Access Control Models: DAC, RBAC and Their Secure Practical Implementation

Majji Hari Krishna
Dept. of Computer Science & Engineering
Indian Institute of Technology Madras
Chennai, India
cs25m028@smail.iitm.ac.in

***Abstract*—Access control is a fundamental pillar of information security, ensuring that users can only access resources for which they are explicitly authorized. Among the myriad of access control models, Discretionary Access Control (DAC) and Role-Based Access Control (RBAC) are two of the most foundational. This report provides a comprehensive examination of these two protocols, beginning with their theoretical underpinnings and concluding with a detailed report on a practical, secure implementation. For DAC, we explore its owner-centric philosophy and ACL-based implementation. For RBAC, we delve into its organization-centric approach and its alignment with the principle of least privilege. This paper then details the architecture and security mechanisms of a hybrid system that practically implements both protocols. We describe the approach, including a "Zero Trust" data model, a secure API, specific cipher algorithms (AES-GCM) and integrity-protection techniques (HMAC-SHA256) to protect data and metadata, and the core enforcement logic. The result is a prototype that bridges the gap between the abstract theory of access control and its tangible, secure application.**

***Index Terms*—Access Control, Information Security, DAC, RBAC, Access Control List (ACL), Least Privilege, Implementation, Flask, SQLite, Envelope Encryption, HMAC, Zero Trust, AES-GCM**

I. Introduction

In the digital age, the protection of sensitive information is paramount. Organizations, governments, and individuals rely on computer systems to store, process, and transmit vast amounts of data. Unauthorized access to this data can lead to catastrophic consequences. The primary mechanism for preventing such unauthorized access is the implementation of a robust access control policy.

Access control is the selective restriction of access to a resource. It is the process by which a system grants or denies requests based on a predefined security policy. The entities requesting access are subjects (users, processes), and the resources being protected are objects (files, databases).

Over the decades, several models have been developed, including Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Role-Based Access Control (RBAC). DAC is characterized by its flexibility, placing control in the hands of the resource owner. Conversely, RBAC takes a structured, centralized approach, assigning permissions based on an individual's role.

This paper is presented in two phases. Phase 1 (Sections II-VIII) provides a detailed comparative study of the DAC and RBAC protocols, dissecting their architectures, formal models, and theoretical applications. Phase 2 (Sections IX-XVI) presents a detailed report on the design and construction of a secure software prototype that implements a hybrid of both DAC and RBAC, focusing on the practical challenges and security solutions required to build such a system.

II. Abbreviations and Acronyms

TABLE I
Abbreviations and Acronyms

Acronym	Description
ACL	Access Control List
AEAD	Authenticated Encryption with Associated Data
AES-GCM	Advanced Encryption Standard - Galois/Counter Mode
AIK	Application Integrity Key
API	Application Programming Interface
DAC	Discretionary Access Control
FEK	File Encryption Key
HMAC	Hash-based Message Authentication Code
KMS	Key Management Service
MAC	Mandatory Access Control
MEK	Master Encryption Key
NIST	National Institute of Standards and Technology
RBAC	Role-Based Access Control
SoD	Separation of Duty
WSGI	Web Server Gateway Interface

III. Formal Models

To understand the precise behavior of access control systems, it is useful to define them with mathematical formalism.

A. Formalizing Discretionary Access Control (DAC)

The state of a DAC system can be described by an access control matrix (ACM), as proposed by Lampson [1]. This matrix, M, describes the rights of subjects over objects.

Let S be the set of subjects and O be the set of objects. Let R be the set of rights (e.g., read, write, own). An entry $M[s, o]$ where $s \in S$ and $o \in O$, contains a subset of rights $r \subseteq R$.

$$M[s, o] = \{r_1, r_2, ..., r_k\} \quad \text{where } r_i \in R \tag{1}$$

The "discretionary" aspect comes from rules that allow subjects to modify the matrix. For example, if 'own' $\in M[s, o]$,

then subject s can execute commands to add or remove rights for another subject s' over object o.

$$\text{grant}(s, s', o, r) \implies \text{add } r \text{ to } M[s', o] \tag{2}$$

This formalization clearly shows how permissions are tied directly to subject-object pairs.

B. Formalizing Role-Based Access Control (RBAC)

The NIST standard for RBAC provides a widely accepted formalization [2]. The core RBAC model consists of several sets and relations:

- U, R, P, S: sets of users, roles, permissions, and sessions.
- $PA \subseteq P \times R$: a many-to-many permission-to-role assignment relation.
- $UA \subseteq U \times R$: a many-to-many user-to-role assignment relation.

A user can exercise a permission p if and only if that permission is authorized for one of the user's active roles.

$$\exists r \in \text{session\_roles}(s) \text{ such that } (p, r) \in PA \tag{3}$$

This model introduces a layer of abstraction—the role—between users and permissions, which is the key to its administrative scalability.

IV. Discretionary Access Control (DAC)

DAC is an access control policy determined by the owner of an object. The owner has the discretion to grant or revoke access permissions.

A. Core Principles and Implementation

The fundamental principle of DAC is owner-controlled access. This is typically managed through an Access Control List (ACL). An ACL is a data structure associated with an object that lists all subjects (users or groups) that have been granted access rights to it.

B. Advantages of DAC

- **Flexibility and Ease of Use:** Owners can share resources quickly without a central administrator.
- **Simplicity:** The concept of ownership is intuitive.
- **Granularity:** Permissions can be assigned on a per-user, per-object basis.

C. Disadvantages of DAC

- **Susceptibility to Trojan Horses:** A program run by a user inherits all of that user's permissions, which a malicious program can abuse [3].
- **Lack of Centralized Control:** It is difficult to enforce a consistent, organization-wide security policy.
- **Administrative Overhead:** Tracking permissions in a large system is complex.

V. Role-Based Access Control (RBAC)

RBAC is a policy-neutral access control model where access decisions are based on the roles that individual users have as part of an organization.

A. Core Principles and Implementation

The central idea of RBAC is to introduce a level of indirection between users and permissions. Permissions are assigned to roles, and users are assigned to roles. A role represents a job function (e.g., 'Accountant', 'HR Manager'). This structure simplifies administration and enforces the principle of least privilege [4].

B. Advantages of RBAC

- **Simplified Administration:** Managing a few roles is simpler than managing thousands of individual users.
- **Enforcement of Least Privilege:** Users are only granted the permissions necessary to perform their job functions.
- **Scalability:** RBAC scales exceptionally well as an organization grows.
- **Policy Compliance:** RBAC helps organizations meet regulatory requirements.

C. Disadvantages of RBAC

- **Initial Complexity:** Defining a logical and effective set of roles requires a thorough analysis of business processes.
- **Role Explosion:** In large organizations, the number of distinct roles can become unmanageably large.

VI. Comparative Analysis: DAC vs. RBAC

While both models aim to secure resources, their philosophical and practical differences are stark, as summarized in Table II.

TABLE II
Comparison of DAC and RBAC Characteristics

Feature	DAC	RBAC
Control Locus	Decentralized; at the discretion of the resource owner.	Centralized; managed by system administrators.
Access Basis	Based on the identity of the subject (user or group).	Based on the assigned role(s) of the subject.
Primary Mech.	Access Control Lists (ACLs).	Role-Permission and User-Role assignments.
Flexibility	Very high; permissions can be changed dynamically.	Moderate; requires administrative action.
Scalability	Poor; unmanageable in large organizations.	Excellent; scales efficiently with growth.
Security	Focuses on ease of sharing.	Enforces principle of least privilege.
Environment	Personal computing, small workgroups.	Enterprise systems, government, healthcare.

VII. Uses and Real-Life Applications

A. Applications of DAC

DAC is prevalent in systems where user flexibility is key:

- **Desktop Operating Systems:** File permissions in Windows (NTFS), macOS, and Linux.
- **Social Media Platforms:** Choosing who can see a post or photo on Facebook or Google Drive.

B. Applications of RBAC

RBAC is the standard for structured, large-scale environments:

- **Enterprise Resource Planning (ERP):** An 'Accountant' role in SAP can access financial ledgers but not HR records.
- **Cloud Computing Platforms:** AWS IAM and Azure RBAC use roles to grant permissions to services and users.

VIII. Challenges and Future Trends

Neither DAC nor RBAC is a perfect solution. DAC's weakness is its lack of central control. RBAC's main challenge is managing complexity and potential "role explosion." The successor to RBAC is widely considered to be Attribute-Based Access Control (ABAC), which makes dynamic access decisions based on attributes of the subject, object, and environment [5].

IX. Phase 2: Secure Implementation Approach

The preceding sections (Phase 1) established the formal theory of DAC and RBAC. This second phase of the project focuses on the practical challenge: implementing a secure, hybrid system that combines both protocols.

The theoretical models are clean and abstract, but a real-world implementation must contend with numerous security threats. An attacker will not just try to bypass the logic; they will try to attack the *implementation itself*. Our approach was therefore based on a "Zero Trust" posture [9], which assumes that no component of the system, including our own database, is inherently trustworthy.

This led to two primary security goals for the implementation:

1) **Confidentiality:** All sensitive data (user files) and sensitive metadata (access control permissions) must be unreadable to an attacker, even one with full access to the database.
2) **Integrity:** The system must be able to detect and prevent any unauthorized modification of access control metadata by an attacker with database access.

To achieve these goals, we designed a system composed of a secure API, an encrypted database, and a robust client-side simulator. The real-world scenario modeled is a "Secure Hybrid Document Management System" (SH-DMS) where an organization sets baseline permissions via RBAC (e.g., 'Engineers' can write to the `/Engineering` folder), but individual users can then use DAC to share their specific files with others (e.g., an Engineer shares a file with a user from 'Marketing').

X. Development Environment & Tooling

The selection of a proper development environment is the first step in building a secure system. The primary goal was isolation and the use of industry-standard tools.

A. Python Virtual Environment (venv)

All development was conducted within a Python `venv`. This is a critical best practice for secure development, as it isolates project dependencies (e.g., Flask, cryptography) from the host operating system's Python installation. This prevents package version conflicts and ensures that the cryptographic libraries are explicitly version-managed for the project. For this project, `python -m venv venv` was used, and all packages were installed via `pip` from a `requirements.txt` file. This guarantees a reproducible and auditable build environment.

B. Core Technologies

- **Python 3.10:** The core programming language, chosen for its clarity and extensive support for security libraries.
- **Flask [6]:** A lightweight Web Server Gateway Interface (WSGI) web framework. It was used to build the secure REST API (`app.py`). Its "micro-framework" nature is ideal for this project, as it carries no unnecessary overhead and forces all security logic to be implemented explicitly.
- **SQLite 3 [8]:** A single-file, serverless database engine (`secure-data.db`). This was chosen for its simplicity of setup (no separate database server required) and ease of use in a prototype setting. Despite its simplicity, it is a robust, ACID-compliant database.
- **Cryptography Library [7]:** The `cryptography` library for Python. This is the *de facto* standard, providing modern, peer-reviewed, and secure implementations of cryptographic primitives. Using this library prevents common, self-implemented crypto flaws.
- **Visual Studio Code:** The IDE used for all development, providing integrated terminal and debugging capabilities.

XI. System Architecture

The system is designed as a classic three-tier architecture, but with security considerations at each layer. The components are:

1) **Presentation/Client Tier:** A command-line script (`client.py`) that simulates user actions.
2) **Application/Logic Tier:** The Flask server (`app.py`) that contains all business logic.
3) **Data Tier:** The SQLite database (`secure-data.db`) and the file system (`uploads/` folder).

A. Server Model (Flask/Werkzeug)

The server in this project is the `Flask` application, which is run by executing `python app.py`. By default, Flask uses the **Werkzeug WSGI development server**. It is important to understand its operation:

- **How it Works:** When started, it binds to a local port (e.g., 127.0.0.1:5000) and listens for incoming HTTP requests. As a development server, it is single-threaded and processes one request at a time.
- **Debug Mode:** We ran the server with `app.run(debug=True)`. This enables two key features: 1. *The Reloader:* Automatically restarts the

server when a code file is changed. 2. *The Debugger:* Provides a detailed, interactive traceback in the browser if an error occurs. This was essential for the iterative debugging process.

- **WSGI:** Flask (and Werkzeug) implement the WSGI standard, which provides a simple interface between the web server and the Python application.

In a production environment, this server would be replaced by a production-grade WSGI server like Gunicorn or uWSGI, which can handle multiple concurrent requests.

B. API-Client Architecture

The system is built on a strict API-centric design.

- **The Server (`app.py`):** This is the "brain" and the "fortress" of the system. It exposes a set of REST API endpoints (e.g., `/api/files/upload`, `/api/files/share`). Crucially, it contains 100% of the security and access control logic. No client is ever trusted to make a security decision.
- **The Client (`client.py`):** This script is "dumb." It holds no logic, no keys, and no permissions. It is merely a command-line tool to send HTTP requests to the server. For this project, authentication was simulated by sending a trusted `X-User-ID` header. In a real system, this would be replaced by an unforgeable JWT (JSON Web Token).

This separation ensures that even if the client application is compromised, the attacker cannot bypass the server's enforcement engine.

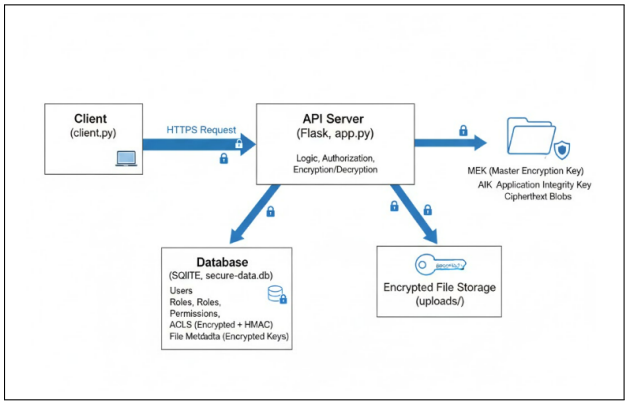

Fig. 1. System Architecture Diagram.

XII. Secure Database Schema

The foundation of the access control logic is the database schema, defined in `database.py`. The tables were designed not just for function, but for security.

- `users`: Stores user information. (e.g., 'alice').
- `roles`: Defines the RBAC roles. (e.g., 'Engineer').
- `permissions`: Defines the available actions. (e.g., 'read', 'write').
- `resources`: The "object" table, representing both files and folders. This table's design is critical:
  - `owner_user_id`: This column is the foundation of the entire DAC protocol.
  - `parent_id`: This column enables the folder hierarchy, which is essential for the RBAC-inheritance check.
- `user_roles`: A simple mapping table linking `user_id` to `role_id`. This is the implementation of the UA relation from the formal model.
- `role_permissions`: A mapping table linking `role_id` and `permission_id` to a `resource_id` (a folder). This is the PA relation.
- `file_metadata`: This table holds the *cryptographic* metadata for files.
  - `encrypted_file_key`: This column stores the encrypted FEK, forming the basis of our envelope encryption strategy.
- `acls`: This is the implementation of the Access Control List for explicit DAC. This table is the most heavily secured:
  - `encrypted_permissions`: Stores the granted permissions (e.g., '['read']') as an encrypted blob, ensuring confidentiality.
  - `hmac`: Stores an HMAC tag, ensuring the integrity of the entire row against tampering.

XIII. Cryptographic Design & Implementation

This project's core security objective was to protect data and metadata from all attackers, including a malicious insider with database access. This was achieved through a multi-layered cryptographic strategy defined in `security.py`.

A. Key Management (Simulated KMS)

In a production system, all cryptographic keys would be stored in a hardware security module (HSM) or a Key Management Service (KMS) like AWS KMS or HashiCorp Vault. For this prototype, we simulated this by storing two master keys in a `secrets.ini` file, which is explicitly forbidden from source control by `.gitignore`.

1) **Master Encryption Key (MEK):** An AES-256 key (32 bytes) used *only* to encrypt other, data-specific keys.
2) **Application Integrity Key (AIK):** A separate, secret key (32 bytes) used *only* for generating HMACs.

This separation of keys is a critical security principle. The key used for encryption should never be used for signing, as this can open theoretical attack vectors. These keys are loaded into application memory once on startup and represent the system's root of trust.

B. File Encryption (Confidentiality)

To protect the files themselves, we used an **Envelope Encryption strategy**. This avoids reusing the MEK for every file, which would be a cryptographic weakness (reusing a key in GCM mode with different nonces can be catastrophic).

The process, handled by `encrypt_file` and `decrypt_file_stream`, is as follows:

1) **On Upload:**
   a) A new, unique, cryptographically random 32-byte **File Encryption Key (FEK)** is generated.
   b) A random 12-byte **nonce** (IV) is generated.
   c) The file's contents are encrypted using this FEK and nonce with our chosen cipher algorithm.
   d) The FEK itself is then encrypted using the **MEK** (and a different nonce).
   e) The *encrypted* FEK, the nonce, and the ciphertext are stored.
2) **On Download:**
   a) The server retrieves the *encrypted* FEK and the file's nonce.
   b) It uses the **MEK** to decrypt the FEK, bringing the plaintext FEK into memory.
   c) It then uses this plaintext FEK and the file's nonce to decrypt the file's ciphertext, streaming it to the user.
   d) The plaintext FEK is immediately discarded from memory.

Security Benefit: An attacker who steals the database only gets encrypted files and encrypted keys. Both are useless without the MEK, which only exists in the application server's memory.

C. Cipher Algorithm: AES-256-GCM

The specific cipher algorithm chosen was **AES-256-GCM**. This was a deliberate choice over simpler ciphers like AES-CBC.

- **AES (Advanced Encryption Standard):** The global standard for symmetric encryption, approved by NIST. We use a 256-bit key length for a high security margin.
- **GCM (Galois/Counter Mode):** This is the critical part. GCM is an **AEAD** (Authenticated Encryption with Associated Data) mode. This means it provides both: 1. **Confidentiality:** It encrypts the data (as an output-feedback cipher). 2. **Integrity/Authentication:** It produces an "authentication tag" (a GCM tag) that verifies the data has not been tampered with.

By using AES-GCM, we ensure that an attacker cannot perform a "bit-flipping" attack on the encrypted files. If an attacker modifies even a single bit of the ciphertext, the GCM tag will not match during decryption, and the `cryptography` library will raise an 'InvalidTag' exception, automatically thwarting the attack.

D. Metadata Integrity (The "Crown Jewels")

The most significant security contribution of this project is the protection of the access control metadata itself. An attacker who can write to the database could simply grant themselves access by bypassing our API.

Threat: `UPDATE acls SET target_user_id = [ATTACKER_ID] WHERE resource_id = [TARGET_FILE_ID]`

Solution: We use a HMAC (Hash-based Message Authentication Code) to protect every row in the `acls` table.

- **Chosen Algorithm: HMAC-SHA256.** We use SHA-256 as the underlying hash function, keyed with our secret **AIK**.
- A simple hash (like SHA-256) is **not** sufficient, as the attacker could just re-compute the hash for their tampered data. HMAC requires the secret `AIK`, which the attacker (who only has DB access) does not possess.

The process, handled by `generate_hmac` and `verify_hmac`, is as follows:

1) **On Write (Share):** When an ACL is created for a user and resource:
   - The permissions (e.g., '['read']') are first encrypted to ciphertext.
   - A "message" is constructed that binds the permission to the user and resource: `message = f"{encrypted_perms}:{resource_id}:{target_user_id}"`
   - The system computes:
   $$\text{hmac\_tag} = \text{HMAC}(\text{AIK}, \text{message})$$
   - This `hmac_tag` is stored in the `acls.hmac` column.
2) **On Read (Access Check):** When the system fetches an ACL row:
   - It reconstructs the expected message from the retrieved data.
   - It computes the expected HMAC tag using the `AIK`.
   - It securely compares the expected HMAC with the stored HMAC from the database.

Security Benefit: If they do not match, the system knows the row has been tampered with. The request is immediately aborted, a critical security event is logged, and the attack is stopped. This makes the access control rules cryptographically read-only to anyone without the `AIK`.

XIV. The Hybrid Enforcement Engine

All security theory and cryptographic mechanisms are centralized into a single, hardened function in `app.py`: `check_permission()`. This function is the "Policy Decision Point" (PDP) for the entire application. All secure API endpoints *must* call this function before performing any action.

The function executes a hybrid check in a specific, logical order:

1) **Check Ownership (Implicit DAC):** The system first checks if the requesting `user_id` matches the `owner_user_id` in the `resources` table. If true, access is granted. This is the fastest and most common check.
2) **Check ACL (Explicit DAC):** If the user is not the owner, the system queries the `acls` table for a specific share. If an entry is found:

- It *first* verifies the HMAC. If invalid, the request is immediately denied.
- It *then* decrypts the `encrypted_permissions`.
- If the requested action (e.g., 'read') is in the decrypted list, access is granted.

3) **Check Roles (RBAC):** If both DAC checks fail, the system performs an RBAC check. This is the most complex logic, as it must account for permission inheritance.
   - The system identifies the resource's parent folder.
   - It performs a recursive query (using the `get_parent_folders` helper) to find *all* parent folders up to the root.
   - It then executes a complex SQL query that joins `user_roles`, `role_permissions`, and `permissions` to see if the user has *any* role that is granted the requested permission on the resource's folder *or any of its parents*.
   - If a valid RBAC permission is found, access is granted.
4) **Default Deny:** If all three checks fail, the request is denied.

This prioritized, multi-step process successfully implements the hybrid logic, allowing organization-level policy (RBAC) to set the baseline, while empowering users with fine-grained control (DAC) over their own resources.

XV. Threat Modeling & Security Analysis

By adopting this "Zero Trust" architecture, the system is secured against multiple attack vectors.

A. Threat Model

We considered two primary attackers:

1) **External Attacker:** A remote user on the internet with no valid credentials.
2) **Malicious Insider:** A disgruntled employee (e.g., a database administrator) with direct read/write access to the `secure-data.db` file and `uploads/` folder, but *not* to the application server's memory or secret keys.

B. Protection Against Attack Vectors

- **Against External Attacker:**
  - All access is denied by default. The attacker must guess a valid `X-User-ID`, which is a brute-force problem.
  - Even with a valid user ID, the server-side `check_permission` engine prevents them from accessing any resource for which they are not explicitly authorized.
  - All communication would be over HTTPS (TLS), preventing man-in-the-middle (MITM) attacks.
- **Against Malicious Insider (Read Access):**
  - **File Confidentiality:** The admin can access the `uploads/` folder but will only find AES-GCM encrypted blobs. They can access the `file_metadata` table but will only find *encrypted* file keys (FEKs). The files are unreadable.
  - **Permission Confidentiality:** The admin can see the `acls` table but cannot determine *what* permissions are granted (e.g., 'read' vs 'write'), as they are encrypted.
- **Against Malicious Insider (Write Access):**
  - **File Tampering:** Attempting to modify an encrypted file in the `uploads/` folder will cause the AES-GCM authentication tag to mismatch, resulting in an `InvalidTag` error on download. The attack fails.
  - **Permission Tampering:** This is the key defense. If the admin tries to `UPDATE acls SET target_user_id=1` (granting themselves access), the HMAC check will fail on the next read because the stored `hmac` field will not match the new, tampered data. The system will detect this, deny access, and log a critical security event. The admin is powerless to bypass the access control logic.

XVI. Implementation Learnings & Project Experience

This project bridged the critical gap between abstract theory and practical application. The process was iterative, and the debugging phase provided the most significant learnings.

A. The Theory-to-Practice Gap

The formal models for DAC and RBAC are clean and mathematical. The implementation, however, is a series of complex, practical decisions.

- "ACL" in theory is a row in a matrix. In this practice, it is an encrypted, integrity-protected, and complex-to-query row in a SQL table.
- "RBAC" in theory is a simple set relation. In practice, it is a recursive, hierarchical SQL query that must be optimized.
- The theory papers do not often discuss the most critical vulnerability: the protection of the policy data itself. Our Zero Trust model addressed this gap directly.

B. The Iterative Debugging Process

The development was not linear; it was a cycle of coding, testing, and fixing. This process was fundamental to understanding the system's inner workings. Key errors encountered included:

- **Environment Errors:** The first hurdle was a `PSSecurityException` on Windows, where PowerShell's `ExecutionPolicy` blocked the `venv` activation script. This was a lesson in how the host OS itself is a security layer.
- **Configuration Errors:** A `KeyError` from `secrets.ini` highlighted the dependency on a correct configuration; a missing key is a fatal, non-recoverable error.
- **Database Errors:** A `sqlite3.OperationalError: near "ALNULL"` was a simple typo (`NOT NULL`) that

brought the entire database creation script to a halt, reinforcing the need for precision.

- **Application Logic Errors:** The most complex bugs were in `app.py`.
  - An 'AssertionError' and later a 'TypeError' related to Flask decorators. This was a deep lesson in Python's metaprogramming, requiring the use of `functools.wraps` to preserve function metadata.
  - A 'TypeError: int() ... not 'NoneType'' occurred because the server did not correctly validate form data before processing, a common but critical vulnerability.
- **Cryptographic Errors:** A ''str' object has no attribute 'hex'' error was a simple data type mismatch. In cryptography, however, such a simple bug can corrupt the entire security chain. It highlighted the need for exact precision when handling keys, nonces, and ciphertext.

C. Key Takeaways

1) **Security is Holistic:** A secure system is not just one strong algorithm. It is the *entire ecosystem*: the `venv` for isolation, `.gitignore` to protect secrets, server-side-only logic, data validation, and finally, the cryptographic primitives.
2) **Centralization is Strength:** The decision to centralize all security logic in the `check_permission` function and all crypto logic in `security.py` was invaluable. It makes the system auditable and ensures that no matter which API endpoint is called, the same, correct security check is performed.
3) **Distrust Your Database:** The most valuable security lesson was learning to treat the database as a potential threat. Encrypting and signing the ACLs is the single most powerful security feature in this prototype, moving it from a standard application to a truly "Zero Trust" system.

XVII. Conclusion and Future Work

This project successfully completed both its theoretical (Phase 1) and practical (Phase 2) objectives. Phase 1 established the formal definitions, advantages, and disadvantages of the Discretionary and Role-Based Access Control models.

Phase 2 built upon this foundation to create a tangible, secure software prototype. By designing a "Zero Trust" architecture, we demonstrated that it is possible to build a system that is secure not only against external attackers but also against malicious insiders with direct database access. The use of modern, authenticated encryption (AES-GCM) and integrity checks (HMAC-SHA256) to protect not just the data, but the *access control rules themselves*, represents a robust and modern approach to security. This hybrid system successfully models a real-world scenario, enforcing organization-wide policy via RBAC while still providing the flexibility of user-centric sharing via DAC.

A. Future Work

This prototype is a strong foundation, but for a production-grade system, several enhancements would be necessary:

1) **Proper Authentication:** The `X-User-ID` header would be replaced with a secure authentication mechanism, such as OAuth 2.0 and JWTs (JSON Web Tokens).
2) **True Key Management:** The `secrets.ini` file would be replaced with a real, network-attached Key Management Service (KMS) or a Hardware Security Module (HSM).
3) **Production-Grade Server:** The Flask development server would be replaced with a robust WSGI server like Gunicorn, running behind an Nginx reverse proxy.
4) **Evolve to ABAC:** The next logical step would be to enhance the `check_permission` engine to include attributes (e.g., time of day, user location), evolving the system toward an Attribute-Based Access Control (ABAC) model.

Acknowledgment

I would like to express profound gratitude to **Prof. Manikantan Srinivasan** from the Department of Computer Science & Engineering at the Indian Institute of Technology Madras for his exceptional guidance and mentorship throughout this project. His invaluable feedback on the theoretical report (Phase 1) and his continuous support during the practical implementation and debugging (Phase 2) were essential to the successful completion of this work.

References

[1] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Communications of the ACM*, vol. 19, no. 8, pp. 461-471, Aug. 1976.

[2] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli, *Role-Based Access Control*, 2nd ed. Norwood, MA: Artech House, 2007.

[3] W. E. Boebert and R. Y. Kain, "A Practical Alternative to Hierarchical Integrity Policies," in *Proceedings of the 8th National Computer Security Conference*, 1985, pp. 18-27.

[4] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-Based Access Control Models," *IEEE Computer*, vol. 29, no. 2, pp. 38-47, Feb. 1996.

[5] V. C. Hu, D. F. Ferraiolo, R. Kuhn, et al., "Guide to Attribute Based Access Control (ABAC) Definition and Considerations," *NIST Special Publication 800-162*, Jan. 2014.

[6] A. Ronacher, "Flask - A Python Framework for Web Applications," *Flask Website*. [Online]. Available: https://flask.palletsprojects.com/

[7] The Cryptography Authors, "Cryptography," *Python Cryptographic Authority*. [Online]. Available: https://cryptography.io/

[8] D. R. Hipp, "SQLite," *SQLite Website*. [Online]. Available: https://www.sqlite.org/

[9] J. Kindervag, "Defining The Zero Trust Model," *Forrester Research*, 2010. [Online].