Sesión 11 : Categorización de Vulnerabilidades

Introducción a la seguridad de aplicaciones y buenas prácticas en codificación

Mauricio Sotelo

- Ingeniero de Seguridad Nivel III
- Seguidor de las Buenas Prácticas, Marcos de Trabajo y Estándares de Seguridad.
- Amo las actividades al aire libre y hacer todo con música de fondo
- https://calendly.com/mauricio_sotelo

Recomendaciones Importantes

Identifícate en Zoom usando tu nombre y apellido.

Silencia tu micrófono durante el curso.

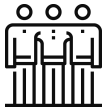
Utiliza el chat para hacer preguntas en la sección asignada para ello.

Centra tus preguntas en el tema presentado.

Mantén tu cámara encendida durante toda la sesión.

Academy Codigo de Conducta

Sé respetuoso, no hay preguntas o ideas malas.



Sé empático y paciente.

Sé cuidadoso con las palabras que eliges.

Objetivo

Al final de esta sesión podrás:

- Entender en qué consiste y para qué el categorizar las vulnerabilidades dentro de la arquitectura de una aplicación.

Tabla de Contenidos

Código Legible y Mantenible

EQUIFAX DATA BREACH

Se estima que el ataque costó más de
$ 1,300,000,000 de dólares

¿Qué hace este código en C#?

```
public class TurboCalculator3000{
public string turbocalculator3000(double var, string var2){
double cheesecake=3.141592653589793846;
double magnificentresult=cheesecake*var*var;
return var2+magnificentresult;}}
```

Un ejemplo de que **NO** hacer

Buenas prácticas

Legibilidad del código

- Usar espacios hace mucho más fácil leer cada una de las líneas.
- Usar tabulaciones ayuda a tener una estructura clara en el código.
- Identificar las líneas de código que están relacionadas entre sí y mantenerlas juntas.
- Separar tus fragmentos de líneas relacionadas, aunque sean de la parte de la misma función.

Comprensión

- Los nombres de cada variable, función, etc... tienen que describir su uso.
- Los nombres de cada variable, función, etc... tienen que ser lo más sencillos posibles.
- No enumerar objetos (por ejemplo, `var1`, `var2`, `var3`, ...)
- Evita los chistes en los nombres de los objetos.

Buenas prácticas

Usar convenciones

- Define convenciones para cada objeto, clases, variables, constantes.
- Evita abreviar de manera excesiva.
- Usa consistentemente las mayúsculas y minúsculas.
- Siempre sigue las convenciones definidas.

Declarar variables y funciones con sentido

- Facilitará saber qué tipo de objeto es.
- Identificará claramente para qué sirve.
- Mostrará de manera clara su diferencia con los demás objetos.

Código estandarizado en C#

```
using System;

public class CircleAreaCalculator
{
    // Evitar números mágicos
    private const double PI = 3.14159265358979323846;

    public string GetAreaWithText(double radius, string prefixText)
    {
        // Inicializar variables en la declaración
        double area = PI * Math.Pow(radius, 2);

        // Construir el resultado del texto cerca de su uso
        string resultText = prefixText + area.ToString();

        return resultText;
    }
}
```

Comenta tu código

"Al igual que una buena historia lleva notas al margen que explican las partes más complejas, nuestro código también necesita esa claridad, no son un lujo, son una necesidad. Son el puente que nos ayuda a entender y a mantener viva la esencia de lo que estamos creando. Sin ellos, nos arriesgamos a perder no solo el entendimiento, sino la intención detrás de cada línea."

Steve Jobs

Comentarios

Agrega comentarios descriptivos:

- Describe qué hace una sección de código, no cómo lo hace, a menos que el "cómo" sea particularmente complejo o ingenioso.

Ejemplo:

```
// Calcula el área de un círculo.
double area = Math.PI * radius * radius;
```

Evita comentarios obvios o redundantes:

- No uses comentarios para cosas que son obvias para cualquier programador que lea el código.

Ejemplo:

```
// Incorrecto:
// Incrementa el contador en 1
counter++;

// Correcto:
counter++;
```

Comentarios

Documenta funciones y métodos:

- Usa comentarios para documentar funciones, explicando sus entradas, salidas y propósito.

Ejemplo:

```
/// <summary>
/// Calcula el área de un círculo.
/// </summary>
/// <param name="radius">Radio del círculo.</param>
/// <returns>Área del círculo.</returns>
public double CalculateCircleArea(double radius)
{
    return Math.PI * radius * radius;
}
```

Comentarios

Comenta para explicar decisiones:

- Si tomaste una decisión de diseño o implementación basada en un requisito específico, problema o solución, documenta esa decisión para futura referencia.

Ejemplo:

```
// Usando burbuja por requerimiento del cliente
BubbleSort(items);
```

Comentarios

Usa Comentarios para "TO-DO" y "FIX-ME":

- Si dejas algo sin terminar o identificas un posible error que aún no puedes solucionar, utiliza comentarios "TO-DO" o "FIX-ME" para marcarlo.

Ejemplo:

```
// TODO: Implementar manejo de errores aquí
// FIXME: Esta función no retorna correctamente si la lista está vacía
```

Comentarios

Usa comentarios para secciones de código:

- Si tienes bloques de código que realizan una función específica dentro de una función más grande, usa comentarios para delimitar y describir esas secciones.

Ejemplo:

```
// Inicio de la sección de inicialización
...
// Fin de la sección de inicialización

// Inicio de la sección de procesamiento
...
// Fin de la sección de procesamiento
```

Control de Versiones

Control de Versiones

1. **Importancia del control de versiones.**
    - **Historial de cambios:** Puedes revisar, revertir o comparar versiones anteriores del código.
    - **Colaboración:** Varios desarrolladores pueden trabajar en diferentes características o áreas del código sin interferir entre sí.
    - **Backup:** En caso de cualquier fallo o pérdida, el repositorio sirve como una copia de seguridad.
2. **Elección del sistema de control de versiones.**

    El sistema más popular en la actualidad es Git, pero hay otros como SVN, Mercurial, etc. La elección depende de las preferencias del equipo y las necesidades del proyecto.

Control de Versiones

3. **Principios básicos:**
    - **Commits:** Son conjuntos de cambios en el código. Deben ser frecuentes, y cada uno debe tener un mensaje descriptivo que indique el propósito del cambio.
    - **Branching:** Permite a los desarrolladores trabajar en características o correcciones específicas sin afectar el código principal.
    - **Merging:** Una vez completado el trabajo en una rama, se pueden combinar esos cambios en la rama principal.
4. **Mensajes de commits descriptivos:**
    - Un buen mensaje de commit proporciona contexto sobre qué se hizo y por qué.
5. **Uso de repositorios remotos:**
    - Sitios como GitHub, GitLab y Bitbucket permiten almacenar copias del código en la nube, facilitando la colaboración y el acceso al código desde cualquier lugar.

Control de Versiones

6. **Buena práctica: Pull antes de Push**
    - Antes de enviar tus cambios al repositorio remoto, es recomendable obtener los últimos cambios del equipo para evitar conflictos.
7. **Manejo de conflictos**
    - A veces, dos o más desarrolladores pueden modificar el mismo segmento de código, lo que resulta en un conflicto. Es esencial saber cómo resolver estos conflictos manualmente.
8. **Tags y releases**
    - Al alcanzar puntos importantes en el desarrollo, como el lanzamiento de una versión, es útil marcar el código con un tag para referencia futura.

Mejores prácticas en control de versiones

1. Principios Básicos.

Commits

Ejemplo 1: Realizar un cambio en el archivo README.md y hacer commit.

```
echo "Descripción del proyecto" >> README.md
git add README.md
git commit -m "Añadida la descripción al archivo README.md"
```

Ejemplo 2: Corregir un error en el archivo index.html.

```
# Supongamos que corregiste el error en tu editor
git add index.html
git commit -m "Corrección de error en el botón de inicio"
```

Ejemplo 3: Actualizar la documentación en la carpeta /docs.

```
# Supongamos que hiciste las actualizaciones
git add docs/*
git commit -m "Actualización de la documentación para la versión 2.0"
```

Mejores prácticas en control de versiones

1. Principios Básicos.

Branching

Ejemplo 1: Crear una rama para una nueva funcionalidad.

```
git branch feature/nueva-funcionalidad
git checkout feature/nueva-funcionalidad
```

Ejemplo 2: Crear una rama para corregir un bug específico.

```
git branch fix/error-login
git checkout fix/error-login
```

Ejemplo 3: Este mensaje indica que la documentación ha sido actualizada para reflejar cambios recientes en la API, incluyendo nuevos endpoints.

```
git branch docs/actualización
git checkout docs/actualización
```

Mejores prácticas en control de versiones

1. Principios Básicos.

Merging

Ejemplo 1: Combinar los cambios de la rama de funcionalidad en la rama principal.

```
git checkout main
git merge feature/nueva-funcionalidad
```

Ejemplo 2: Combinar correcciones de errores en la rama principal.

```
git checkout main
git merge fix/error-login
```

Ejemplo 3: Combinar actualizaciones de documentación.

```
git checkout main
git merge docs/actualización
```

Mejores prácticas en control de versiones

2. Mensajes de Commit Descriptivos.

Ejemplo 1: Este mensaje indica que se ha implementado una funcionalidad para que la interfaz de usuario soporte varios idiomas.

```
git checkout main
git merge feature/nueva-funcionalidad
```

Ejemplo 2: Este mensaje señala la corrección de un error específico donde el texto se desbordaba en una sección determinada.

```
git checkout main
git merge fix/error-login
```

Ejemplo 3: Combinar actualizaciones de documentación.

```
git checkout main
git merge docs/actualización
```

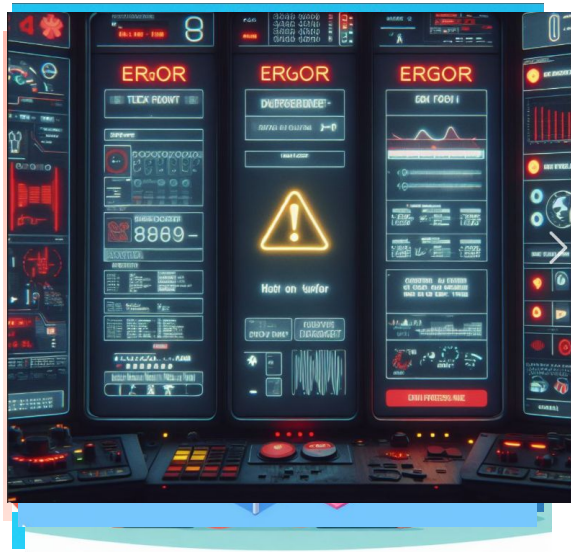
Mejores prácticas en control de versiones

- Haz commits por funciones, no por jornada de trabajo (podrán ser varios commits por día)
- Entre menos código haya en cada commit, mejor
- NUNCA dejes secretos en el código
- Utilizar los gestores de secretos proporcionados por los sistemas de control de versiones
- En caso de subir un secreto, borra el historial de cambios, no solo hagas un nuevo commit sin secretos
- Protege cada uno de los ambientes (por medio de autorización para poder enviar cambios)
- Procura siempre usar repositorios privados
- No utilices tu ambiente personal para trabajo

Manejo de Errores

6 Estrategias para el manejo de errores

1. Graceful Degradation (Degradación Elegante)
2. Progressive Enhancement (Mejora Progresiva)
3. Logging and Monitoring (Registro y Monitoreo)
4. User Feedback (Retroalimentación del Usuario)
5. Automated Testing (Pruebas Automatizadas)
6. Clear Error Messaging (Mensajes de Error Claros)



Buenas prácticas

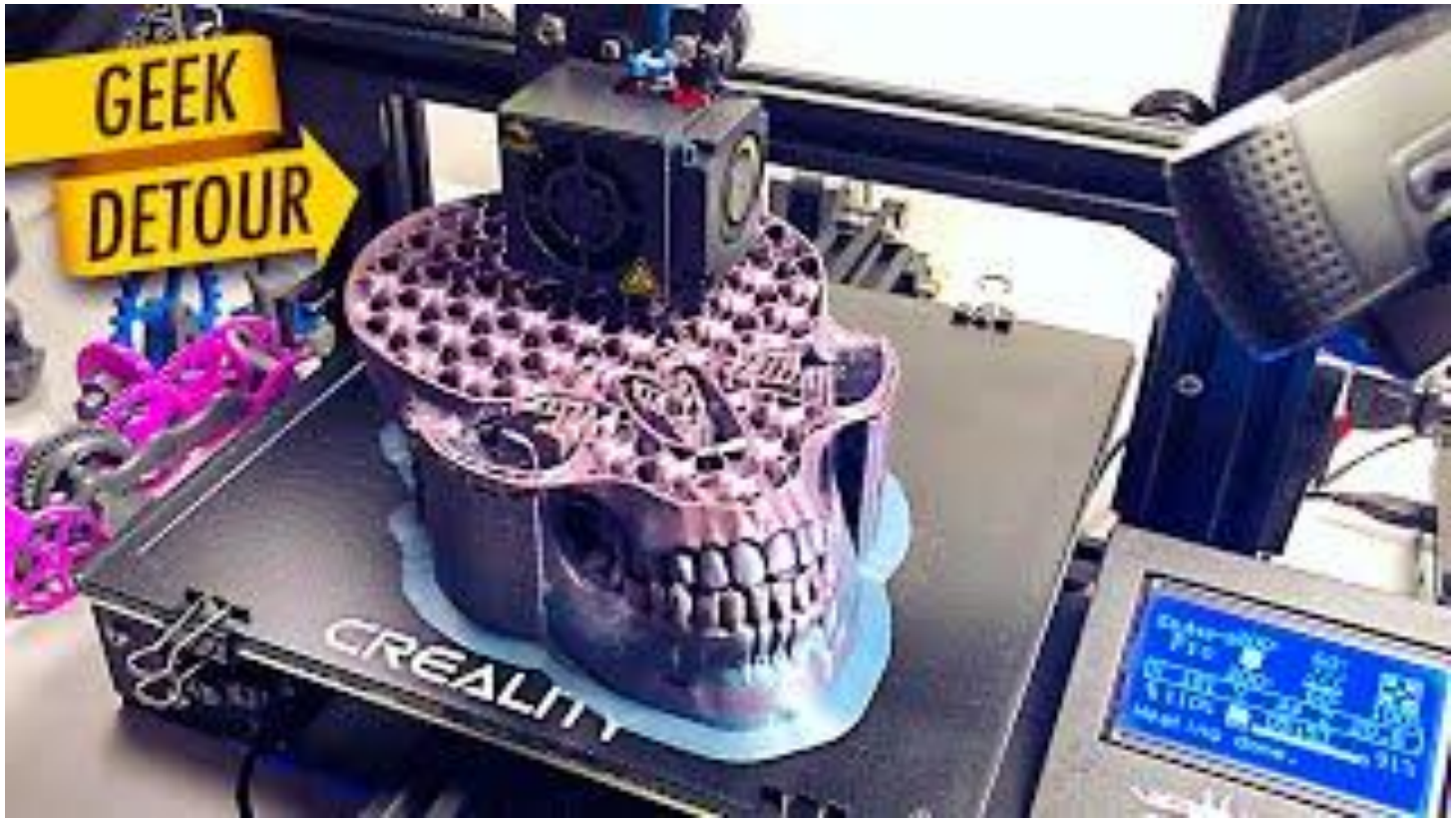
- Utiliza las funciones de manejo de excepciones (como try; catch;) de manera estructurada; es decir, que sea claro para qué fue cada catch.
- Evita exponer código y errores genéricos en los mensajes de error.
- Evita a toda costa exponer datos del funcionamiento del sistema y su arquitectura en los mensajes de error.
- Utiliza siempre errores personalizados en los mensajes mostrados al cliente.
- Centraliza tu código de manejo de errores
- Sanitiza los errores antes de almacenarlos en un log
- Considera que alguien tiene que leer los logs, mantenlos lo más estructurados posibles.
- Los logs tienen que ser útiles, manda la mayor información necesaria, evita la innecesaria.
- Maneja distintos archivos de logs por propósito (seguridad, debugging, soporte al usuario).

Descanso

5 minutos.

Test Driven Development

Qué es Test Driven Development ?



Ciclo Red

```
using NUnit.Framework;

public class FactorialCalculator
{
    // Método para calcular el factorial
    public static int Factorial(int n)
    {
        // Tu código irá aquí
        return 0; // Valor temporal para compilar
    }
}

[TestFixture]
public class TestFactorial
{
    [Test]
    public void TestFactorialOfO()
    {
        // Placeholder value, fallará al inicio
        Assert.AreEqual(1, FactorialCalculator.Factorial(0));
    }
}
```

Al ejecutar la prueba, falla, como era esperado.

```
TestFactorial.TestFactorialOfO: Failed
Expected: 1
But was:  0
```

Nota: Para usar NUnit en C#, necesitarás instalar el paquete NUnit a través de NuGet y, probablemente, también el paquete NUnit3TestAdapter para que las pruebas sean descubiertas y ejecutadas.

Ciclo Green

```
using NUnit.Framework;

public class FactorialCalculator
{
    // Método para calcular el factorial
    public static int Factorial(int n)
    {
        // Corrección: Añadido código para devolver 1 cuando n es 0.
        if (n == 0)
            return 1;
        else
            return n * Factorial(n - 1);
    }
}

[TestFixture]
public class TestFactorial
{
    [Test]
    public void TestFactorialOfO()
    {
        Assert.AreEqual(1, FactorialCalculator.Factorial(0));
    }
}
```

Al ejecutar la prueba, es exitosa.

```
TestFactorial.TestFactorialOfO: Passed
```

Ciclo Refactor

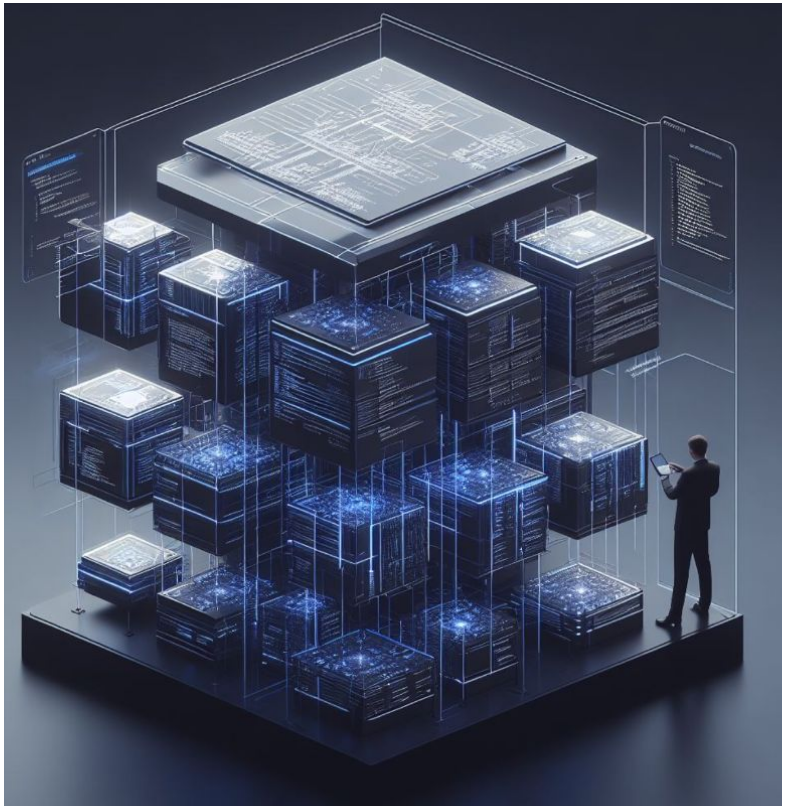
```
using NUnit.Framework;

public class FactorialCalculator
{
    // Método que devuelve el factorial de un número entero n
    public static int Factorial(int n)
    {
        // Devuelve 1 si n es 0, de lo contrario, calcula el factorial
        if (n == 0)
            return 1;

        return n * Factorial(n - 1);
    }
}

[TestFixture]
public class TestFactorial
{
    [Test]
    public void TestFactorialOfO()
    {
        // La prueba espera que el factorial de 0 sea 1
        Assert.AreEqual(1, FactorialCalculator.Factorial(0));
    }
}
```

Al ejecutar la prueba, sigue siendo exitosa.

```
TestFactorial.TestFactorialOfO: Passed
```

Programación Modular

Programación por módulos



Componentes de la modularidad

Encapsulación

- Requiere la menor cantidad de parámetros posibles
- Regresa solo la información necesaria
- Incluye el manejo de errores

Asbtracción

- Enfócate en lo que tu módulo hará, no el cómo.

Separación de los temas

- Cada módulo tiene que tener una sola responsabilidad distinta a las demás.

Acoplamiento bajo

- No tiene que haber dependencias entre los módulos
- Los usuarios del módulo tienen que decidir si relacionar sus funcionalidades o no

Alta cohesión

- Mantén las funcionalidades similares en el mismo módulo
- La elección de funciones puede ser un parámetro

Ejemplo de módulo para inserción de datos

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data.SqlClient;

public class DatabaseHelper
{
    private SqlConnection _connection;

    // Constructor que inicializa la conexión a la base de datos.
    public DatabaseHelper(string connectionString)
    {
        _connection = new SqlConnection(connectionString);
    }
```

Ejemplo de módulo para inserción de datos

```
   /// <summary>
    /// Inserta datos en una tabla específica.
    /// </summary>
    /// <param name="data">Diccionario con los datos a insertar.</param>
    /// <param name="table">Nombre de la tabla en la que se insertarán los datos.</param>
    public void InsertData(Dictionary<string, string> data, string table)
    {

        var columns = data.Keys;
        var values = data.Values.Select(value => $"'{value}'"); // Se agregan las comillas para
SQL.
        // Construye el query de inserción.
        string query = $"INSERT INTO {table} ({string.Join(",", columns)}) VALUES
({string.Join(",", values)})";

        // Ejecuta el query.
        ExecuteQuery(query);
    }
```

Ejemplo de módulo para inserción de datos

```
    /// <summary>
    /// Ejecuta un query SQL en la base de datos.
    /// </summary>
    /// <param name="query">El query a ejecutar.</param>
    private void ExecuteQuery(string query)
    {
        using SqlCommand cmd = new SqlCommand(query, _connection);
        _connection.Open();
        cmd.ExecuteNonQuery();
        _connection.Close();
    }
}
```

Componentes Open Source

Qué son los componentes Open-Source?

- Los componentes Open-Source son colecciones de funciones desarrolladas por alguien más.
- Ese “alguien más” normalmente es una comunidad.
- Estos componentes están presentes en muchos desarrollos.
- Esa comunidad hace mejoras continuas al componente.
- Sus mejoras son relacionadas a problemas de seguridad.
- Te ahorran mucho trabajo al no tener que programar las funciones que contienen.
- Son distribuidas con una licencia por medio de la cual aceptas respetar su uso y sus reglas.

Licencias Open Source

- Son documentos legales con reglas sobre el uso de los componentes Open Source.
- Incluyen términos y condiciones de la distribución del componente y del desarrollo que los usa.
- Especifican reglas de uso muy específicas..
- Normalmente tiene reglas que te obligan a informar al usuario final sobre el uso del componente.

Ejemplos

- https://developer.spotify.com/third-party-licenses
- https://www.meta.com/legal/portal/third-party-notices

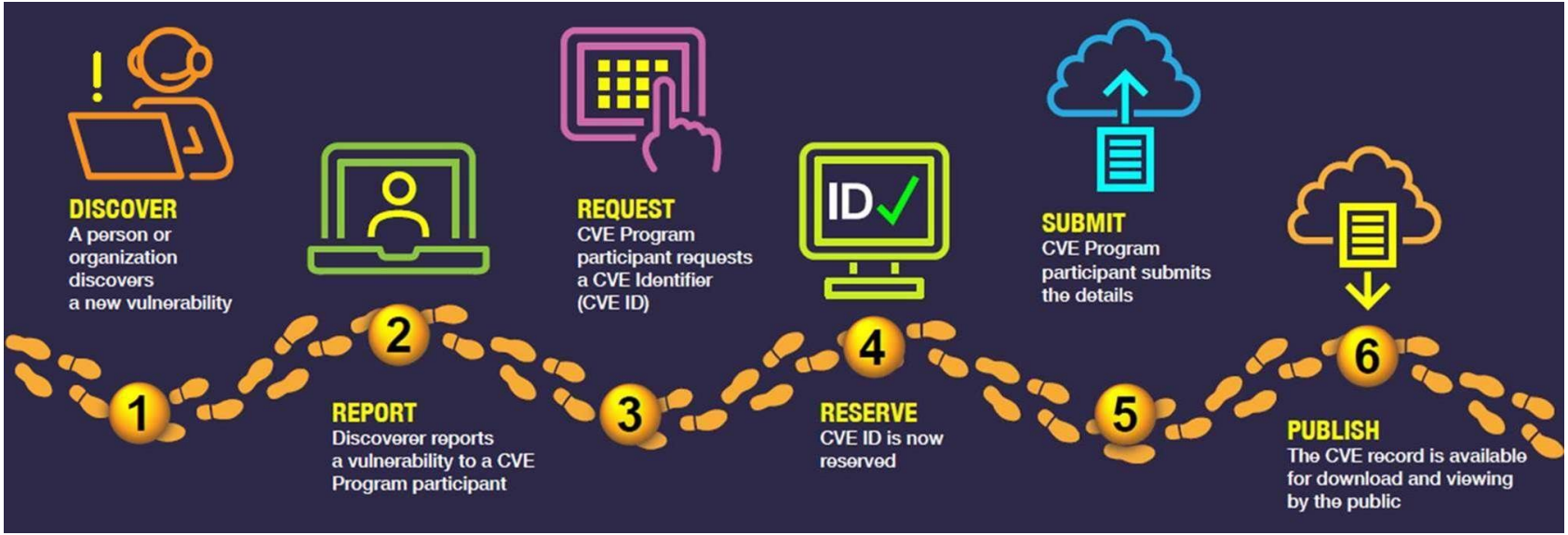
GPL (General Public License)

- Copyleft.
- Licencia recíproca.
- Libre de restricciones.

LGPL (Lesser General Public License)

- Permite el enlace dinámico y estático con programas no libres.
- Permite la modificación y redistribución del código LGPL sin modificaciones.
- Permite la redistribución del código LGPL con o sin código propietario.

Vulnerabilidades en los componentes



https://www.cve.org/

Herramientas útiles

OWASP Dependency Check

- Herramienta de línea de comandos
- Busca vulnerabilidades en los componentes
- Informa acerca de versiones seguras
- Informa acerca de sus licencias

Fossology

- Servicio web
- Requiere instalación y configuración
- Analiza las licencias utilizadas por los componentes

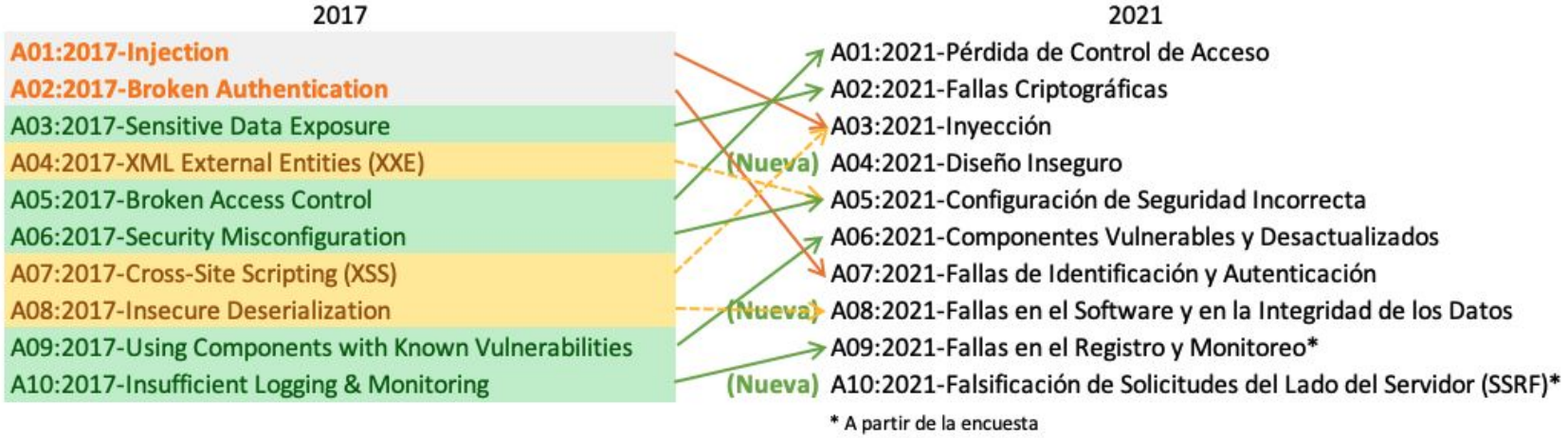
Dependabot

- Herramienta incluida en GitHub
- Fácilmente automatizable
- Informa acerca de componentes vulnerables en tu repositorio

Seguridad por Defecto

TOP 10

Qué ha cambiado en el Top 10 de 2021



Recapitulando

- Desarrolla todo tu código como módulos.
- Recuerda considerar cada módulo por lo que hace y que tiene que ser reutilizable
- Manten tus módulos legibles y entendibles
- Usa Test Driven Development para hacer tus módulos a prueba de fallas
- Controla todos los errores y excepciones dentro de tus módulos
- Haz un módulo que sanitice todas las entradas al log y los almacene
- Valida las licencias de los componentes open-source
- Siempre actualiza tus componentes y analízalos constantemente
- Comparte los módulos con tu equipo de trabajo

¿Preguntas?

Gracias!