# Local Functions, Their Full Disassembly, and Strings from Object Files, Refactor and Securty Enhancemnts.

## File: Project1.o

Format: ELF64-x86-64

### Disassembly of Section .text:

Let us elaborate on the direct correspondence between the assembly code and the C++ lines included in the main, ChangeCustomerChoice, CheckUserPermissionAccess, and DisplayInfo methods. This elucidation will facilitate your comprehension of how the disassembly correlates with the behavior of your C++ code. I will elucidate the essential components of your assembly and correlate them with the relevant C++ code.

1. Correlating Assembly with the main() Function

C++ Source Code:

int main() {

int userChoice = 0;

do {

std::cout << "What would you like to do?" << std::endl;

std::cout << "1. DISPLAY the client list (enter 1)" << std::endl;

std::cout << "2. CHANGE a client's choice (enter 2)" << std::endl;

std::cout << "3. Exit the program (enter 3)" << std::endl;

std::cin >> userChoice;

if (userChoice == 1) { /\* code \*/ }

else if (userChoice == 2) { /\* code \*/ }

else if (userChoice != 3) { /\* code \*/ }

} while (userChoice != 3);

std::cout << "Exiting the program..." << std::endl;

return 0;

}

Key Assembly Code:

0000000000000000 <main>:

0: 55 push %rbp

1: 48 89 e5 mov %rsp, %rbp

...

22: 8b 05 00 00 00 00 mov 0x0(%rip), %eax # Retrieve user input

28: 83 f8 01 cmp $0x1, %eax # Check if user input is 1

2b: 74 13 je 40 <main+0x40> # If yes, jump to the code that displays the client list

...

49: 74 02 je 4d <main+0x4d> # Check if user input is 2

4b: eb ca jmp 17 <main+0x17> # If input is not 2 or 3, repeat

4d: 48 8d 35 00 00 00 00 lea 0x0(%rip), %rsi # Load the string to display "Exiting the program"

Clarification:

The push and mov instructions at the beginning establish the stack frame for the main function, a standard procedure in assembly to maintain the stack for local variables.

The juxtaposition The instruction cmp $0x1, %eax verifies if the value in the eax register, representing userChoice, equals 1. This pertains to the line if (userChoice == 1) in your C++ code.

The je instruction is a conditional jump that executes if the comparison result is equal. This indicates that if the user selects 1, the program transitions to the section where the client list is shown, analogous to your C++ code.

Likewise, the instruction cmp $0x2, %eax and the following commands address the remaining alternatives in your C++ code (otherwise if (userChoice == 2) and else if (userChoice != 3)).

The concluding lea and call directives manage the presentation of the "Exiting the program..." notification before to termination.

1. Correlating Assembly with ChangeCustomerChoice()

C++ Source Code:

void ChangeCustomerChoice() {

int choice = 0;

std::cout << "Enter the number of the client that you wish to change: ";

std::cin >> choice;

if (choice == 1) { /\* Change to Brokerage \*/ }

else if (choice == 2) { /\* Change to Retirement \*/ }

...

else { std::cout << "Invalid client number."; }

}

Key Assembly Code:

000000000000042d <\_Z20ChangeCustomerChoicev>:

42d: 55 push %rbp

42e: 48 89 e5 mov %rsp, %rbp

483: 83 f8 01 cmp $0x1, %eax # Compare choice with 1

486: 75 0e jne 496 <...> # Jump if choice is not 1

494: eb 62 jmp 4f8 <...> # Jump after handling the choice

...

494: 8b 05 00 00 00 00 mov 0x0(%rip), %eax # Read user input for choice

Clarification:

The assembly first preserves the current stack and establishes the frame for the ChangeCustomerChoice function.

The instruction cmp $0x1, %eax evaluates if the variable choice equals 1, analogous to the condition if (choice == 1) in C++.

If the result is not equal (jne), the program proceeds to the subsequent comparison or incorrect choice handling, analogous to the behavior of C++ logic transitioning between else if statements.

The assembly transitions to various parts contingent upon the value of choice (whether it is 1, 2, etc.), aligning with the flow of your C++ function.

1. Correlating Assembly with CheckUserPermissionAccess()

C++ Program:

bool CheckUserPermissionAccess() {

std::string username, password;

std::cout << "Enter your username: ";

std::cin >> username;

std::cout << "Enter your password: ";

std::cin >> password;

if (username == "admin" && password == "password") {

std::cout << "Access granted.";

return true;

} else {

std::cout << "Invalid username or password.";

return false;

}

}

Key Assembly Code:

0000000000000120 <\_Z25CheckUserPermissionAccessv>:

120: 55 push %rbp

121: 48 89 e5 mov %rsp, %rbp

13c: 48 89 c7 mov %rax, %rdi # Load the username

167: c7 45 bc 00 00 00 00 movl $0x0, -0x44(%rbp) # Store comparison result (password check)

//..

Clarification:

The mov instruction facilitates the management of input by storing it in memory or registers, shown by loading the username from input (std::cin >> username).

The assembly thereafter does memory comparisons to verify if the username and password correspond to the hardcoded values ("admin" and "password").

The outcomes of these comparisons are retained in a register (%eax or %rbp), and contingent upon the result, the program outputs either "Access granted" or "Invalid username or password," mirroring the functionality of your C++ function.

1. Correlating Assembly with DisplayInfo()

C++ Source Code:

void DisplayInfo() {

std::cout << "Client's Name Service Selected (1 = Brokerage, 2 = Retirement)" << std::endl;

std::cout << "----------------------------------------" << std::endl;

std::cout << "Bob Jones 1" << std::endl;

// ...

}

Key Assembly Code:

0000000000000241 <\_Z11DisplayInfov>:

241: 55 push %rbp

242: 48 89 e5 mov %rsp, %rbp

24c: 48 8d 3d 00 00 00 00 lea 0x0(%rip), %rdi # Load the address of the strings (client names)

// ...

Explanation:

The lea (Load Effective Address) instructions in the assembly code load the addresses of the strings (like client names and service selections) into registers so that they can be printed.

These lea and mov instructions correspond to the std::cout lines in your C++ code, where you display the client information.

### Security Analysis:

The C++ code exhibits several potential security flaws regarding user input management, authentication, and validation.

#### 1. **Input Validation in** main()

* The std::cin >> userChoice; does not validate whether the input is a proper integer, which may lead to unexpected behavior or crashes if a non-integer input is provided.
* Mitigation: Validate input before processing:

if (!(std::cin >> userChoice)) {

std::cerr << "Invalid input!" << std::endl;

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

continue;

}

#### 2. **ChangeCustomerChoice()**

* Lacks input validation and authorization checks, leading to potential information leaks.
* Mitigation: Add range checks and access control before modifying client data:

if (choice < 1 || choice > 5) {

std::cout << "Invalid client number." << std::endl;

return;

}

#### 3. **CheckUserPermissionAccess()**

* Issues:
  + Hardcoded credentials (admin, password) introduce a major security flaw.
  + No hashing or salting of passwords.
  + No rate-limiting or protection against brute-force attacks.
* Mitigation:
  + Replace hardcoded credentials with a secure authentication framework.
  + Use password hashing and salting techniques.
  + Enforce rate-limiting or account lockout after a number of failed login attempts.

#### 4. **DisplayInfo()**

* This function displays sensitive client data without authorization checks.
* Mitigation: Implement access controls to ensure only authorized users can access client data.

### Recommendations for Security Improvement:

* Implement strict input validation throughout the codebase.
* Use secure password storage techniques such as bcrypt or Argon2.
* Avoid storing sensitive data, such as credentials, in the source code.
* Mitigate brute-force login attempts with account lockouts or CAPTCHA systems.

// Example for proper input validation in C++:

if (!(std::cin >> userChoice)) {

std::cerr << "Invalid input!" << std::endl;

std::cin.clear(); // Clear error flag

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n'); // Ignore invalid input

}

This is the action I undertook to enhance security:

1. Input Validation and Sanitization

In main.cpp and other sections of the program that pertain to user input (such as usernames, passwords, and client selections), you may enhance the code by using input validation. At present, input from std::cin is used without validation.

Recommendations:

Restrict the length of inputs (e.g., for usernames) to prevent buffer overflows or denial of service attacks.

Sanitize inputs to avoid injection attacks or erroneous data. This is especially critical when user data is kept or used in sensitive procedures.

if (username.length() > MAX\_USERNAME\_LENGTH || password.length() > MAX\_PASSWORD\_LENGTH) {

std::cerr << "Input too long. Please try again." << std::endl;

return 1;

}

"Input exceeds permissible length." Please try once more." << std::endl; return 1;}

Enhanced Password Management

The password is now managed by SHA-256 hashing in bank.cpp, representing a commendable first measure. Nevertheless, there are supplementary measures you may use to enhance security:

Implement salting: Incorporate a salt when hashing passwords to mitigate rainbow table attacks. Every user must own a distinct salt that is kept with the hashed password.

Transition to a more robust password hashing algorithm: Utilize functions such as bcrypt or argon2, which are particularly designed for the safe hashing of passwords, as SHA-256 is susceptible to brute-force attacks owing to its rapidity.

Illustration using salting:

std::string hashPasswordWithSalt(const std::string& password, const std::string& salt) {

std::string saltedPassword = salt + password;

// Use a stronger hashing algorithm like bcrypt

// This is an example; you can use a bcrypt library or argon2 instead of SHA-256.

std::vector<unsigned char> hash(SHA256\_DIGEST\_LENGTH);

SHA256(reinterpret\_cast<const unsigned char\*>(saltedPassword.c\_str()), saltedPassword.length(), hash.data());

// Convert to hexadecimal string

std::stringstream ss;

for (unsigned char byte : hash) {

ss << std::hex << std::setw(2) << std::setfill('0') << static\_cast<int>(byte);

}

return ss.str();

}

1. Manage Invalid Input with Elegance

In ChangeCustomerChoice() and other routines, the program just verifies the validity of a number, neglecting to address erroneous or non-numeric inputs, which may result in unpredictable behavior or program failure.

Recommendations:

Verify the input's numeric status prior to casting or use.

Reset std::cin's error status upon receiving a non-integer input and prompt the user once again.

if (!(std::cin >> choice)) {

std::cin.clear(); // Clear error state

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n'); // Discard invalid input

std::cout << "Invalid input. Please enter a number." << std::endl;

continue;

}

1. Client Information Access Control

The ClientMap in your bank.h file contains confidential customer data, including usernames and hashed passwords. It is vital to guarantee that sensitive info is accessible just to authorized individuals.

Enforce access control verifications prior to permitting any actions on client data.

Implement logging for sensitive processes, such as login attempts and unsuccessful attempts to modify client data, to track possible unwanted access attempts.

1. CWE-119/120 Alert: Static Arrays

The alert about statically sized arrays in bank.cpp at line 53 indicates a possible buffer overflow hazard. Verify that any statically sized arrays or buffers have bounds checking, or transition to dynamically allocated data structures (e.g., std::vector).

Illustrative enhancement:

std::vector<char> buffer(BUFFER\_SIZE);

// Ensure buffer access is within bounds

if (index >= 0 && index < buffer.size()) {

// Safe to access buffer[index]

}

1. Error Management in deposit() and withdraw() Functions

Your deposit and withdrawal functions in the client.CPP manages transactions; nevertheless, it might enhance its functionality with supplementary safety checks and recording mechanisms. For example, you may document transactions and verify that erroneous transactions (such as overdrafts) are appropriately managed and reported.

Implement logging for each transaction, regardless of success or failure.

Mitigate integer overflow during deposits or withdrawals by verifying limitations prior to adjusting balances.

Eliminate Embedded Passwords

The CheckUserPermissionAccess() method has the hardcoded passwords "admin" and "password". This is a security concern and should be replaced by a system that verifies hashed passwords against securely stored hashes, using salting as previously indicated.

\