Reporte de Vulnerabilidad

Archivo: IUsuarioRepositorio.cs

Código Analizado:

```
?using ApiPeliculas.Modelos;
using ApiPeliculas.Modelos.Dtos;

namespace ApiPeliculas.Repositorio.IRepositorio
{
    public interface IUsuarioRepositorio
    {
        ICollection<AppUsuario> GetUsuarios();
        AppUsuario GetUsuario(string usuarioId);
        bool IsUniqueUser(string usuario);
        Task<UsuarioLoginRespuestaDto> Login(UsuarioLoginDto usuarioLoginDto);
        Task<UsuarioDatosDto> Registro(UsuarioRegistroDto usuarioRegistroDto);
    }
}
```

Análisis: ```html
Vulnerabilidades y Calidad del Código

Análisis

El código proporcionado es una interfaz (`IUsuarioRepositorio`) en C#. Como tal, no contiene implementación, por lo que las vulnerabilidades de seguridad a nivel de código directo son bajas. Sin embargo, la *ausencia* de validación de entradas y manejo de excepciones en la *implementación* de esta interfaz, así como un diseño de la API que exponga datos sensibles, podrían introducir riesgos significativos.
Vulnerabilidades Potenciales (en la Implementación):

- **Inyección SQL (si `usuarioId` no se sanitiza en la implementación de `GetUsuario`):** Si la implementación de `GetUsuario` usa el `usuarioId` directamente en una consulta SQL sin parametrización, un atacante podría inyectar código SQL malicioso. Imagina una implementación que haga algo como: `"SELECT * FROM Usuarios WHERE Id = '" + usuarioId + "'"`. En este caso, la variable `usuarioId` podría ser algo como `"1' OR '1'='1"` y dejaría expuestos todos los datos de la tabla.
- **Fuerza Bruta/Ataque de Diccionario (en la Implementación de `Login`):** Si no hay limitación de intentos fallidos de inicio de sesión, un atacante podría intentar adivinar las contraseñas mediante fuerza bruta o un ataque de diccionario.
- **Exposición de Información Sensible (a través de los DTOs):** Si los DTOs (`UsuarioLoginRespuestaDto`, `UsuarioDatosDto`) contienen información innecesaria y sensible (por ejemplo, el hash de la contraseña o detalles internos del sistema), podría facilitar ataques.
- **Falta de Validación de Entradas (en la Implementación de `Registro` y `Login`):** Si no se validan correctamente los datos de entrada (por ejemplo, formato del email, longitud de la contraseña, etc.), se podrían crear usuarios con datos inválidos o explotar vulnerabilidades de desbordamiento de buffer (menos común con C# moderno, pero aún posible en casos extremos).
- **Falta de Manejo de Excepciones:** Si las implementaciones de los métodos no manejan las excepciones correctamente (por ejemplo, al acceder a la base de datos), se podría revelar información sensible a través de mensajes de error.

Métricas de Calidad del Código (de la Interfaz):

- **Complejidad:** Baja. La interfaz es simple y define un conjunto reducido de operaciones.
- **Duplicación:** No aplicable directamente a una interfaz. La duplicación sería un problema en las clases que *implementan* esta interfaz.
- **Legibilidad:** Buena. Los nombres de los métodos y parámetros son descriptivos.

- **Acoplamiento:** El acoplamiento es bajo, ya que la interfaz solo define un contrato y no depende de implementaciones concretas.

Solución Propuesta

Mitigación y Mejoras

Mitigación de Vulnerabilidades (en la Implementación):

- **Inyección SQL:** Usar siempre consultas parametrizadas o ORMs (Entity Framework Core) para evitar la inyección SQL. Nunca concatenar cadenas directamente en una consulta.
- **Fuerza Bruta:** Implementar mecanismos de limitación de velocidad (rate limiting) y bloqueo de cuentas después de un número determinado de intentos fallidos de inicio de sesión. Considerar el uso de CAPTCHA.
- **Exposición de Información Sensible:** Diseñar cuidadosamente los DTOs para que solo contengan la información necesaria para la funcionalidad requerida. Evitar exponer datos internos o sensibles.
- **Validación de Entradas:** Validar *todas* las entradas del usuario en el backend (y también en el frontend para una mejor experiencia de usuario). Utilizar anotaciones de validación en los DTOs y/o implementar lógica de validación personalizada.
- **Manejo de Excepciones:** Implementar bloques `try-catch` para capturar excepciones y registrarlas adecuadamente. Evitar revelar información sensible en los mensajes de error. Utilizar logging robusto.
- **Autenticación y Autorización Robusta:** Implementar un sistema de autenticación y autorización robusto (por ejemplo, usando JWT y roles) para controlar el acceso a los recursos.

Mejoras de Calidad del Código (Consideraciones para la Implementación):

- **Single Responsibility Principle (SRP):** Asegurarse de que cada clase tenga una única responsabilidad bien definida. Separar la lógica de autenticación, autorización, validación y acceso a datos en clases separadas.
- **Dependency Injection (DI):** Usar la inyección de dependencias para desacoplar las clases y facilitar las pruebas unitarias.
- **Testing:** Escribir pruebas unitarias para verificar que la implementación de la interfaz funciona correctamente y que los métodos manejan correctamente los casos de borde y los errores.
- **Código Limpio:** Seguir las convenciones de codificación de C# y escribir código legible y fácil de mantener. Usar nombres descriptivos para variables, métodos y clases. Escribir comentarios claros y concisos. Evitar código duplicado.

La interfaz actual es un buen punto de partida. Las mejoras más significativas se lograrán al implementar esta interfaz de manera segura y siguiendo las mejores prácticas de desarrollo de software.

```
```