Danmarks Tekniske Universitet

02239 Data Security: Assignment 2

Authors

Sara Homme Daasvand - s241669
Eva Maria Benito Sanz - s243313
Jimena Bermudez Bautista - s243532

August 13, 2025

1 Introduction

In modern client/server applications, authentication and access control are important elements that guarantee that only authorized users can interact with sensitive systems and data. Communication between clients and servers needs a robust mechanism to validate the users' identities and permissions. This laboratory was divided into two parts: Authentication and Access Control. We address different challenges by implementing a secure mock print server, designed as a client/server application.

For the first part of the laboratory, the authentication was implemented using a password-based scheme, which includes password hashing, certification of credentials, and the management of user sessions. We handle the secure handling of passwords during storage and transport, and a session management mechanism was implemented to authenticate subsequent requests without requiring repeated logins. Additionally, it renews the session validity during active use while invalidating expired sessions, intending to have a more balanced usability and security.

The second part of the laboratory extends the system to implement two access control models: Access Control Lists (ACL) and Role-Based Access Control (RBAC). Both mechanisms were implemented with dynamically loadable policy files, encrypted for security, to allow runtime flexibility in adapting to organizational changes.

In the end, our application meets the following goals:

- **Secure authentication:** Provides secure access by only letting valid users in by using hashed passwords and secure session tokens.
- **Session management:** Uses real-time session validity with secure renewal and invalidation mechanisms.
- **Dynamic Access control:** Implements a flexible authorization, giving the option to use either ACL or RBAC models, for future changes in users and roles.
- **Security enhancements:** Encrypts sensitive policy files and tokens, ensuring confidentiality and integrity during storage and transport.

2 Authentication

Password-based authentication is foundational to security in client-server systems. It involves verifying the identity of users through their passwords, ensuring that only authenticated users gain access to sensitive information or services. A secure implementation must consider how passwords are stored, transported, and verified, as well as how sessions are managed after authentication.

2.1 Password Storage on the Server

When storing passwords, three primary approaches are evaluated for ensuring confidentiality and integrity: storing passwords in a system file, in a public file secured by cryptography, and in a database management system (DBMS).

System File In this method, passwords are stored in a protected file managed by the operating system. Access control lists (ACLs) and file permissions restrict file access to authorized system processes, typically leveraging OS-level security mechanisms such as SetUID on Unix. While simple, this method depends heavily on the OS for security and is less flexible for scaling or complex access requirements.

Public File with Cryptographic Protections In the public file approach, passwords are stored in a file accessible for reading by all users but only writable by system processes. Passwords are hashed (and possibly salted) with cryptographic algorithms (e.g., SHA-256, bcrypt) to protect confidentiality. Although widely used in older Unix systems, this method presents risks if cryptographic protections are compromised and poses challenges for secure updates.

Database Management System (DBMS) In the DBMS approach, passwords are stored in a database table secured by the database's access control mechanisms. Access control within the DBMS restricts who can read or modify password data. Passwords are typically hashed and encrypted before storage, enhancing confidentiality and providing robust control over access, integrity, and audit logging. The DBMSâs flexibility and support for encryption make it ideal for scalable, secure password management.

2.1.1 Comparison and Justification for the DBMS Solution

A DBMS is selected for secure password storage due to its robust security architecture, which includes:

- **Fine-grained Access Control:** Allows precise control over who can access password information, preventing unauthorized access.
- **Encryption and Hashing Support:** Enables secure password storage and retrieval, protecting against password theft.
- **Scalability and Flexibility:** Easily manages large user bases and integrates with other system components.
- **Auditing and Monitoring:** Facilitates tracking access attempts and modifications, supporting compliance and forensics.

2.2 Password Transport

In our system, addressing the password transport problem is essential to ensure secure communication between clients and servers. We have implemented both individual request authentication and authenticated sessions to provide robust security for user interactions. Importantly, we assume that TLS (Transport Layer Security) is being used throughout all communications. TLS encrypts the data transmitted between the client and server, safeguarding sensitive information such as passwords and session tokens from potential interception or eavesdropping.

During the initial authentication phase, a user provides their username and password to log in. These credentials are sent over a TLS-secured connection, ensuring that the information is encrypted and protected during transit. The server receives these credentials and verifies them against the stored hashed passwords. Upon successful authentication, the server generates a secure session token. This token is a unique identifier for the user's session and is also transmitted back to the client over the TLS connection, maintaining the confidentiality and integrity of the token.

With the authenticated session established, the session token is used to implicitly authenticate subsequent messages exchanged between the client and server. The client includes this session token in the headers or parameters of future requests instead of resending the username and password each time. This approach enhances security by reducing the exposure of sensitive credentials and improves efficiency by eliminating the need for repeated authentication checks using passwords.

The use of TLS continues to play a critical role during the session. All messages containing the session token are encrypted, preventing unauthorized parties from accessing or tampering with the session information. This ensures that the authenticated session remains secure throughout its lifetime.

If the authentication of certain requests requires the transfer of additional parametersâ-such as timestamps, nonces, or specific authentication tokensâthese are seamlessly integrated into the existing interface. For instance, the session token can be added as an additional parameter in the method signatures or included in the request headers. Because we are operating over TLS, these additional parameters are also protected during transmission, maintaining the overall security of the communication.

By combining TLS encryption with both individual request authentication and authenticated sessions, we effectively address the password transport problem. The initial authentication securely establishes the user's identity, and the authenticated session allows for secure and efficient ongoing communication. Any necessary additional parameters for authentication are incorporated into the interface and benefit from the same level of security provided by TLS. This comprehensive approach ensures that all interactions between the client and server are both authenticated and confidential, providing a secure environment for users to access and utilize the system's services.

2.3 Password verification

Our system's password verification is intimately linked to how we store passwords, affecting both security and authentication effectiveness. Instead of storing plain text passwords, we use the SHA-256 hashing algorithm to convert user passwords into unique, fixed-size hashes. This process ensures that the original passwords are not directly stored, enhancing security.

Currently, we do not use "salting" in our hashing process. Salting involves adding a unique, random value to each password before hashing, ensuring that even identical passwords result in different hashes. Without salting, the same password will produce the same hash, which can be a security concern.

When a user attempts to log in, the system retrieves the stored hash associated with their username. It then hashes the password they've entered using SHA-256 and compares this new hash to the stored one. If they match, access is granted; if not, authentication fails.

While using unsalted SHA-256 hashing simplifies verification, it introduces security vulnerabilities. Attackers could use precomputed tables, known as rainbow tables, to match hashes to potential passwords, especially if users share common passwords resulting in identical hashes.

To enhance security, we recognize the need to incorporate salting. By adding a unique salt to each password before hashing, we can prevent attackers from easily exploiting identical hashes. Additionally, adopting stronger hashing algorithms like bcrypt, scrypt, or Argon2 which are designed for password security and include salting by defaultâwould make our system more resistant to attacks.

Implementing these changes would involve storing salts alongside hashed passwords and adjusting the authentication process to use the salt when hashing input passwords. This would significantly strengthen our password verification mechanism, better protecting user information and improving overall trust in our system.

2.4 Session management

It would be tedious to go through an authentication process for every action in a client-server system. Session management solves this by allowing users to remain authenticated for a period of time. It is important to analyse the security of the session token because is the sole proof of authentication between client and server during its lifetime.

Session Generation The session token in this implementation is generated using a cryptographically secure random number generator (SecureRandom) to ensure that it's unique and unpredictable. The generated token is 256-bit value, which is then Base64-encoded for safe transmission and storage. After the token is generated, it is encrypted using AES-256 before being stored on the server. This approach enhances security by ensuring that even if an attacker gains access to the storage of the session tokens, the encrypted tokens are computationally infeasible to reverse without the encryption key. Additionally, using a secure random generator reduces the risk of token collisions and prevents predictable tokens, which could otherwise be exploited in brute-force or replay attacks.

Assumption of Secure Communication An important assumption in this implementation is the use of secure communication (for example TLS) between the client and server. A secure communication ensures that session tokens are protected during transmission, which prevents attackers from intercepting or modifying them. Without a secure transport, attackers can hijack sessions by stealing tokens through methods like session sniffing or Man-in-the-Middle attacks. TLS would make these types of attacks practically impossible because it encrypts all communication.

Session Lifetime To further protect the system, session tokens in this implementation have a limited lifetime defined by a timeout value (SESSION_TIMEOUT). When a token is issued, its expiration time is stored alongside it on the server. Every time a session token is used in a request the server checks if the current time exceeds the stored expiration time. If the token has expired, the session is invalidated and the user is required to log in again.

Threats Despite these measures, session management systems remain vulnerable to certain types of attacks, such as session hijacking and session fixation. Session hijacking involves stealing a valid session token, potentially through Cross-Site Scripting (XSS) or insecure communication. Session fixation attacks is when the attacker forces a user to use a predefined session token, which the attacker can then exploit. The risk of this type of attack is reduced by generating a fresh session token upon successful login, ensuring that tokens are unique and unpredictable every time.

In conclusion, this implementation leverages secure random generation, encryption, limited session lifetime, and secure communication to create a robust session management system. While certain threats like session hijacking, and session fixation remain concerns, the design choices significantly reduce their impact, making the system resilient to common attacks while maintaining usability for users.

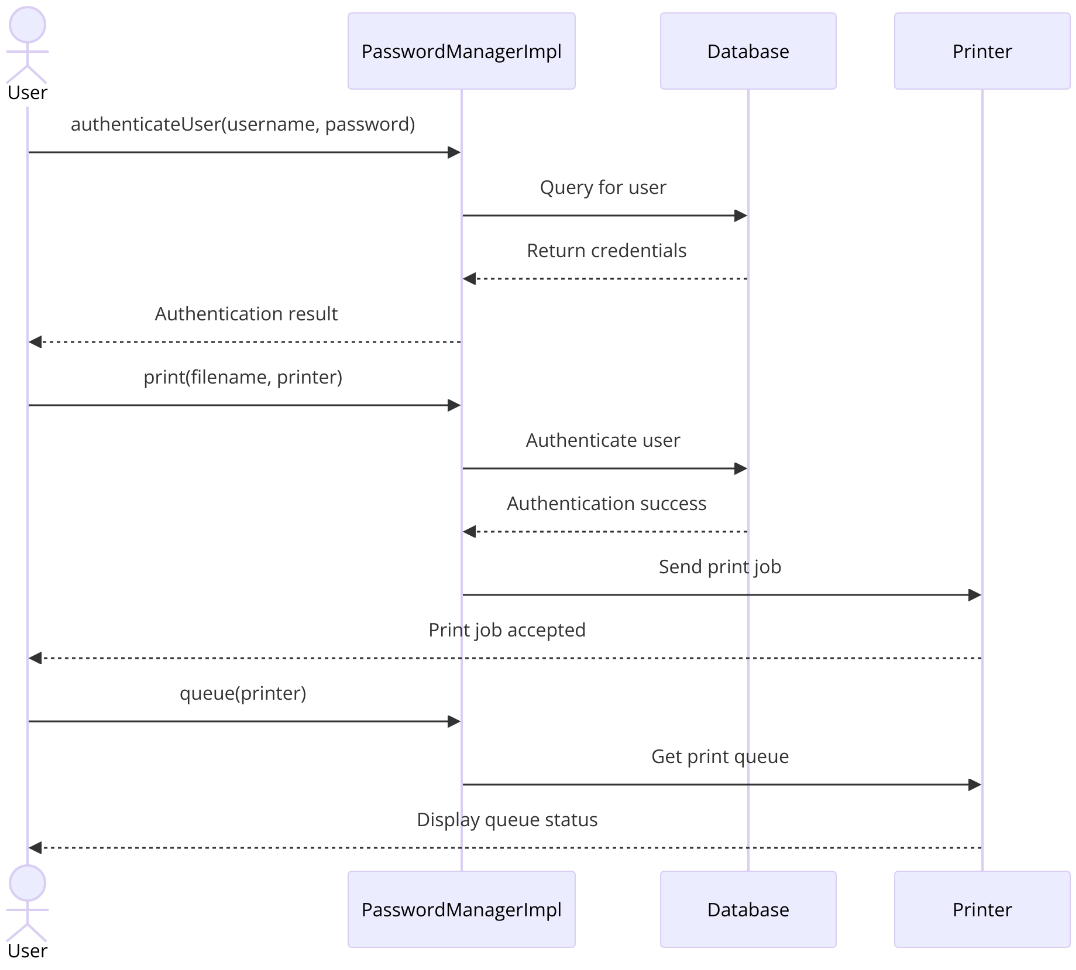
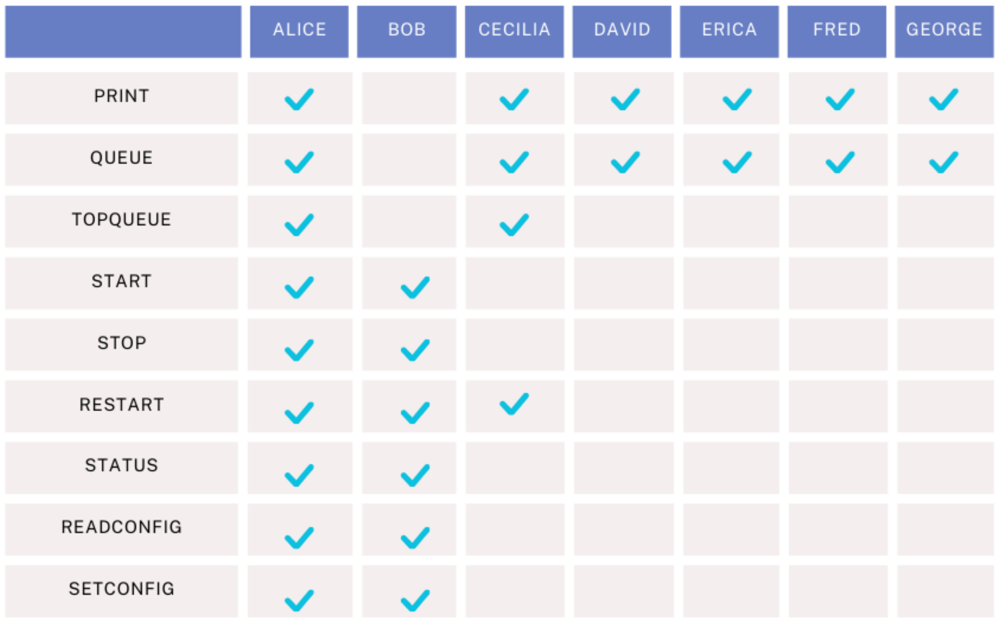


Figure 1: Effect of individual attributes

figure[1]

3 Access Control Lists

Firstly, we implemented an Access Control List (ACL) model to ensure the right permissions for our print server were implemented. For this as seen in Figure 2, we made a matrix to visually represent the relationship between the users (subjects) and the operations (objects). This mapping is *Identity Based Access Control* because individual users have individual permissions that we defined in an access control list file. The matrix contains all the operations that our print server can realize and shows with a tick the ones that the user can perform. By listing each user/operation pair we ended up having the design of the policies that needed to be implemented.



	ALICE	BOB	CECILIA	DAVID	ERICA	FRED	GEORGE
PRINT	✓		✓	✓	✓	✓	✓
QUEUE	✓		✓	✓	✓	✓	✓
TOPQUEUE	✓		✓				
START	✓	✓					
STOP	✓	✓					
RESTART	✓	✓	✓				
STATUS	✓	✓					
READCONFIG	✓	✓					
SETCONFIG	✓	✓					

Figure 2: Access control list

To implement this mapping we created a policy file that loads once the password manager server has been started. The policy file is external, which provides flexibility in changing it without having to change the source code. Our `acl_policy.txt` file contains this structure:

```
Alice: print, queue, topQueue, start, stop, restart, status, readConfig, setConfig
George: start, stop, restart, status, readConfig, setConfig
Cecilia: print, queue, topQueue, restart
David: print, queue
```

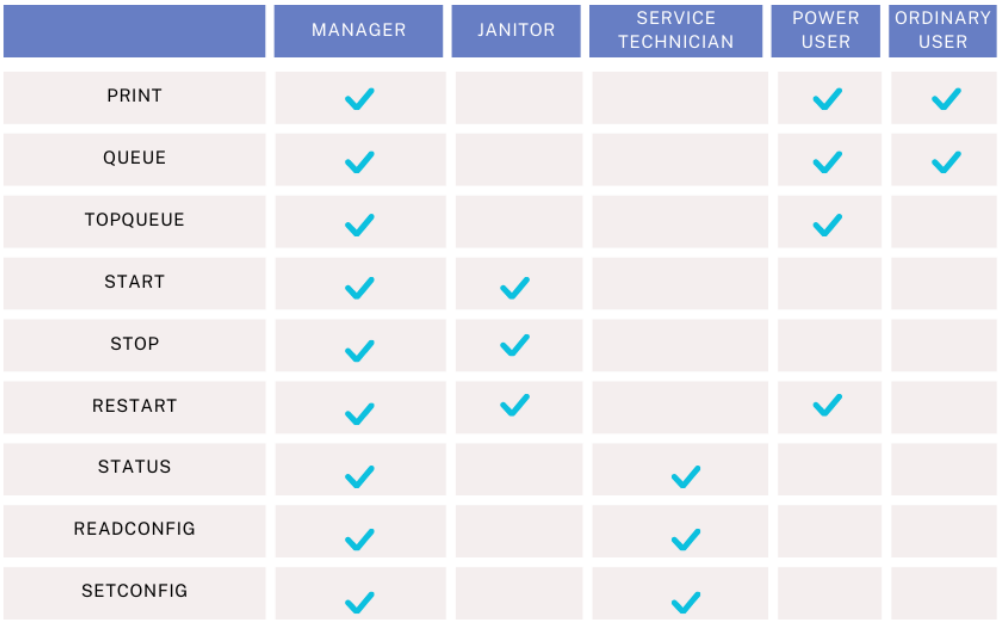
The data is parsed and loaded into a `Map<String, Set<String»` structure, where the key is the user's username and the value is the set of operations it has permission to perform. With this architecture during runtime, the server validates if the user authenticated has the required permissions to operate what it wants to perform. If the user is not allowed as defined in the policy, then the request is denied. The userâs authenticated session is used to match a specific session token, which is validated with each request to the server.

We decided to encrypt the policy file during server runtime using *AES encryption* to enhance the security of our approach to implementing the Access control list. When the `PasswordManagerServer` has started the policy file is decrypted into memory. We generate a unique AES 256-bit key using the library `javax.crypto` when the `PasswordManagerServer` is started for the first time and the generated key is stored in `policy_key.txt`. When the server is turned down, the policy is decrypted with the existing key, making it easier to modify the information.

4 Role-Based Access Control

4.1 Role Mining Results

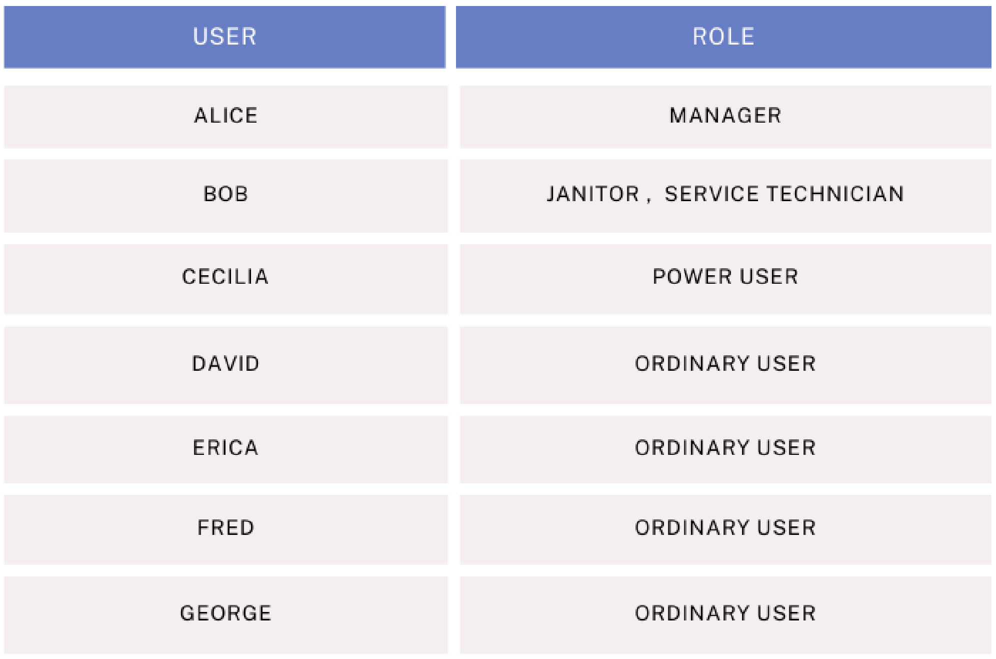
The implementation of a Role-Based Access Control (RBAC) for the print server required a structured approach to ensure that the permissions were assigned based on the original structure of the organization but now the permissions should be granted by roles. So firstly, by performing role mining on task 2 we defined five main roles needed: *Manager*, *Janitor*, *Service Technician*, *Power User*, and *Ordinary User*. We mapped these roles in a matrix as seen in Figure 3, to the operations each one has permission to perform.



	MANAGER	JANITOR	SERVICE TECHNICIAN	POWER USER	ORDINARY USER
PRINT	✓			✓	✓
QUEUE	✓			✓	✓
TOPQUEUE	✓			✓	
START	✓	✓			
STOP	✓	✓			
RESTART	✓	✓		✓	
STATUS	✓		✓		
READCONFIG	✓		✓		
SETCONFIG	✓		✓		

Figure 3: Access control list

We also made then a matrix as seen in Figure 4 to map the user with the role or roles that each one has, ensuring that the RBAC implementation was directly aligned with the organizational hierarchy and access needs. This mapping provided a clear and structured way to assign permissions based on roles, reducing redundancy and simplifying management.



USER	ROLE
ALICE	MANAGER
BOB	JANITOR , SERVICE TECHNICIAN
CECILIA	POWER USER
DAVID	ORDINARY USER
ERICA	ORDINARY USER
FRED	ORDINARY USER
GEORGE	ORDINARY USER

Figure 4: Access control list

4.2 Implementation of the RBAC Mechanism

The RBAC mechanism was implemented using external policy files to define roles and permissions dynamically. Two primary files are used:

1. `rbac_policy.txt` This file defines the roles and their corresponding permissions in the following format:

```
Role: permission1, permission2, permission3
```

An example of the contents of this file is shown below:

```
Manager: print, queue, topQueue, start, stop, restart, status, readConfig, setConfig
PowerUser: print, queue, topQueue, restart
OrdinaryUser: print, queue
```

2. `user_roles.txt` This file maps users to their assigned roles in the following format:

```
Username: Role1, Role2
```

For instance:

```
Alice: Manager
Bob: Janitor, ServiceTechnician
Cecilia: PowerUser
David: OrdinaryUser
```

These files are loaded the same way as the *Identity-Based Access Control* approach in the password manager server when it starts. This design choice enables administrators to adapt the system to changes in organizational roles or policies efficiently. The same as we did with the access control list approach, we provided security by using *AES encryption* using an AES 256-bit key. This way, every time the `PasswordManagerServer` is up, the `user_roles.txt` and `rbac_policy.txt` get encrypted.

5 Changes in the Policy

To make the policy changes, we made updates to the access control policies in response to organizational changes. The modifications include replacing Bob with George as the service technician, adding Henry as an ordinary user, and granting Ida the same permissions as Cecilia, a power user.

5.1 Updates in the ACL Policy

Firstly, we updated the permissions for each user to align with the new responsibilities and add the new users. To accomplish this we had to modify the `acl_policy.txt` that now ended looking as follows:

```
Alice: print, queue, topQueue, start, stop, restart, status, readConfig, setConfig
George: start, stop, restart, status, readConfig, setConfig
Cecilia: print, queue, topQueue, restart
David: print, queue
Erica: print, queue
Fred: print, queue
Henry: print, queue
Ida: print, queue, topQueue, restart
```

5.2 Updates in the RBAC Policy

For this approach, we managed the organizational changes by updating the `user_roles.txt` to now include the two new hires, delete Bob, and add George with his permissions as a Service technician. `user_roles.txt`

```
Alice: Manager
George: ServiceTechnician
Cecilia: PowerUser
David: OrdinaryUser
Erica: OrdinaryUser
Fred: OrdinaryUser
Henry: OrdinaryUser
Ida: PowerUser
```

5.3 Comparison between RBAC Policy and ACL

The implementation of both the Access Control List (ACL) and Role-Based Access Control (RBAC) policies brought some advantages and disadvantages, particularly when implementing organizational changes.

1. **Facility of policy updates:**
   - **ACL:** Policy updates required manual changes to the `acl_policy.txt` file for every individual user. For example, adding Henry as an Ordinary User and granting Ida the same permissions as Cecilia necessitated multiple updates to the policy file.
   - **RBAC:** Policy updates were more easy due to the hierarchical nature of roles. By assigning roles to users in `user_roles.txt`, we avoided repetitive changes. For example, replacing Bob with George as the `ServiceTechnician` only required a single update to the `user_roles.txt` file, leaving the role definitions in `rbac_policy.txt` unchanged.

2. **Scalability and Maintainability:**
   - **ACL:** ACL is more efficient for smaller systems where maybe there are not a lot of users, this is because when the system grows it's more difficult to maintain. The lack of role abstraction leads to redundancy in permissions and complicates policy management.
   - **RBAC:** RBAC is more scalable due to its role abstraction. By defining roles such as `Manager`, `PowerUser`, and `OrdinaryUser`, permissions were grouped by role, making it easier to manage. If there is a growth of users, modifying the role definition is enough without having to modify each user.

3. **Flexibility and Security:**
   - **ACL:** Provides flexibility because it gives specific permission to each user. However, this can result in a mistake while configuring the permissions and may result in a security vulnerability. In the end, you must verify all your users when you make a change, causing that consistency to be compromised.
   - **RBAC:** Has more balance between flexibility and security by centralizing permissions at the role level. This happens because the users with the same role have consistency and reduce the possibility of having misconfiguration. Additionally, the hierarchical structure supports inheritance, enabling roles to build upon one another if necessary.

Conclusion: Both ACL and RBAC showed good performance and helped establish the necessary access permissions for the users while maintaining security. The RBAC approach proved to be more efficient, scalable, and maintainable for bigger systems. The abstraction provided by roles made easier the implementation and ensured consistency across the system, making RBAC the preferred choice for long-term policy management.

6 Evaluation

This section evaluates how the implemented authentication and access control mechanisms satisfy the requirements defined in the previous sections. We demonstrate that the user is always authenticated by the server before any service is invoked and that the access control policies are effectively enforced for both ACL and RBAC models, both before and after policy changes. A summary is provided to indicate which requirements have been satisfied and which areas need improvement.

6.1 Authentication Verification

To ensure that users are authenticated before accessing any services, the server logs each service invocation, recording the username and the method name. This logging mechanism provides clear evidence that authentication occurs prior to service execution.

Logging Implementation The server uses a logging feature that writes entries to a logfile every time a method is called. Each log entry includes:

- **Timestamp**: The exact date and time when the service was invoked.
- **Username**: The authenticated user who initiated the request.
- **Method Name**: The name of the service method invoked.

If a user attempts to invoke a service without proper authentication or with insufficient permissions, the server denies the request and logs the unauthorized access attempt.

Session Token Validation Before executing any service method, the server validates the session token provided by the client. This validation process includes:

- Checking if the session token exists in the active sessions map.
- Verifying that the session token has not expired based on the session timeout setting.
- Ensuring that the session token corresponds to an authenticated user.

If the session token is invalid or expired, the server rejects the request and prompts the user to log in again.

6.2 Enforcement of Access Control Policies

The prototype enforces the defined access control policies effectively for both ACL and RBAC models. We tested the system before and after policy changes to confirm that permissions are correctly applied and enforced.

6.2.1 Access Control List (ACL) Enforcement

Before Policy Changes Using the initial `acl_policy.txt`:

- **Alice**: Had permissions to invoke all methods.
- **Cecilia**: Could perform `print`, `queue`, `topQueue`, and `restart`.
- **David**: Limited to `print` and `queue`.

Testing Results

- **Successful Access**: Users were able to perform actions according to their permissions.
- **Access Denied**: Attempts to invoke unauthorized methods resulted in access denial with appropriate error messages.

After Policy Changes The updated `acl_policy.txt` included new users and modified permissions:

- **Henry**: Added as an `OrdinaryUser` with permissions for `print` and `queue`.
- **Ida**: Granted the same permissions as `Cecilia`, able to perform `print`, `queue`, `topQueue`, and `restart`.
- **Bob**: Replaced by **George** as the `ServiceTechnician`.

Testing Results

- **New Users**: *Henry* and *Ida* were able to perform actions according to their assigned permissions.
- **Role Changes**: *George* had the permissions of a `ServiceTechnician`, and *Bob* no longer had access.
- **Policy Enforcement**: The system continued to enforce permissions accurately after the policy changes.

6.2.2 Role-Based Access Control (RBAC) Enforcement

Before Policy Changes Roles and permissions were defined in `rbac_policy.txt`, and users were assigned roles in `user_roles.txt`:

- **Roles**: `Manager`, `PowerUser`, `OrdinaryUser`, etc.
- **User Assignments**: Users were assigned appropriate roles reflecting their responsibilities.

Testing Results

- Users could perform actions permitted by their roles.
- Unauthorized attempts were denied, and appropriate error messages were provided.

After Policy Changes Updates reflected organizational changes:

- **User Updates**: *Bob* was removed, *George* assigned as `ServiceTechnician`, *Henry* and *Ida* added with respective roles.
- **Role Definitions**: Remained consistent, ensuring that role-based permissions were maintained.

Testing Results

- Role changes took effect immediately upon server restart.
- Users gained or lost permissions according to the updated roles.
- The system enforced the RBAC policies consistently after the changes.

6.3 Summary of Requirements Satisfaction

- **Secure Authentication**: *Satisfied.* Users are authenticated before accessing services, with credentials securely managed and session tokens validated.
- **Session Management**: *Satisfied.* Authenticated sessions are established, with session tokens used for subsequent requests and proper validation and expiration handling.
- **Password Storage and Verification**: *Partially Satisfied.* Passwords are stored hashed using SHA-256, but without salting, which poses potential security risks.
- **Password Transport Security**: *Satisfied.* The assumption of TLS usage ensures secure transmission of credentials and session tokens.
- **Access Control Enforcement (ACL and RBAC)**: *Satisfied.* Both ACL and RBAC policies are correctly enforced, both before and after policy changes.
- **Policy Adaptability and Updates**: *Satisfied.* The system allows for dynamic updates to access control policies, with changes reflected upon server restart.
- **Security Enhancements (Encryption of Policies)**: *Satisfied.* Policy files are encrypted using AES-256, protecting sensitive access control information.

6.4 Requirements Not Fully Satisfied

- **Use of Salting in Password Hashing**: The current password hashing mechanism lacks salting, making it potentially vulnerable to rainbow table attacks. Incorporating salting would enhance security and fully satisfy secure password storage requirements.
- **Advanced Password Hashing Algorithms**: While SHA-256 provides basic hashing, it is not the most secure option for password storage. Implementing algorithms like bcrypt or Argon2 would provide better resistance against brute-force attacks.

6.5 Overall Assessment

The implementation meets the majority of the specified requirements, providing a secure authentication system with effective session management and robust access control enforcement. The logging mechanism confirms that users are always authenticated before any service is invoked, and both ACL and RBAC policies are enforced accurately, even after organizational changes.

Areas identified for improvement include enhancing password storage security by incorporating salting and using more advanced hashing algorithms. Addressing these issues would further strengthen the system's resilience against potential attacks and fully satisfy all the initial requirements.

7 Conclusion

In this report, we addressed the challenges of implementing secure authentication and access control in a client-server environment by developing a mock print server. The main problems tackled included secure password storage and verification, safe password transport, effective session management, and the implementation of both Access Control Lists (ACL) and Role-Based Access Control (RBAC).

For authentication, we securely stored passwords in a database by hashing them with the SHA-256 algorithm. While this method avoids storing plain text passwords, we acknowledge that the lack of salting could make the system vulnerable to certain attacks. Future improvements include incorporating unique salts and adopting stronger hashing algorithms like bcrypt or Argon2.

To ensure secure password transport, we assumed the use of Transport Layer Security (TLS) to encrypt communications between the client and server, protecting sensitive data from interception. We also implemented authenticated sessions using secure session tokens, reducing the need to repeatedly transmit credentials and enhancing overall security.

Session management was strengthened by generating secure session tokens using a cryptographically secure random number generator and encrypting them. We enforced session timeouts to mitigate risks such as session hijacking. While these measures improve security, additional protections could further enhance the system.

In terms of access control, we implemented both ACL and RBAC mechanisms. The ACL approach assigned permissions directly to users but proved less scalable. The RBAC system streamlined permission management by assigning roles to users, improving scalability and ease of policy updates, especially when adapting to organizational changes.

Requirements Satisfied:

- Secure authentication with hashed passwords and session tokens.
- Effective session management with secure tokens and timeouts.
- Implementation of ACL and RBAC for flexible access control.
- Ability to update policies efficiently in response to changes.
- Encryption of sensitive policy files and tokens.

Requirements Not Fully Satisfied:

- Password hashing lacks salting, leaving potential vulnerabilities.
- Stronger hashing algorithms like bcrypt or Argon2 are not yet implemented.
- Additional measures are needed to fully address session-related threats.

Future Work:

We plan to enhance the system by incorporating salting and stronger hashing algorithms, improving session security, expanding access control features, integrating comprehensive logging and monitoring, and enhancing the user interface for better usability.

In summary, we successfully addressed the essential aspects of authentication and access control in a client-server environment, meeting most of the key requirements and establishing a solid foundation for future improvements.

A Source Code

The source code is in three different zip files attached, named:

- Authentication in the "datasecurityassigment2Sara_new"
- AccessControlACL in the "datasecurityassigment2accesscontrol"
- AccessControlRBAC "datasecurityassigment2rbac_branch"