper (ORM) like Entity Framework Core. Parameterized queries treat the input as data rather than code, preventing SQL injection. ORMs generally handle parameterization automatically.
- **Input Validation:** Validate the `nombre` parameter to ensure it contains only allowed characters. Use a whitelist approach rather than a blacklist. For integer IDs, ensure they are positive and within expected ranges.
- **Least Privilege:** Ensure that the database user account used by the application has only the necessary permissions. This limits the damage an attacker can do if they successfully inject SQL.

Code Quality Metrics & Improvements

Code Quality Considerations

Complexity: The interface itself has low complexity. The complexity will be determined by the implementation in the concrete class. However, if `GetPeliculasEnCategoria` and `BuscarPelicula` use complex LINQ queries or raw SQL, they could become complex.
Duplication: There is no immediate duplication in the interface definition. However, duplication might exist within the concrete implementation if similar data access patterns are used across different methods. For example, the pattern of constructing a database query based on a search term is susceptible to duplication.
Readability: The interface is reasonably readable. Using clear naming conventions for the parameters would enhance readability. For example, naming the integer catId parameter to categoryId is easier to read.
Coupling: The interface `IPeliculaRepositorio` is tightly coupled to the `Pelicula` model. This is expected. However, consider using Data Transfer Objects (DTOs) if the API evolves and you want to decouple the API from the database model. For example, if the Pelicula model changes often, exposing it directly through the API can cause breaking changes for clients.
Pagination Inconsistencies There are two ways to retrieve movies: `GetPeliculas()` and `GetPeliculas(int pageNumber, int pageSize)`. This can be confusing.
Improvements:

- **Abstraction Level Consistency:** Ensure all methods perform at a similar level of abstraction. The interface defines low-level data access operations.
- **Consider DTOs:** If the API is public, use Data Transfer Objects (DTOs) to decouple the API from the database models.
- **Consistent Pagination** Either deprecate the `GetPeliculas()` and only use pagination or have `GetPeliculas()` call `GetPeliculas(int pageNumber, int pageSize)` using a default page size/number.

Proposed Solution

Recommendations and best practices

Parameterized Queries or ORM Use parameterized queries or an ORM to prevent SQL injection.
Input Validation Validate all inputs to prevent malicious data from entering the system.
Data Transfer Objects (DTOs) Decouple the API from the database models using DTOs. This helps to reduce breaking changes and improve the maintainability of the application.
Consistent Pagination Remove duplicate methods and have only one method to retrieve movies using pagination.
Unit Tests Write unit tests to ensure that the application is working as expected. This will help to prevent regressions and improve the overall quality of the application.

```
```