Secured Software Engineering – AE2

1)

A) The source of tainted data is the ‘productName’ variable. This variable is taken directly from user input using the ‘scanner.nextLine()’ function, without any sanitisation. This ‘productName’ variable is then concatenated with ‘sqlStatement’ string variable, and then executed using ‘PreparedStatement’. The SQL query execution (‘searchStatement.executeQuery()’) is the sensitive operation where the tainted data is used, meaning that it is a sink. The output from the tainted command (information leakage) is then saved to the variable ‘searchResult’. This is used in the while loop. Then this tained variable is used in the print statement. The results from the tainted query are shown. This is a sink, and it may result in information leakage. The second print statement is another sink, the tainted SQL query is printed to the console.

   The code is vulnerable to SQL Injection because the user input (‘productName’) is directly embedded into the SQL query without any sanitisation or parameterisation. An attacker can input a malicious string (e.g., ' OR '1'='1) to change the query and access unauthorized data, or even perform a full database takeover. There exist tools like ‘sqlmap’ that exist for this purpose, and are commonly used by both penetration testers and malicious hackers.

   To reference the code:

- String productName = scanner.nextLine(); is the source of the tainted input.
- String sqlStatement = "SELECT * FROM products WHERE product_name = '" + productName + "'"; is the line where the tainted value propogates to the variable ‘sqlStatement’.
- try (PreparedStatement searchStatement dbConnect.prepareStatement(sqlStatement)) { is the line where the taint propogates to the variable ‘searchStatement’.
- ResultSet searchResult = searchStatement.executeQuery(); is where the tainted value reaches the sink, is executed and saved to searchResult.
- while (searchResult.next()) { loops based off the tainted value.
- System.out.println("Product found: " + searchResult.getString("product_name")); prints out information gathered by the tainted query. This is printed out on the console. This is a sink.
- System.out.println("SQL query executed: [" + sqlStatement +"]"); is also a sink, as the tainted SQL query is printed out.

B) The goal of taint analysis is to track the flow of data from a ‘source’ of user input all the way down to its usage in sensitive operations, the ‘sink’. When the taint from the source moves from the source or a tainted variable to another variable, this is referred to as ‘propogation’. This assists in identifying SQL injections as it allows us to track variables tainted with user input and check if they sanitised or otherwise dealt with before being used in sensitive operations, such as lines of code where this code could be evaluated. In the given example, the ‘productName’ variable is the source of user input. This tainted input propogates down to the lines of that execute it, due to it being concatenated to the SQL query string as well as the lines that print out the results of the tainted SQL query.

2)

A) In order to mitigate the SQLi vulnerability, the code should use parameterised queries instead of concatenating user input directly to the SQL query. Refactored Java code:

```java
import java.sql.*;
import java.util.Scanner;

public class ProductSearcher {
    Run | Debug
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the product name");
        String productName = scanner.nextLine();
        scanner.close();

        try (Connection dbConnect = DriverManager.getConnection("jdbc:productdatabase", "admin", "adminpsd")) {
            String sqlStatement = "SELECT * FROM products WHERE product_name = ?";
            try (PreparedStatement searchStatement = dbConnect.prepareStatement(sqlStatement)) {
                searchStatement.setString(1, productName);
                ResultSet searchResult = searchStatement.executeQuery();
                while (searchResult.next()) {
                    System.out.println("Product found: " + searchResult.getString("product_name"));
                }
                System.out.println("SQL query executed: [" + sqlStatement + "]");
            }
        } catch (SQLException sqlErr) {
            sqlErr.printStackTrace();
        }
    }
}
```

This still has a hardcoded database password and a reliance on default encoing with the Scanner object (as found by SpotBugs), however this is outside the scope of the task.

B) My refactored code uses a paramterised query instead of concatenating tainted user input directly into the SQL string. ‘PreparedStatement’ is used to make sure that the user input is treated as a parameter and not as part of the SQL query.

This means that the tainted ‘productName’ is safely handled by PreparedStatement which treats the tainted input as a literal string instead of SQL code, and so it prevents the tainted data from propogating down to the sinks. So the application is no longer vulnerable to SQL injection. Other ways of dealing with SQL injection include validating the input (usually through regex) and sanitising it (usually with functions buillt into the language), although parameterised queries are usually the preferred method.