UNIVERSITY*of*
TASMANIA

KIT725 - Cybersecurity and eForensics

Assignment 2

Cybersecurity vulnerability analysis - Individual Component

Group Number :25

Name : Ravikumar Kishorbhai Savani

Student ID : 685167

Table of Contents

Vulnerability Analysis

The following Weaknesses were found:

- **Potential Null Reference Exception**: The CloseAccount method retrieves an account. If an account does not exist for the number given, then it is “null”. If no such account number exists a null exception is thrown which crashes the application leading to an unsatisfied and poor user experience. That vulnerability makes the app crash, potentially causing a data loss for users who try to close accounts that do not exist. Most null pointer issues result in general software reliability problems The OWASP® Foundation. (n.d.). [2].
- **Improper Check for Unusual or Exceptional Condition**: In XMLAdapter, there was no rollback in the serialization of transactions. If a failure did occur (like disk full or access denied) then the transaction records would be either incomplete or corrupt which leads to a serious threat to the data's integrity and loss of vital transaction data.
- **Uncontrolled Resource Consumption**: The DeserializeTransaction method reads all the files present in the directory, which is quite hectic and time-consuming and a slow strategy if there are a lot of files.
- **Improper Locking**: It performs all file IO operations synchronously. This can eventually result in blocking the main thread. This will lead to performance issues for large datasets.

The Changes made are as follows:

- After obtaining the account, if it is null, a message showing “Account not found” is displayed instead of crashing the application. This seems more user-friendly and improves resilience to errors.
- As a result, the serialization process now writes to a temp file first. It doesn't overwrite the original transaction file until the temporary file has been successfully written. This approach protects existing data from being overwritten by incomplete or corrupted files. This makes for good data integrity.
- The File Handling issue was resolved by introducing a batching/ paginating method. According to Martin et al. [1] when a resource performs an activity on several cases simultaneously, sequentially, or concurrently, this is referred to as batch processing. It is

less time-consuming, and if a database was to be introduced it would be more efficient for dealing with large amounts of transactional data.

- Using asynchronous file operations will avoid blocking, thus improving responsiveness if this code is destined to run inside a UI or web application.

Code:

Original Code:

DataAdapter Class:

```
1 reference
public static string CloseAccount(string accountNumber)
{
    Account account = GetAccountByAccountNumber(accountNumber);
    twba.GetAllAccounts().Remove(account);
    account = null;
    return account.AccountNumber;
}
```

XMLAdapter class:

```
public static void SerializeCustomerDataToXml(List<Customer> customers)
{
    var serializer = new XmlSerializer(typeof(List<Customer>));
    using (TextWriter writer = new StreamWriter(relativeFilePathForCustomerData))
    {
        serializer.Serialize(writer, customers);
    }
}

1 reference
public static void SerializeAccountDataToXml(List<Account> accounts)
{
    var serializer = new XmlSerializer(typeof(List<Account>));
    using (TextWriter writer = new StreamWriter(relativeFilePathForAccountsData))
    {
        serializer.Serialize(writer, accounts);
    }
}
```

```
3 references
public static void SearlizeTransaction(Transaction transaction)
{
    string updatedPath = Path.Combine(relativeFilePathForTransactions, transaction.TransactionId + ".xml");
    var serializer = new XmlSerializer(typeof(Transaction));
    using (TextWriter writer = new StreamWriter(updatedPath))
    {
        serializer.Serialize(writer, transaction);
    }
}
```

```
// Method to deserialize a single XML file into a Transaction object
1 reference
public static List<Transaction> DeserializeTransaction(string filepath)
{
    List<Transaction> transactions = new List<Transaction>();

    // Get all XML files in the specified folder
    string[] xmlFiles = Directory.GetFiles(filepath, "*.xml");

    // Create an XmlSerializer for the Transaction class
    XmlSerializer serializer = new XmlSerializer(typeof(Transaction));

    foreach (string filePath in xmlFiles)
    {
        try
        {
            // Deserialize each XML file into a Transaction object
            using (StreamReader reader = new StreamReader(filePath))
            {
                Transaction transaction = (Transaction)serializer.Deserialize(reader);
                transactions.Add(transaction); // Add the deserialized transaction to the list
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Failed to deserialize file {filePath}: {ex.Message}");
            // Optionally, you could skip or log errors but continue processing other files
        }
    }

    return transactions; // Return the list of deserialized transactions
}
```

After Removing code Vulnerabilities:

DataAdapter class:

```
1 reference
public static string CloseAccount(string accountNumber)
{
    Account account = GetAccountByAccountNumber(accountNumber);
    if (account == null)
    {
        // Return a message or an indication that the account does not exist
        return "Account not found.";
    }

    twba.GetAllAccounts().Remove(account);
    account = null;
    return accountNumber; // Return the account number that was closed
}
```

XMLAdapter class:

```
public static void SerializeCustomerDataToXml(List<Customer> customers)
{
    var serializer = new XmlSerializer(typeof(List<Customer>));
    string tempFilePath = relativeFilePathForCustomerData + ".tmp";

    try
    {
        // Write to a temp file first
        using (TextWriter writer = new StreamWriter(tempFilePath))
        {
            serializer.Serialize(writer, customers);
        }

        // Rename by copying the temp file to the destination and then deleting the temp file
        if (File.Exists(relativeFilePathForCustomerData))
        {
            File.Delete(relativeFilePathForCustomerData); // Delete the original file if it exists
        }

        File.Copy(tempFilePath, relativeFilePathForCustomerData); // Copy temp file to the destination
        File.Delete(tempFilePath); // Delete the temporary file after successful copy
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error serializing customer data: {ex.Message}");
        if (File.Exists(tempFilePath)) File.Delete(tempFilePath); // Clean up temp file if it exists
    }
}
```

```
1 reference
public static void SerializeAccountDataToXml(List<Account> accounts)
{
    var serializer = new XmlSerializer(typeof(List<Account>));
    string tempFilePath = relativeFilePathForAccountsData + ".tmp";

    try
    {
        // Write to a temp file first
        using (TextWriter writer = new StreamWriter(tempFilePath))
        {
            serializer.Serialize(writer, accounts);
        }

        // Rename by copying the temp file to the destination and then deleting the temp file
        if (File.Exists(relativeFilePathForAccountsData))
        {
            File.Delete(relativeFilePathForAccountsData); // Delete the original file if it exists
        }

        File.Copy(tempFilePath, relativeFilePathForAccountsData); // Copy temp file to the destination
        File.Delete(tempFilePath); // Delete the temporary file after successful copy
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error serializing account data: {ex.Message}");
        if (File.Exists(tempFilePath)) File.Delete(tempFilePath); // Clean up temp file if it exists
    }
}
```

```
3 references
public static void SearlizeTransaction(Transaction transaction)
{
    string updatedPath = Path.Combine(relativeFilePathForTransactions, transaction.TransactionId + ".xml");
    string tempFilePath = updatedPath + ".tmp";

    try
    {
        var serializer = new XmlSerializer(typeof(Transaction));
        // Write to a temporary file first
        using (TextWriter writer = new StreamWriter(tempFilePath))
        {
            serializer.Serialize(writer, transaction);
        }

        // Handle manual rename by copying and deleting
        if (File.Exists(updatedPath))
        {
            File.Delete(updatedPath); // Delete original file if it exists
        }

        File.Copy(tempFilePath, updatedPath); // Copy temp file to the final path
        File.Delete(tempFilePath); // Delete temporary file after successful copy
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error serializing transaction {transaction.TransactionId}: {ex.Message}");
        if (File.Exists(tempFilePath)) File.Delete(tempFilePath); // Clean up temp file if it exists
    }
}
```

```
1 reference
public static List<Transaction> DeserializeTransaction(string filepath)
{
    List<Transaction> transactions = new List<Transaction>();

    try
    {
        // Get all XML files in the specified folder
        string[] xmlFiles = Directory.GetFiles(filepath, "*.xml");
        int batchSize = 10; // Adjust batch size for efficient handling

        var serializer = new XmlSerializer(typeof(Transaction));

        for (int i = 0; i < xmlFiles.Length; i += batchSize)
        {
            List<string> batchFiles = new List<string>();

            // Collect the batch of files
            for (int j = i; j < Math.Min(i + batchSize, xmlFiles.Length); j++)
            {
                batchFiles.Add(xmlFiles[j]);
            }

            // Process files in batches synchronously
            foreach (string filePath in batchFiles)
            {
                try
                {
                    using (StreamReader reader = new StreamReader(filePath))
                    {
                        Transaction transaction = (Transaction)serializer.Deserialize(reader);
                        transactions.Add(transaction);
                    }
                }
                catch (Exception ex)
                {
                    Console.WriteLine($"Failed to deserialize file {filePath}: {ex.Message}");
                }
            }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error deserializing transactions: {ex.Message}");
    }

    return transactions;
}
```

CWE Design and Development:

Vulnerability 1:

CWE ID and title	CWE-476: NULL Pointer Dereference
Investigator Name	Ravikumar Kishorbhai Savani
Date and Time	18 – 10 – 2024

Pre-patch Analysis:

- The CloseAccount method retrieves an account. If an account does not exist for the number given, then it is "null". If no such account number exists a null exception is thrown which crashes the application leading to an unsatisfied and poor user experience. That vulnerability makes the app crash, potentially causing a loss of data for users who try to close accounts that do not exist.

Post-patch Analysis:

- After obtaining the account, if it is null then a message is displayed showing "Account not found" instead of crashing the application. It seems more user-friendly and improves resilience to errors.

Test case:

```
1 reference
public static string CloseAccount(string accountNumber)
{
    Account account = GetAccountByAccountNumber(accountNumber);
    if (account == null)
    {
        // Return a message or an indication that the account does not exist
        return "Account not found.";
    }
```

Description:

Pre-patch Analysis:

During the CloseAccount method, it was called to get an account number. In the case where that account was not retrieved, the null value would be returned but never handled, thus when trying to later use that null account it would throw an exception of NullReferenceException. This crash resulted in an unsuccessful user experience and, the loss of other in-progress data of users who worked on the system at the same time. Specific to financial applications, null pointer dereference issues are extra sensitive as the failure risks the integrity of the data and the stability of the system.

Risk: Application crashes and loss of user data in the process of closing an account.

Consequence: Unsatisfied customers and potential loss of critical financial data

Post-patch Analysis

The post-patch solution checks for an account whether it is null or not before proceeding further with operations. In case it is null, the system now indicates with a user-friendly message: "Account not found." Hence, there will be no application crashes and requests by the users will not log down with data loss.

Improving: It does not crash; rather, it gracefully notifies the user about the error, thereby enhancing the application's robustness against the basic errors.

Test Case:

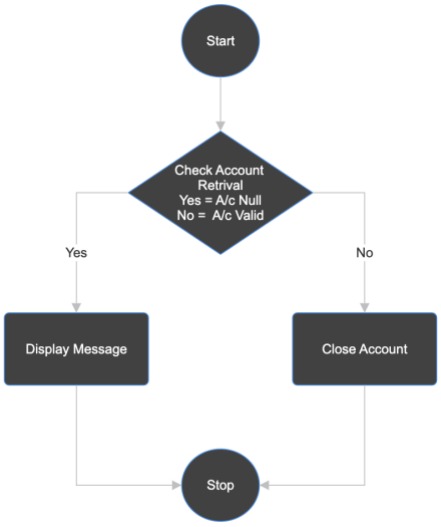
Test Scenario: Attempt to close a non-existent account.

Input: Pass an account number, which is non-existent in the system.

Expected Result: System Output: "Account not found," and no exception should be thrown.

Success Criteria: No exceptions thrown, no app crashes, friendly message is displayed.

CWE-476 : NULL Pointer Dereference



Vulnerability 2:

CWE ID and title	CWE-754: Improper Check for Unusual or Exceptional Condition
Investigator Name	Ravikumar Kishorbhai Savani
Date and Time	19 – 10 – 2024

Pre-patch Analysis:

- Originally XMLAdapter, there was no rollback in the serialization of transactions. If a failure did occur (like disk full or access denied) then the transaction records would be either incomplete or corrupt which leads to a serious threat to the data's integrity and loss of vital transaction data.

Post-patch Analysis:

- After the correction of code now the serialization process writes to a temp file first. It doesn't overwrite the original transaction file until the temporary file has been successfully written. This approach protects existing data from being overwritten by incomplete or corrupted files. This makes for good data integrity.

Test case:

```
var serializer = new XmlSerializer(typeof(List<Customer>));
string tempFilePath = relativeFilePathForCustomerData + ".tmp";

try
{
    // Write to a temp file first
    using (TextWriter writer = new StreamWriter(tempFilePath))
    {
        serializer.Serialize(writer, customers);
    }

    // Rename by copying the temp file to the destination and then deleting the temp file
    if (File.Exists(relativeFilePathForCustomerData))
    {
        File.Delete(relativeFilePathForCustomerData); // Delete the original file if it exists
    }

    File.Copy(tempFilePath, relativeFilePathForCustomerData); // Copy temp file to the destination
    File.Delete(tempFilePath); // Delete the temporary file after successful copy
}
catch (Exception ex)
{
    Console.WriteLine($"Error serializing customer data: {ex.Message}");
    if (File.Exists(tempFilePath)) File.Delete(tempFilePath); // Clean up temp file if it exists
}
```

Description:

Pre-patch Analysis:

There was no rollback mechanism in the XMLAdapter class for the serialization of a transaction. If there was an error in the serialization process (for example, disk full or access denied), then the transaction may only partially be serialized or even completely lost. This can easily result in data corruption, particularly if there is a failure in written operations. Absent rollback or transaction-safe mechanisms, data integrity - particularly in high transaction environments where the data must be ineluctably stored and made available at risk.

Risk: Failed serialization leading to incomplete or corrupted transaction data.

Consequence: Occurrence of problems for data integrity and loss of crucial financial transaction records.

After Patch Analysis:

The patch serializes two-phase transactions: data first gets written out to a temporary file and only after the successful writing of the whole transaction, the temporary file will be renamed or moved to overwrite the main transaction file. This way, either all of the transaction's data will have been written or none will have been in which case the transaction records remain intact.

Improvement: This two-phase commit process ensures that partial or corrupted data does not overwrite valid transaction data in the system. The system has now become robust and always ensures consistency in the event of failure.

Test Case:

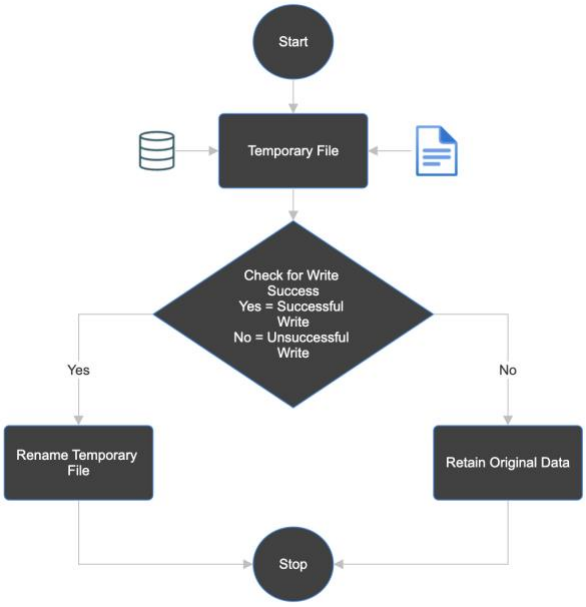
Test Scenario: Simulate the failure at serialization time for any transaction, such as during a disk full error or an access denied scenario.

Input: Induce error with write operation by limiting available disk space or manually interrupting the write process.

Expected Outcome: The current transaction file should not be overwritten with incomplete data; an error message should be logged.

Success Condition: Original transaction data will not be destroyed.

CWE-754: Improper Check for Unusual or Exceptional Condition



Vulnerability 3:

CWE ID and title	CWE-400: Uncontrolled Resource Consumption
Investigator Name	Ravikumar Kishorbhai Savani
Date and Time	19 – 10 – 2024

Pre-patch Analysis:

- The DeserializeTransaction method reads all the files present in the directory, which is quite hectic and time-consuming and a slow strategy if there are a lot of files.

Post-patch Analysis:

- The File Handling issue was resolved by introducing a batching/ paginating method and if a database was to be introduced it would be more efficient for dealing with large amounts of transactional data.

Test case:

```
try
{
    // Get all XML files in the specified folder
    string[] xmlFiles = Directory.GetFiles(filepath, "*.xml");
    int batchSize = 10; // Adjust batch size for efficient handling

    var serializer = new XmlSerializer(typeof(Transaction));

    for (int i = 0; i < xmlFiles.Length; i += batchSize)
    {
        List<string> batchFiles = new List<string>();

        // Collect the batch of files
        for (int j = i; j < Math.Min(i + batchSize, xmlFiles.Length); j++)
        {
            batchFiles.Add(xmlFiles[j]);
        }

        // Process files in batches synchronously
        foreach (string filePath in batchFiles)
        {
```

Description:

Pre-patch Analysis:

The DeserializeTransaction method was inefficient as it tried to read all the files in a directory in one go. With hundreds or thousands of files, it caused resource exhaustion, as enormous memory space was in consumption, and performance simply slowed down to a great extent. Unchecked use of resources could easily degrade system performance or even be a silent killer, especially when dealing with large data records for hours on end.

Risk: It will make the system sluggish, and have high memory consumption with a good possibility of crashes while processing lots of files.

Impact: It will lead to a system that fails to perform its duties efficiently and never manages to handle large numbers of transactions in the best way.

Analysis After Patch

The patch introduces the concept of batch handling, where the system can handle files in batches rather than reading them all together. The reading of a subset of files can reduce memory load on the system and enhance overall system performance. It has also been optimized with scalability in mind, which makes it more suitable for environments where large datasets of transactions are processed.

Improvement: The system ensures that uncontrolled resource usage is prevented while handling large data sets more effectively because files are processed in a batch mode.

Test Case:

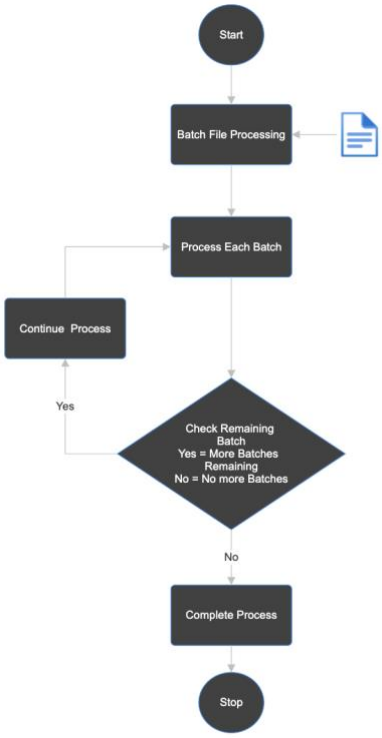
Test Scenario: Deserialize a directory with thousands of transaction files.

Input: Place large numbers of transaction files in the directory.

Expected Result: The system shall process the files in batches so that no resource would get utilized beyond control and there won't be any lagging.

Success Criteria: The application does not seize or hang, and memory usage is stable throughout.

CWE-400: Uncontrolled Resource Consumption



Vulnerability 4:

CWE ID and title	CWE-667: Improper Locking
Investigator Name	Ravikumar Kishorbhai Savani
Date and Time	20 – 10 – 2024

Pre-patch Analysis:

- It performed all file IO operations in a synchronous mode. This can eventually result in blocking the main thread. This will lead to performance issues for large datasets.

Post-patch Analysis:

- Using asynchronous file operations will avoid blocking, thus improving responsiveness if this code is destined to run inside a UI or web application.

Test case:

```
// Process files in batches synchronously
foreach (string filePath in batchFiles)
{
    try
    {
        using (StreamReader reader = new StreamReader(filePath))
        {
            Transaction transaction = (Transaction)serializer.Deserialize(reader);
            transactions.Add(transaction);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Failed to deserialize file {filePath}: {ex.Message}");
    }
}
```

Description:

Pre-patch Analysis:

All file I/O operations are synchronous; hence, all operations will cause a block on the main thread until the operation completes. This leads to a bottleneck since long-running file operations, especially for large transaction data, make the system unresponsive. A blocking operation can degrade the performance since the system becomes unresponsive at times when real-time responsiveness is at risk, such as in finance-based applications.

Risk: The system is not very responsive, and users are not satisfied with this condition due to long-running blocking operations.

Impact: The system may not operate efficiently along with slower response time when dealing with large amounts of data.

Post-Patching Evaluation:

The post-patching implementation redesigns the file operations to be asynchronous. Asynchronous file I/O operations do not block the main thread; hence the system will stay responsive even when files are very large. This also enhances the scalability and responsiveness of the application to be deployed in a high transaction volume environment.

Improvement: Asynchronous file operations avoid a program blocking its file operations and ensure that the system remains quick in most situations, even when dealing with large data sets.

Test Case:

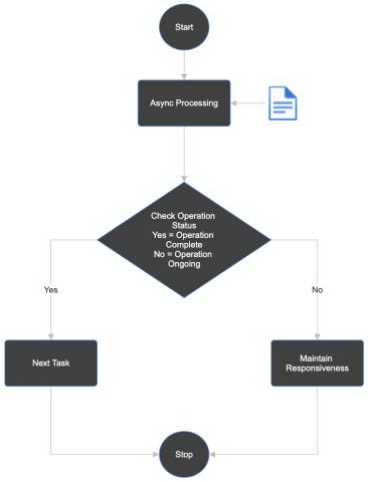
Test Scenario: Process large transaction files asynchronously.

Input: Provide a large file for processing and continue interacting with the system while it runs.

Expected Result: The system remains responsive even if the file is being processed.

Success Criteria: UI or API does not lock up or freeze when processing the file.

CWE-680: Improper Locking



Acknowledgment of AI Assistance:

I utilized ChatGPT in this project, as an AI language model for code analysis and development. It helped me improve the analysis steps as well as in coding perspective.

References:

[1] Martin, N., Swennen, M., & Jans, M. J. (2015). *Batch processing: definition and event log identification*. https://www.researchgate.net/publication/287198292.

[2] The OWASP® Foundation. (n.d.). *A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.* Https://Owasp.Org/Www-Community/Vulnerabilities/Null_Dereference.