

# Pautas para codificación segura



# Fundamentos

➤ Conjunto de buenas prácticas a la hora de conseguir generar código seguro, que evite efectos indeseados ante posibles ataques externos.

➤ Se organizan en nueve secciones:


- Denegación de servicio
- Confidencialidad
- Inyección e inclusión
- Accesibilidad
- Validación de datos
- Mutabilidad
- Construcción de objetos
- Serialización
- Control de acceso

# Denegación de servicio

- Debemos comprobar la entrada de datos en un sistema para evitar que cause un consumo excesivo de recursos y que pueda afectar a la CPU o la memoria
- Liberar recursos en todas las situaciones
- Definir un sistema robusto de gestión de errores y excepciones

```
try(resource1;resource2){  
  
}  
catch...
```

# Confidencialidad

- Los datos confidenciales solo deberían ser visibles dentro de un contexto limitado
  - Eliminar información sensible de volcados de error
  - No realizar registros de log de información sensible
  - Considerar eliminar información sensible de la memoria después de su uso
- 

# Inyección

➤ Evitar la inyección de SQL mediante el uso de PreparedStatement. Utilizar parámetros en la instrucción SQL , en lugar de concatenar valores:

## Incorrecto

```
String sql="select * from alumnos where nombre='"+nombre+"'";  
Statement st=con.createStatement();  
st.execute(sql);
```


## Correcto

```
String sql="select * from alumnos where nombre=?";  
PreparedStatement st=con.prepareStatement(sql);  
st.setParameter(1,nombre);  
st.execute();
```

➤ Evitar inyección de valores excepcionales en punto flotante:

```
if (Double.isNaN(untrusted_double_value)) {  
    // specific action for non-number case  
}  
  
if (Double.isInfinite(untrusted_double_value)){  
    // specific action for infinite case  
}  
  
// normal processing starts here
```

# Accesibilidad

- Limitar la accesibilidad de clases, métodos y atributos al mínimo requerido por nuestro código
  - Limitar la extensibilidad de clases y métodos. Para evitar herencias y sobrescrituras maliciosas, debemos definirlos como final. Si una clase debe usar otra, preferible composición a herencia
  - Evitar cambios en una superclase para que las subclasses no se vean afectadas
- 

# Validación de datos

- Validar siempre datos de entrada, a fin de evitar que puedan alterar el flujo de nuestro código o corromper el estado de objetos:

```
public void process(String name, int value){  
    if(name.equals("")){...}  
  
    if(value < 0 && value > 100){...}  
}
```

- No se debe confiar solo en la validación del lado cliente
- Definir envoltorios alrededor de métodos nativos

# Revisión conceptos

**Dado el siguiente código, indica que acciones deberíamos realizar para que se ajustase a las pautas de codificación segura**

```
public class MyClass{  
    int data;  
    public int getData(){  
        return data;  
    }  
    void setData(String f){  
        data=f;  
    }  
}
```

- A. Definir data como privado
- B. Definir setData como público
- C. definir getData como privado
- D. Validar el dato de entrada en setData

**Respuesta**

Las respuestas son A y D. Los atributos deben ser siempre privados (no los métodos de acceso) y los valores de entrada deberían validarse antes de asignarlos



# Mutabilidad

➤ Crear copias de valores de salida mutables. Cuando un método devuelve un dato que es mutable, preferible devolver una copia:

```
public class CopyOutput {  
    private final java.util.Date date;  
    public java.util.Date getDate(){  
        return (java.util.Date)date.clone();  
    }  
}
```

➤ No exponer colecciones modificables


➤ Definir atributos públicos estáticos como finales

# Construcción de objetos

- Evitar constructores públicos de clases sensibles
- No establecer valores de atributos en el constructor, hasta que todas las comprobaciones se hayan realizado
- Evitar desde los constructores llamadas a métodos que pueden ser sobrescritos

```
public class Test{  
    private int data;  
    public Test(int x){  
        if(x > 10){...}  
        method(); // llamada no segura  
    }  
    public void method(){..  
}
```

validación de  
parámetros  
antes de  
asignación



# Serialización

- Evitar la serialización de clases sensibles
- Proteger datos sensibles durante la serialización
- Evitar serializar determinados datos durante el proceso de serialización, definiéndolos como transient:

```
public class Profile implements Serializable {  
    private transient String password;  
}
```

- Ser consciente de que, durante el proceso de deserialización, se crea un objeto de la clase sin invocar a constructor alguno

# Control de acceso

## ➤ Invocaciones seguras mediante doPrivileged:

```
public class LibClass {  
  
    private static final String OPTIONS = "xx.lib.options";  
  
    public static String getOptions() {  
        return AccessController.doPrivileged(  
            new PrivilegedAction<String>(){  
                public String run() {  
                    // this is checked by SecurityManager  
                    return System.getProperty(OPTIONS);  
                }  
            }  
        );  
    }  
}
```

El valor debe estar "controlado", no permitir cualquier entrada

Las llamadas a propiedades de sistema se realizan desde doPrivileged para que, quien utilice la clase acceda con los permisos de la misma

# Revisión conceptos



¿Qué deberíamos modificar en el siguiente código para que cumpliera con las pautas de seguridad?

```
public String getData(String name) {  
    return AccessController.doPrivileged(  
        new PrivilegedAction<String>() {  
            public String run() {  
                return System.getProperty(name);  
            }  
        }  
    );  
}
```

**Respuesta**

Dentro del código privilegiado, utilizamos como nombre de propiedad un valor recibido como parámetro en el método, lo que permite acceder a cualquier propiedad desde el exterior.

No se debe utilizar un valor de entrada como nombre de propiedad o, si se hace, realizar una verificación del mismo