

# Práctica 5: Condiciones de Carrera

## Seguridad Informática

Pedro Allué Tamargo (758267)

Juan José Tambo Tambo (755742)

13 de diciembre de 2020

### Índice

<b>1. Programa vulnerable</b>	<b>1</b>
<b>2. Explotación de la vulnerabilidad</b>	<b>1</b>
2.1. ¿Cuáles son los recursos compartidos? ¿Quién los comparte? ¿Cuál es la ventana temporal de la vulnerabilidad? . . . . .	1
<b>3. Corrección de las <i>API</i></b>	<b>2</b>
3.1. Creación del método <code>sacarDinero</code> de <i>Cuenta.java</i> . . . . .	2
3.2. ¿Hay otras <i>API</i> vulnerables a la condición de carrera? . . . . .	2
3.3. Implementar una versión de <i>CompraSegura.java</i> . . . . .	2
<b>4. Anexo 1: Códigos</b>	<b>3</b>
4.1. Código vulnerable <i>Compra.java</i> . . . . .	3
4.2. Código de la implementación de <code>sacarDinero()</code> . . . . .	4
4.3. Código seguro <code>compraSegura</code> . . . . .	5

## 1. Programa vulnerable

Se ha implementado el programa *Compra.java* (Código 4.1) para poder explotar la vulnerabilidad. Se puede observar que el programa sigue el comportamiento esperado pero que existe una variable (*saldoActual*) que almacena el saldo al inicio. Esta variable será clave para explotar la vulnerabilidad de condiciones de carrera.

## 2. Explotación de la vulnerabilidad

### 2.1. ¿Cuáles son los recursos compartidos? ¿Quién los comparte? ¿Cuál es la ventana temporal de la vulnerabilidad?

- Los recursos compartidos son los ficheros: *carrito.txt* y *cuenta.txt*
- Estos recursos son compartidos por las distintas instancias de *Cuenta* y *Carrito*.
- La ventana temporal es el tiempo en el que un recurso puede ser modificado por otra secuencia de código que se ejecuta concurrentemente. Por lo tanto, en el código implementado la ventana temporal sería cualquier ejecución mientras haya un programa esperando la entrada de información por parte del usuario. En este caso el programa esperando la entrada del usuario habrá almacenado el saldo disponible cuando comenzó su ejecución. La variable culpable de esta vulnerabilidad es *saldoActual* (línea 20).

No obstante, si no hubiese existido esta variable se hubiera podido explotar la vulnerabilidad con una ventana temporal más reducida en el momento de la escritura del saldo en el fichero *cuenta.txt*. Esto es así ya que al introducir un retraso entre la lectura del saldo y su escritura en el fichero existe la posibilidad de que otro programa lea el saldo no actualizado.

Para explotar la vulnerabilidad se han ejecutado concurrentemente dos instancias de *Compra.java* (Figura 1). Al inicio ambos programas almacenan el saldo actual proporcionado por la *API* de *Cuenta*. Ahora, ambos programas pedirán la compra de un coche. Al ejecutarse de manera concurrente e independientemente del orden (ya que ambos han almacenado el saldo al principio de la ejecución) se comprarán dos coches. Se puede observar el saldo resultante y el carro de la compra en la Figura 2.

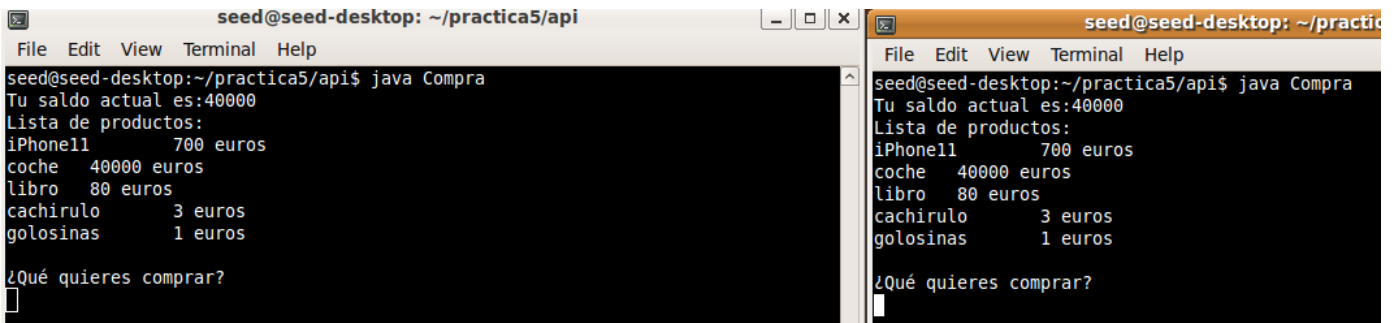


Figura 1: Captura de pantalla de la ejecución concurrente de dos instancias de *Compra.java*

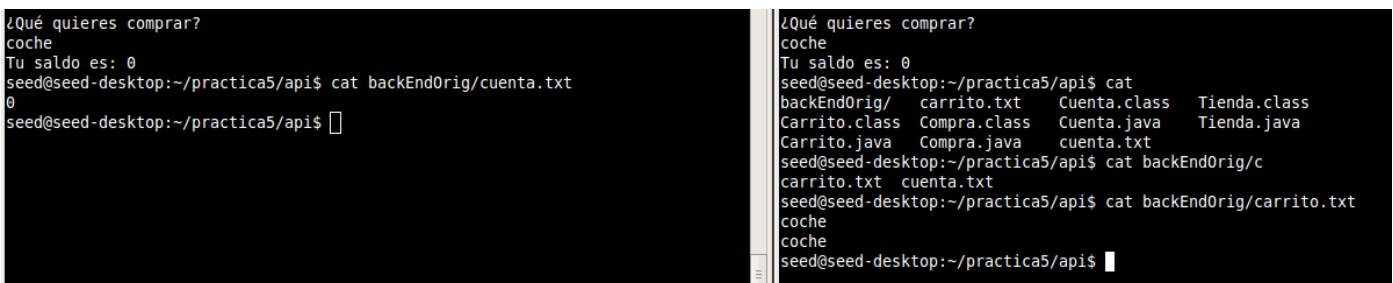


Figura 2: Captura de pantalla de los resultados de ejecución concurrente de dos instancias de *Compra.java*

### 3. Corrección de las *API*

#### 3.1. Creación del método `sacarDinero` de *Cuenta.java*

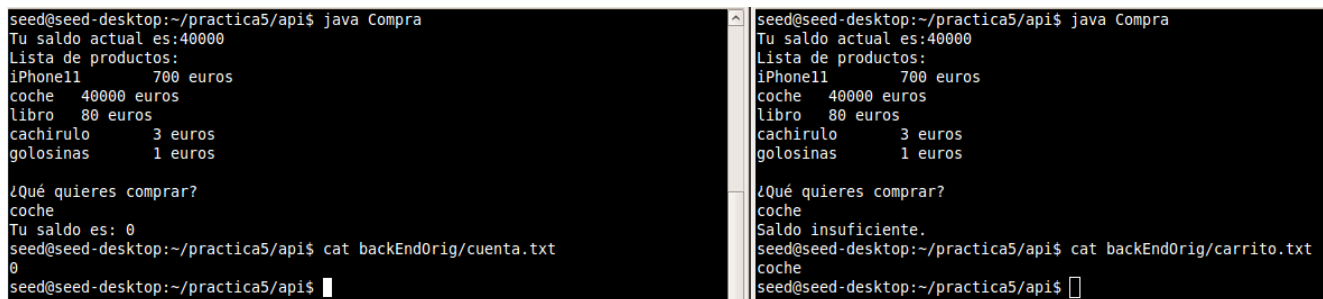
Para impedir la vulnerabilidad de condición de carrera se va a implementar una función en la clase *Cuenta* que combine el comportamiento de `obtenerSaldo` y el de `establecerSaldo`. Este cambio reducirá la ventana de temporal pero no impedirá que existan entrelazados que permitan la existencia de condiciones de carrera. Para resolver esto se han utilizado clases *Java* para permitir el acceso en exclusión mutua al fichero *cuenta.txt*. Los mecanismos utilizados han sido `FileChannel` para obtener un canal por el cual manipular el fichero y bloquearlo, y `FileLock` para obtener el bloqueo proporcionado por el canal. Se utilizará el método `lock()` de la clase `FileChannel` que tiene comportamiento bloqueante para obtener el lock sobre el fichero *cuenta.java*. El código de esta *API* se puede encontrar en el Código 4.2.

#### 3.2. ¿Hay otras *API* vulnerables a la condición de carrera?

Otra de las *APIs* vulnerables sería *Carrito.java*, ya que podría darse el caso de que dos procesos intentasen escribir (método `writeBytes` de la clase `RandomAccessFile`) al mismo tiempo y en la misma parte del fichero el producto comprado. Para evitarlo, se podría utilizar una solución como la comentada en el apartado anterior (*FileLock*) para evitar la posibilidad de que dos procesos se solapen en la escritura del fichero.

#### 3.3. Implementar una versión de *CompraSegura.java*

Se ha implementado una versión segura de *Compra.java* que reduce la ventana temporal de la vulnerabilidad de condición de carrera y además utiliza la función `sacarDinero` comentada anteriormente para evitar fallos debido a una escritura después de lectura en el fichero *cuenta.txt*. Si se intenta replicar la ejecución de la Figura 1 se obtiene que una de las dos instancias informa de que el saldo no es suficiente. Si se observan los ficheros *cuenta.txt* y *carrito.txt* se comprobará que no hay saldo en la cuenta (*saldo* = 0) y que en el carrito solo se ha añadido el elemento *coche* una vez. Este comportamiento se puede observar en la Figura 3. El código de esta clase se puede encontrar en el Código 4.3.



```
seed@seed-desktop:~/practica5/api$ java Compra
Tu saldo actual es:40000
Lista de productos:
iPhone11      700 euros
coche         40000 euros
libro         80 euros
cachirulo     3 euros
golosinas     1 euros

¿Qué quieres comprar?
coche
Tu saldo es: 0
seed@seed-desktop:~/practica5/api$ cat backEndOrig/cuenta.txt
0
seed@seed-desktop:~/practica5/api$
```

```
seed@seed-desktop:~/practica5/api$ java Compra
Tu saldo actual es:40000
Lista de productos:
iPhone11      700 euros
coche         40000 euros
libro         80 euros
cachirulo     3 euros
golosinas     1 euros

¿Qué quieres comprar?
coche
Saldo insuficiente.
seed@seed-desktop:~/practica5/api$ cat backEndOrig/carrito.txt
coche
seed@seed-desktop:~/practica5/api$
```

Figura 3: Captura de pantalla del comportamiento de *CompraSegura.java*

## 4. Anexo 1: Códigos

### 4.1. Código vulnerable *Compra.java*

```
import java.io.IOException;
import java.util.Arrays;
import java.util.List;
import java.util.Scanner;

public class Compra {

    public static void main(String[] args) {
        Carrito carrito = null;
        Cuenta cuenta = null;
        try {
            carrito = new Carrito();
            cuenta = new Cuenta();
        } catch (Exception e) {
            e.printStackTrace();
        }
        // Escribir el saldo en pantalla
        int saldoActual = 0;
        try {
            saldoActual = cuenta.obtenerSaldo();
            System.out.println("Tu saldo actual es:" + saldoActual);
        } catch (IOException e) {
            e.printStackTrace();
        }
        // Escribir la lista de productos con sus precios
        System.out.println("Lista de productos:");
        String products = Tienda.muestraProductos();
        System.out.println(products);
        // Pide el producto que se quiere comprar
        Scanner scanner = new Scanner(System.in);
        System.out.println(" ¿Qu ¿quieres comprar?");
        String comprar = scanner.nextLine();
        // Comprueba que el saldo es suficiente para comprar el producto
        int saldoCompra = Tienda.obtenerPrecioProducto(comprar);
        if (saldoActual >= saldoCompra) {
            // Saldo suficiente
            try {
                carrito.meterProducto(comprar);
                int saldoFinal = saldoActual - saldoCompra;
                cuenta.establecerSaldo(saldoFinal);
                System.out.println("Tu saldo es:␣" + saldoFinal);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        scanner.close();
        if (carrito != null) {
            try {
                carrito.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

    }
    if (cuenta != null) {
        try {
            cuenta.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
}
}

```

## 4.2. Código de la implementación de sacarDinero()

```

/**
 * Retira de la cuenta la cantidad de dinero indicada
 */
public void sacarDinero(int cantidad) throws Exception {
    FileLock fileLock = null;
    // fc.lock es bloqueante
    fileLock = fc.lock(0L, Long.MAX_VALUE, false);
    int dineroActual = obtenerSaldo();
    if (dineroActual >= cantidad) {
        establecerSaldo(dineroActual - cantidad);
    } else {
        throw new RuntimeException("Saldo_insuficiente.");
    }

    if (fileLock != null) {
        fileLock.release();
    }
}
}

```

### 4.3. Código seguro compraSegura

```
import java.io.IOException;
import java.util.Arrays;
import java.util.List;
import java.util.Scanner;

public class Compra {

    public static void main(String[] args) {
        Carrito carrito = null;
        Cuenta cuenta = null;
        try {
            carrito = new Carrito();
            cuenta = new Cuenta();
        } catch (Exception e) {
            e.printStackTrace();
        }
        // Escribir el saldo en pantalla
        int saldoActual = obtenerSaldoActual(cuenta);
        System.out.println("Tu saldo actual es:" + saldoActual);
        // Escribir la lista de productos con sus precios
        System.out.println("Lista de productos:");
        String products = Tienda.muestraProductos();
        System.out.println(products);
        // Pide el producto que se quiere comprar
        Scanner scanner = new Scanner(System.in);
        System.out.println(" ¿Qu quieres comprar?");
        String comprar = scanner.nextLine();
        // Comprueba que el saldo es suficiente para comprar el producto
        int saldoCompra = Tienda.obtenerPrecioProducto(comprar);
        try {
            cuenta.sacarDinero(saldoCompra);
            carrito.meterProducto(comprar);
            int saldoFinal = obtenerSaldoActual(cuenta);
            System.out.println("Tu saldo es: " + saldoFinal);

        } catch (Exception e) {
            System.out.println(e.getMessage());
        }

        scanner.close();
        if (carrito != null) {
            try {
                carrito.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        if (cuenta != null) {
            try {
                cuenta.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
    }  
  
    private static int obtenerSaldoActual(Cuenta cuenta) {  
        int ret = 0;  
        try {  
            ret = cuenta.obtenerSaldo();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
        return ret;  
    }  
}
```