EDGES
OUR EDGE IS YOUR SUCCESS

Report Bugs For Application "EDGES Reservation software"

Created By	Bassam Ashraf
Reviewed By	

Test Function (Add Account):

Test ID	Test Case	Technique	Status
3	Add Account with (vaild name)	Boundary Value Analysis	Failed

1. Expected Output vs Actual Output

A. Expected Output:

- The database should be updated with the user inputs when the name length is within the valid range (3 to 32 characters).
- For a name of exactly 32 characters (**"ThisIsExactly32CharactersLong123"**), the function **Add_Account** should return **TRUE**, and the user should be successfully added to the database.

B. Actual Output:

- For names with lengths of 3 and 10 characters, the function behaves as expected.
- However, for a name of exactly 32 characters, the function fails to add the user to the database. The string is not stored, and the validation fails.

2. How to Reproduce the Issue

To reproduce the issue, follow these steps:

1. Initialize the database and ensure at least one user is assigned to it.
2. Call the **Add_Account** function with the following inputs:
   - Name: **"ThisIsExactly32CharactersLong123"** (32 characters).
   - Age: **26**.
   - DOB_day: **1**.

- DOB_Month: **2**.
- DOB_Year: **1998**.
- Gender: **Male**.
- Educational_Status: **Student**.
- UserName: **"AdminUser1"**.
- Password: **"Edges123"**.
- Password Recheck: **"Edges123"**.

3. Observe the result of the function and check whether the user is added to the database.

The issue occurs because the name of exactly 32 characters is not added to the database, and the function fails to handle this boundary case correctly.

3. Root Cause

The root cause of the bug is related to memory allocation and buffer overflow:

- A 32-character string requires **33 bytes** of storage **(32 characters + 1 byte for the null terminator \0).**
- The current implementation uses a fixed-size buffer of 32 bytes to store the name.

```
char name [32]; // Allocate 32 bytes for the name
```

- This causes undefined behavior, leading to memory corruption and failure in the validation process.

4. Suggested Fix

To resolve this issue, the following change recommended:

1. **Increase Buffer Size:**

Update the size of the buffer used to store the name from **32** to **33** to accommodate the null terminator.

Example:

```
char name [33]; // Allocate 33 bytes for the name (32
characters + null terminator)
```

Test ID	Test Case	Technique	Status
5	Add Account with (valid Age)	Boundary Value Analysis	Failed

1. Expected Output vs Actual Output

- **Expected Output:**
  - The database should be updated with the user inputs when the age is within the valid range (**0** to **100**).
  - For ages of **0**, **26**, and **100**, the function **Add_Account** should return **TRUE**, and the user should be successfully added to the database.

- **Actual Output:**
  - For age **26**: The function behaves as expected, returning **TRUE** and adding the user to the database.
  - For age **0**: The function fails to add the user however its (meaningless), even though **0** is within the valid range.
  - For age **100**: The function fails to add the user however its (impossible nowadays), even though **100** is within the valid range.

2. How to Reproduce the Issue

To reproduce the issue, follow these steps:

1. Initialize the database and ensure at least one user is assigned to it.
2. Call the **Add_Account** function with the following inputs:
   - Age = 0: A valid but edge-case input.
   - Age = 26: A valid and typical input.
   - Age = 100: Another valid but edge-case input.
3. Observe the result of the function and check whether the user is added to the database.

The issue occurs because the function fails to handle boundary values (**0** and **100**) correctly, even though they are within the specified valid range.

3. Root Cause

The root cause of the bug lies in the condition used to validate the age range:

```
else if (Form->PersonalInfo_Form.Age < 0 ||
Form->PersonalInfo_Form.Age > 100)
{
    RET = FALSE;
}
```

- This condition excludes the boundary values **0** and **100** from the valid range due to incorrect logical handling.Specifically:
  - For **Age = 0**: The condition incorrectly treats **0** as invalid, even though it is explicitly stated as part of the valid range.
  - For **Age = 100**: Similarly, the condition incorrectly treats **100** as invalid, despite being within the valid range.

This results in the function rejecting valid inputs (**0** and **100**) and failing to add the corresponding users to the database.

4. Suggested Fix

To resolve this issue, the following change recommended:

1. **Correct the Condition:**

   Update the condition to properly include the boundary values (**0** and **100**) in the valid range.

```
else if (Form->PersonalInfo_Form.Age <= 0 ||
Form->PersonalInfo_Form.Age >= 100)
{
    RET = FALSE;
}
```

Test ID	Test Case	Technique	Status
11	Add Account with (vaild username)	Boundary Value Analysis	Failed

1. Expected Output vs Actual Output

- **Expected Output:**
  - The database should be updated with the user inputs when the username length is within the valid range (**8** to **32** characters).
  - For usernames of **8**, **10**, and **32** characters, the function **Add_Account** should return **TRUE**, and the user should be successfully added to the database.
- **Actual Output:**
  - For usernames of **8** and **10** characters: The function behaves as expected, returning **TRUE** and adding the user to the database.
  - For a username of exactly **32** characters: The function fails to add the user to the database. The string is not stored, and the validation fails.

2. How to Reproduce the Issue

To reproduce the issue, follow these steps:

1. Initialize the database and ensure at least one user is assigned to it.
2. Call the **Add_Account** function with the following inputs:
   - Username: **"ThisIsExactly32CharactersLong123"** (32 characters).
   - Other valid inputs (e.g., Name: **"Ahmed"**, Age: **26**, DOB: **1/2/1998**, Gender: **Male**, Educational_Status: **Student**, Password: **"Edges123"**, Password Recheck: **"Edges123"**).
3. Observe the result of the function and check whether the user is added to the database.

The issue occurs because the username of exactly **32** characters is not added to the database, despite being within the valid range.

3. Root Cause

The root cause of the bug is related to memory allocation and buffer overflow:

- A **32**-character string requires 33 bytes of storage (32 characters + 1 byte for the null terminator **\0**).
- The current implementation uses a fixed-size buffer of 32 bytes to store the username.

```
char Username [32]; // Allocate 32 bytes for the Username
```

- This causes undefined behavior, leading to memory corruption and failure in the validation process.

4. Suggested Fix

To resolve this issue, the following changes are recommended:

1. **Increase Buffer Size:**

Update the size of the buffer used to store the username from **32** to **33** to accommodate the null terminator.Example:

```
char Username [33]; // Allocate 33 bytes for the Username (32 characters + null terminator)
```

Test ID	Test Case	Technique	Status
13	Add Account with (vaild password length)	Boundary Value Analysis	Failed

1. Expected Output vs Actual Output

- **Expected Output:**
  - The database should be updated with the user inputs when the password length is within the valid range (**8** to **32** characters).
  - For passwords of **8**, **10**, and **32** characters, the function **Add_Account** should return **TRUE**, and the user should be successfully added to the database.
- **Actual Output:**
  - For passwords of **8** and **10** characters: The function behaves as expected, returning **TRUE** and adding the user to the database.
  - For a password of exactly **32** characters: The function fails to add the user to the database. The string is not stored, and the validation fails.

2. How to Reproduce the Issue

To reproduce the issue, follow these steps:

1. Initialize the database and ensure at least one user is assigned to it.
2. Call the **Add_Account** function with the following inputs:
   - Password: **"ThisIsExactly32CharactersLong123"** (32 characters).
   - Password Recheck: **"ThisIsExactly32CharactersLong123"**.
   - Other valid inputs (e.g., Name: **"Ahmed"**, Age: **26**, DOB: **1/2/1998**, Gender: **Male**, Educational_Status: **Student**, UserName: **"AdminUser1"**).

3. Observe the result of the function and check whether the user is added to the database.

The issue occurs because the password of exactly **32** characters is not added to the database, despite being within the valid range.

3. Root Cause

The root cause of the bug is related to memory allocation and buffer overflow:

- A **32**-character string requires 33 bytes of storage (32 characters + 1 byte for the null terminator **\0**).
- The current implementation uses a fixed-size buffer of 32 bytes to store the password.

```
char Password [32]; // Allocate 32 bytes for the
Password
```

- This causes undefined behavior, leading to memory corruption and failure in the validation process.

4. Suggested Fix

To resolve this issue, the following changes are recommended:

1. **Increase Buffer Size:**

Update the size of the buffer used to store the password from **32** to **33** to accommodate the null terminator, example:

```
char Password [33]; // Allocate 33 bytes for the
Password (32 characters + null terminator)
```

Test ID	Test Case	Technique	Status
15	Add Account with (vaild password recheck length)	Boundary Value Analysis	Failed

1. Expected Output vs Actual Output

- **Expected Output:**
  - ➢ The database should be updated with the user inputs when the password recheck length is within the valid range (**8** to **32** characters).
  - ➢ For password recheck lengths of **8**, **10**, and **32** characters, the function **Add_Account** should return **TRUE**, and the user should be successfully added to the database.
- **Actual Output:**
  - ➢ For password recheck lengths of **8** and **10** characters: The function behaves as expected, returning **TRUE** and adding the user to the database.
  - ➢ For a password recheck length of exactly **32** characters: The function fails to add the user to the database. The string is not stored, and the validation fails.

---

2. How to Reproduce the Issue

To reproduce the issue, follow these steps:

1. Initialize the database and ensure at least one user is assigned to it.
2. Call the **Add_Account** function with the following inputs:
   - ➢ Password: **"ThisIsExactly32CharactersLong123"** (32 characters).
   - ➢ Password Recheck: **"ThisIsExactly32CharactersLong123"**.

- Other valid inputs (e.g., Name: **"Ahmed"**, Age: **26**, DOB: **1/2/1998**, Gender: **Male**, Educational_Status: **Student**, UserName: **"AdminUser1"**).

3. Observe the result of the function and check whether the user is added to the database.

The issue occurs because the password recheck of exactly **32** characters is not added to the database, despite being within the valid range.

3. Root Cause

The root cause of the bug is related to memory allocation and buffer overflow:

- A **32**-character string requires 33 bytes of storage (32 characters + 1 byte for the null terminator **\0**).
- The current implementation uses a fixed-size buffer of 32 bytes to store the password recheck.

```
char Password_recheck [32]; // Allocate 32 bytes for
the Password_recheck
```

- This causes undefined behavior, leading to memory corruption and failure in the validation process.

4. Suggested Fix

To resolve this issue, the following changes are recommended:

1. **Increase Buffer Size:**

Update the size of the buffer used to store the password recheck from **32** to **33** to accommodate the null terminator, example:

```
char Password_recheck [33]; // Allocate 33 bytes for
the Password_recheck (32 characters + null terminator)
```

Test ID	Test Case	Technique	Status
18	Add Account (Adding Extra User Above Maximum Users - 5)	Equivalence Partitioning	Failed

1. Expected Output vs Actual Output

- **Expected Output:**
  - The database should reject adding a sixth user when it already contains the maximum allowed number of users (5).
  - The function **Add_Account** should return **FALSE** to indicate that the user was not added to the database.
- **Actual Output:**
  - The sixth user is not added to the database, as expected.
  - However, the function **Add_Account** incorrectly returns **TRUE**, indicating success even though the user was not added.

2. How to Reproduce the Issue

To reproduce the issue, follow these steps:

1. Initialize the database and add five valid users to it.
2. Call the **Add_Account** function with inputs for a sixth user:
   - Name: **"Ahmed"**.
   - Age: **26**.
   - DOB: **01/02/1999**.
   - Educational_Status: **Masters_Student**.
   - Gender: **Male**.
   - UserName: **"EdgesAcademy"**.
   - Password: **"Edges123"**.

- Password Recheck: **"Edges123"**.

3. Observe the result of the function and check whether the user is added to the database.

The issue occurs because the function returns **TRUE** even though the sixth user is not added to the database.

3. Root Cause

The root cause of the bug lies in the logic of the **Add_Account** function:

- The function calls **DBM_Add_User(Form)** to attempt adding the user to the database.
- However, instead of returning the result of **DBM_Add_User(Form)** directly, it unconditionally sets **RET = TRUE** after calling the function.
- This causes the function to incorrectly indicate success (**TRUE**) even when **DBM_Add_User(Form)** fails to add the user due to the database being full.

```
else
{
    DBM_Add_User(Form);
    RET = TRUE; // Always sets RET to TRUE,
regardless of whether the user was actually added
}
```

4. Suggested Fix

To resolve this issue, the following changes are recommended:

1. Return the Result of **DBM_Add_User** :

   Modify the logic to return the actual result of **DBM_Add_User(Form)** instead of unconditionally setting **RET = TRUE**.

```
else
{
    RET = DBM_Add_User(Form);
    TRUE; // Return the result of DBM_Add_User
}
```

Test Function (Delete Account):

Test ID	Test Case	Technique	Status
19	Delete Account (Valid and Invalid User ID)	Boundary Value Analysis	Failed

1. Expected Output vs Actual Output

- **Expected Output:**
  - For invalid user IDs (**-1** and **6**), the function should return **FALSE**, indicating that no user was deleted.
  - For valid user IDs (**0**, **1**, and **5**), the function should return **TRUE**, indicating that the corresponding user was successfully deleted.
- **Actual Output:**
  - For invalid user ID **-1**: The function correctly returns **FALSE**.
  - For invalid user ID **6**: The function incorrectly returns **TRUE** and deletes the user with ID **5**.
  - For valid user IDs (**0**, **1**, and **5**): The function behaves as expected, returning **TRUE** and deleting the corresponding users.

2. How to Reproduce the Issue

To reproduce the issue, follow these steps:

1. Initialize the database and add five valid users to it.
2. Call the **Delete_Account** function with the following inputs:
   - User ID: **-1** (invalid negative ID).
   - User ID: **6** (invalid out-of-range ID).
   - User ID: **0** (valid first user ID).
   - User ID: **5** (valid last user ID).

3. Observe the result of the function and check whether the users are deleted from the database.

The issue occurs because the function incorrectly handles the invalid user ID **6**, treating it as valid and deleting the user with ID **5**.

3. Root Cause

The root cause of the bug lies in the condition used to validate the user ID range:

```
if (user_id < 0 || user_id > MAX_USERS)
{
    RET = FALSE;
}
```

- This condition incorrectly allows user IDs equal to **MAX_USERS** (e.g., **6**) to pass validation.
- Since array indices are zero-based, valid user IDs should range from **0** to **MAX_USERS - 1**. The condition should explicitly exclude **MAX_USERS** by using **>=** instead of **>**:

4. Suggested Fix

To resolve this issue, the following changes are recommended:

1. **Correct the Validation Condition:**

Update the condition to properly validate the user ID range:

```
if (user_id < 0 || user_id >= MAX_USERS)
{
    RET = FALSE;
}
```

This ensures that only valid user IDs (**0** to **MAX_USERS - 1**) are accepted.

Test Function (Show Student Courses):

Test ID	Test Case	Technique	Status
26	Show Student Courses (Valid User with No Registered Courses)	Equivalence Partitioning	Failed

1. Expected Output vs Actual Output

- **Expected Output:**
  - The function **ShowStudentCourses** should print:
    "You Are Currently Enrolled in:" but function correctly identifies that the user has no registered courses but fails to provide a meaningful message to the user.
- **Actual Output:**
  - The function prints only:
    "You Are Currently Enrolled in: “and does not display any additional message indicating that the user has no registered courses.

2. How to Reproduce the Issue

To reproduce the issue, follow these steps:

1. Initialize the database and add a valid user with ID = **1**.
2. Ensure that the user is not enrolled in any courses.
3. Call the **ShowStudentCourses** function with the user's ID (**0**).
4. Observe the output of the function.

The issue occurs because the function lacks a clear message indicating that the user has not registered for any courses.

3. Root Cause

The root cause of the bug lies in the logic of the **ShowStudentCourses** function:

- The function correctly identifies that the user has no registered courses but fails to provide a meaningful message to the user.
- Instead of printing a default or generic message like **"You Are Currently Not Registered To Any Courses Yet"**, it simply omits any further information after **"You Are Currently Enrolled in:"**.

This creates confusion for the user, as they are left without clear feedback about their course registration status.

4. Suggested Fix

To resolve this issue, the following changes are recommended:

1. **Add a Default Message for No Courses:**

Modify the function to include a message when no courses are registered. For example:

```
void ShowStudentCourses(unsigned int User_ID)
{
    unsigned char RET = FALSE;

    if (User_ID < 0 || User_ID >= MAX_USERS)
    {
        printf("Invalid User Id \n");
        RET = FALSE;

    }
    else
    {
        RET = DBM_ShowCourse(User_ID);
        if (RET == FALSE)
        {
        printf("You Are Not Enrolled in Any Course\n");
        }
        else
        {
         printf("You Are Enrolled in Some Courses\n");
        }
    }
```

Test Function (Add Student To Courses):

Test ID	Test Case	Technique	Status
28	Add Student to Course (Valid User and Valid Course)	Boundary Value Analysis	Failed

1. Expected Output vs Actual Output

- **Expected Output:**
  - ➢ The function **AddStudentToCourse** should not add any courses to invalid user IDs (**-1**, **0**, or **6**).
  - ➢ For invalid user IDs, the function should return **Enroll_Failed** or a similar error code indicating that the operation was unsuccessful.
  - ➢ Additionally, the system should not crash when invalid user IDs are provided as input.
- **Actual Output:**
  - ➢ For user ID **-1** and **0**: The system crashes (Run-Time Error) due to an out-of-bounds array access in the **Enrollments** array.
  - ➢ For user ID **6**: The function incorrectly returns **Enrolled**, even though the user ID is invalid.

2. How to Reproduce the Issue

To reproduce the issue, follow these steps:

1. Initialize the database and assign valid users with IDs (**-1**, **0**, and **6**) to it.
2. Call the **AddStudentToCourse** function with the following inputs:
   - ➢ User ID: **-1** (invalid negative ID), Course ID: **1** (Standard_Diploma).
   - ➢ User ID: **0** (invalid zero ID), Course ID: **1** (Standard_Diploma).
   - ➢ User ID: **6** (invalid out-of-range ID), Course ID: **1** (Standard_Diploma).
3. Observe the behavior of the function:
   - ➢ For user IDs **-1** and **0**, the system crashes during execution.
   - ➢ For user ID **6**, the function incorrectly returns **Enrolled**.

3. Root Cause

The root cause of the bug lies in the logic of the **AddStudentToCourse** function:

1. **System Crash for Invalid IDs (-1 and 0):**
   - ➢ The function accesses the **Enrollments** array without validating the user ID.

- For invalid IDs (**-1** and **0**), the resulting array index becomes negative (**-2** and **-1**, respectively), causing an out-of-bounds memory access and leading to a runtime error.

2. **Incorrect Return Value for Out-of-Range ID (6):**
    - The function does not validate whether the user ID is within the valid range (**0** to **MAX_USERS - 1**).
    - As a result, the function proceeds to enroll the user in the course and returns **Enrolled**, even though the user ID is invalid.
    - This line assumes that Student_id is always valid, which leads to undefined behavior for invalid IDs.

```
if (Enrollments[Student_id][Course_id - 1] == TRUE)
{
    // Check if the student is already enrolled
}
```

4. Suggested Fix

To resolve this issue, the following changes are recommended:

1. **Validate User ID Before Accessing the Array:**
Add a check at the beginning of the function to ensure the user ID is within the valid range (**0** to **MAX_USERS - 1**)

```
if (user_id < 0 || user_id >= MAX_USERS)
{
    return Enroll_Failed; // Return failure for invalid
user IDs
}
```

Test ID	Test Case	Technique	Status
29	Add Student to Course (Valid User and Invalid Course)	Equivalence Partitioning	Failed

1. Expected Output vs Actual Output

- **Expected Output:**
  - The function **AddStudentToCourse** should not add any courses to invalid course IDs (**0** and **7**).
  - For invalid course IDs, the function should return an error code such as **InvalidCourseID** or a similar value indicating that the operation was unsuccessful.
  - Additionally, the system should not crash when invalid course IDs are provided as input.
- **Actual Output:**
  - For course ID **0**: The system crashes during execution due to an out-of-bounds array access in the **Enrollments** array.
  - For course ID **7**: The function incorrectly returns **CapacityCompleted**, even though the course ID is invalid.

2. How to Reproduce the Issue

To reproduce the issue, follow these steps:

1. Initialize the database and assign valid users with IDs (**1** and **2**) to it.
2. Call the **AddStudentToCourse** function with the following inputs:
   - User ID: **1**, Course ID: **0** (invalid course ID).
   - User ID: **2**, Course ID: **7** (invalid out-of-range course ID).
3. Observe the behavior of the function:
   - For course ID **0**, the system crashes during execution.
   - For course ID **7**, the function incorrectly returns **CapacityCompleted**.

3. Root Cause

The root cause of the bug lies in the logic of the **AddStudentToCourse** function:

1. **System Crash for Invalid Course ID (0):**
   - The function accesses the **Enrollments** array without validating the course ID.
   - For an invalid course ID (**0**), the resulting array index becomes negative (**-1**), causing an out-of-bounds memory access and leading to a runtime error.
2. **Incorrect Return Value for Out-of-Range Course ID (7):**
   - The function does not validate whether the course ID is within the valid range (**1** to **MAX_COURSES**).
   - As a result, the function proceeds to check the enrollment status and returns **CapacityCompleted**, even though the course ID is invalid.

```
if (Enrollments[Student_id][Course_id - 1] == TRUE)
{
    // Check if the student is already enrolled
}
```

4. Suggested Fix

To resolve this issue, the following changes are recommended:

1. **Validate Course ID Before Accessing the Array:**

   Add a check at the beginning of the function to ensure the course ID is within the valid range (**1** to **MAX_COURSES**)

```
if (Course_id < 1 || Course_id > MAX_COURSES)
{
    return InvalidCourseID; // Return failure for invalid
course IDs
}
```

Test Function (Detect User Type):

Test ID	Test Case	Technique	Status
35	Non-Numeric Input Handling	Equivalence Partitioning	Failed

1. Expected Output vs Actual Output

- **Expected Output:**
  - ➢ The function **Detect_User_Type** should reject non-numeric input (e.g., **"abc"**, **"$"**) and return **IncorrectLogin**.
- **Actual Output:**

  When non-numeric input is provided, the function does not handle it correctly:

  - ➢ The **scanf()** function fails to read the invalid input, leaving the variable **User_Type** uninitialized or containing garbage values.
  - ➢ In some cases, this causes the function to incorrectly interpret the input as **AdminMohamedTarek** (value **0**) instead of returning **IncorrectLogin**.
  - ➢ Additionally, the program may enter an infinite loop or exhibit undefined behavior due to the unhandled invalid input.

2. How to Reproduce the Issue

To reproduce the issue, follow these steps:

1. Simulate user input by redirecting input to a non-numeric value (e.g., **"abc"** or **"$"**).
2. Call the **Detect_User_Type** function.
3. Observe the behavior of the function and check the returned value.

The issue occurs because the function uses **scanf("%d", &User_Type)** without validating whether the input is numeric. If the input is non-numeric, **scanf()** fails to assign a value to **User_Type**, leading to undefined behavior.

3. Root Cause

The root cause of the bug lies in the use of **scanf()** without proper error handling:

- The **scanf()** function does not validate whether the input matches the expected format (**%d** for integers).
- When non-numeric input is provided, **scanf()** fails to assign a value to **User_Type** and returns **0** (indicating failure).
- The function does not check the return value of **scanf()**, which leaves **User_Type** uninitialized or containing garbage values.
- This can lead to incorrect results (e.g., interpreting invalid input as **AdminMohamedTarek**) or runtime issues (e.g., infinite loops).
- This line assumes that the input will always be numeric, which is not guaranteed.

```
scanf("%d", &User_Type);
```

4. Suggested Fix

To resolve this issue, the following changes are recommended:

1. **Check the Return Value of scanf():**

   Add a check to ensure that **scanf()** successfully reads an integer before proceeding:

```
int result;
result = scanf("%d", &User_Type);



// Check if scanf successfully read an integer
```

```
if (result != 1)
{
    // Clear the input buffer to discard invalid input
    while (getchar() != '\n'); // Discard characters
until newline

    // Return IncorrectLogin for invalid input
    return IncorrectLogin;
}
```