Project Title: Role-Based Access Control (RBAC) System

1. Objective To develop a security module that authorizes user actions based on their assigned "Role" rather than their individual identity. This project distinguishes itself from simple "User Management" by focusing specifically on the **authorization logic** and **permission enforcement** matrix used in enterprise environments.

2. Scope & Features

- **Role Definition:** Define distinct roles (e.g., Viewer, Editor, Admin) with specific permission sets.
- **Permission Logic:** Implement a "Gatekeeper" function that intercepts requests (e.g., "Open Document A") and checks if the user's role has the necessary rights.
- **Separation of Duties:** Demonstrate that a user with the Viewer role is strictly denied Editor privileges.
- **Access Denied Logging:** Create a secure log that records every time a user attempts an action their role does not permit.

3. Proposed Technologies

- **Language:** Python or C++.
- **Data Storage:** JSON or SQLite to store the "User-to-Role" and "Role-to-Permission" mappings.
- **Logic:** Simple conditional structures (If/Else) to validate permissions against the database.

4. Expected Outcome A demonstration tool where you can log in as different users (e.g., "Alice the Admin" vs. "Bob the Intern") and show that Bob is automatically blocked from performing Admin tasks, proving that the access control logic works correctly.

The Core Concept: "Who are you, and what can you do?"

In a standard system, you might give "Bob" permission to "Read File A." In **RBAC**, you don't give permission to Bob directly.

1. **Users** are assigned to **Roles** (e.g., Bob is an "Intern").
2. **Roles** are assigned **Permissions** (e.g., "Interns" can "Read" but not "Delete").
3. **Result:** If Bob moves to the "Manager" role, you just change his role tag. You don't have to rewrite every single file permission.

This system is the industry standard because it scales. If you have 1,000 employees, you manage 5 roles, not 1,000 individual permission sets .

---

1. Technical Architecture (How to build it)

A. The Data Structure (The "Database")

You need to store three things. A simple **JSON** file or **SQLite** database is perfect for this.

- **Users Table:** Links a person to a role.
    - Alice -> Admin
    - Bob -> Viewer
- **Roles & Permissions Table:** Links a role to what it can do.
    - Admin -> ['read_file', 'write_file', 'delete_file', 'add_user']
    - Viewer -> ['read_file']
- **Resources Table:** The "files" or "actions" in your system.
    - File_A.txt (Sensitivity: Low)
    - Payroll_Data.db (Sensitivity: High)

B. The "Gatekeeper" Logic (The Code)

This is the heart of your project. You will write a function (let's call it AccessControl) that runs every time a user tries to do something.

The Logic Flow:

1. **Intercept Request:** User Bob tries to delete File_A.txt.

2. **Lookup Role:** System checks Bob's role -> "Viewer".
3. **Lookup Permissions:** System checks "Viewer" permissions -> ['read_file'].
4. **Compare:** Is delete in ['read_file']? -> **NO**.
5. **Action:** Deny access and log the incident .

---

2. Step-by-Step Implementation Plan

Phase 1: The Setup (Data Entry)

Create a script to initialize your "dummy" users and roles.

- *Action:* Define a Python dictionary or JSON file.
- *Example Data:*

Phase 2: The Login (Authentication)

Create a simple login screen.

- *Input:* Username/Password.
- *Process:* Verify user exists.
- *Output:* Load the user's **Current Role** into the session memory.

Phase 3: The Enforcer (Authorization)

Create the function that checks permissions.

- *Code Concept (Python):*

Phase 4: The Audit (Logging)

Implement the "Access Denied Logging" feature .

- *Why:* In cybersecurity, knowing *who* tried to break in is as important as stopping them.
- *Task:* Whenever Access Denied happens, append a line to security_log.txt:
    - [2026-02-12 00:00:00] ALERT: User 'Bob' attempted 'delete' - DENIED.
