Data security: Lab 3

Autumn 22, 02239
November 25, 2022

s174391, Sölvi Pálsson
s174434, Mads Esben Hansen
s180857, Ditte Aarøe Jepsen
s221969, Rémi Lejeune

Danmarks
Tekniske Universitet

Contents

1 Introduction

In previous lab, *Lab 2 - Authentication* an application for a print server was implemented with authentication mechanism where a subject (user) needed to be authenticated before access to the objects (methods) of the application was enabled. In this server prototype, all authenticated subjects were granted access to all of the methods. The aim for this lab is to extend the print server with access control so subjects are only granted access to objects as it is defined in the active access control policy.
The access control mechanisms are crucial for any IT system, as it is a mean (among others) to avoid fraud and end ensure confidentiality, integrity and availability.

"Access control is limiting who can access what in what ways" [1].

The problem of developing an access control policy for a system mainly concerns the trade of between ensuring confidentiality and integrity versus ensuring availability of information. It is a mechanical process where an already *authenticated* subject (user) either can or can not access a particular object (file, table, method etc.) in a specified way [1].

This will implement two different prototypes of access control mechanisms, *Access Control List* (ACL) and *Role Based Access Control* (RBAC).
When using ACL there is one access list for each object. Each list shows the access privileges each subject has to the respective object. An advantage from the ACL mechanism is the ability to set general default entries to each object respectively and furthermore, ACL provide a quick overview of access rights for a specific object.
RBAC assigns privileges to a subject based on the role-group they belong to in the system environment, and accesses are thus based on common needs of all members in the respective group and easily updated if a subject changes role within the system. [1].

Since IT system environments (often) are everchanging, the implementations must support the ability to update the access control mechanism according to changes in the policy in addition to enforcing current access rights.

2 Access Control Lists

This section describes the implementation of the access control policy, *Access Control Lists* (ACL), for the print server application. The ACL policy, protects the objects in the system by maintaining a list of pairs <subject userId, access rights> for every object in the system. A subject is granted access to the object if it's userId is in the access control list associated with the object and the access rights include the requested operation. If the subject's userId is not in the list or if the requested operation is not included in the subject's access rights, access to the object is denied.

2.1 ACL Prototype Implementation

The outline of the initial state of the implemented ACL policy for the print server can be seen in Figure 1. As previously stated, one of the advantages of ACL is that it is very easy to know who can access an object, Alice and Bob are for example the only subjects that have access to the start object, since Alice has access to all objects and Bob is the only one on the access control list for the start object. On the other hand, one of the disadvantages is that it can be difficult to know what objects a subject may access, except Alice who has access to all objects.

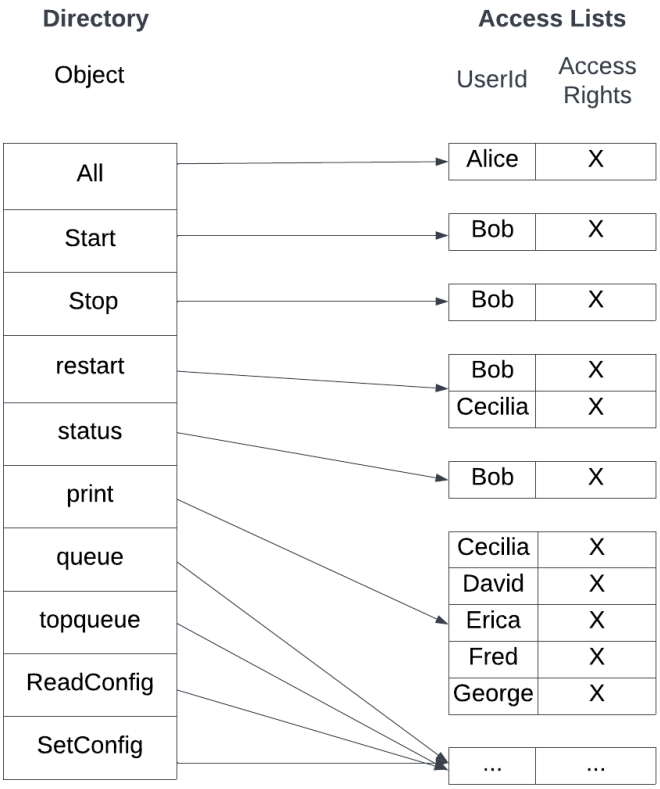


Figure 1 – Access Control List Policy for Print Server: Initial State

The policy outlined in Figure 1 is specified in an external policy file, *.access* that is loaded when the print server starts. In Figure 2, the content of the *.access* file can be seen, that captures the initial policy as described in Figure 1. Since the only access rights granted

for the print server is X (execute) it is assumed that is the requested operation and therefore it is not specified in the external policy file and not considered in the authorization process. When the print server is started the policy file is read into a private hash map with a method *readAccessFile*. The keys for the hash map are the names of the objects and the values are hash sets containing the userId's for the subjects that have access to the object. This way it is easy to look up if a subject should have access to an object.

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

Figure 2 – *.access* External Policy File : Initial State

One of the advantages of access control lists is that it can include general default entries that apply for any subject. Since there are no objects that should be available to everyone in the organization, there is no access list that has a default entry, and the authorization services does not check for it.

When a subject sends a request to the system to access an object it must first be authenticated as implemented in Lab 2 with a method *hasAccess*. For the implementation of the access control we have extended the method as it can be seen in Figure 3. The method first tries to authenticate the subject like implemented in Lab 2, and if successful the request continues to the authorization services. There, the request is evaluated against the access list for the requested object. The *hasAccess* method can be interpreted as a *reference monitor* for the system as it mediates all access by subjects to objects. It guards access to the objects by interpreting the access control policy.

```
private Boolean hasAccess(String usr, String pswd, String methodName){
    var database = (HashMap<String, String>) ApplicationServerListBased.hashMapFromTextFile(ApplicationServerListBased.pswdFile);

    // Authentication
    if(!database.containsKey(usr)) return false;
    if(!database.get(usr).equals(ApplicationServerListBased.hash(usr,pswd))) return false;

    // Authorization - Reference Monitor
    if(methodToUser.get(key: "all").contains(usr) || methodName.equals(anObject: "authenticate")) return true;
    if(methodToUser == null) readAccessFile();
    if(!methodToUser.containsKey(methodName)) return false;
    return methodToUser.get(methodName).contains(usr);
}
```

Figure 3 – *hasAccess* Authentication and Authorization

Sometimes administrators for a system need to revisit the access policy and determine if it is working as it should or if changes are required. For this purpose, we have implemented methods *addUser* and *removeUser* to add or remove users from access lists.

3 Role Based Access Control

This section implement an alternative prototype for enforcing a defined access control policy, role based access control (RBAC). Within the print server setting access rights are configured for each role as per below:

admin:	*all privileges*
technician:	*start, stop, restart, modify service parameters (status, readConfig, setCong)*
power:	*print, restart, manage queue (display queue, topQueue)*
ordinary:	*print, display queue*

User privileges are then determined based on the user's role(s). The RBAC setup for this prototype system will initially use the user-role mapping illustrated in figure 4 below to enforce the outlined access control policy.

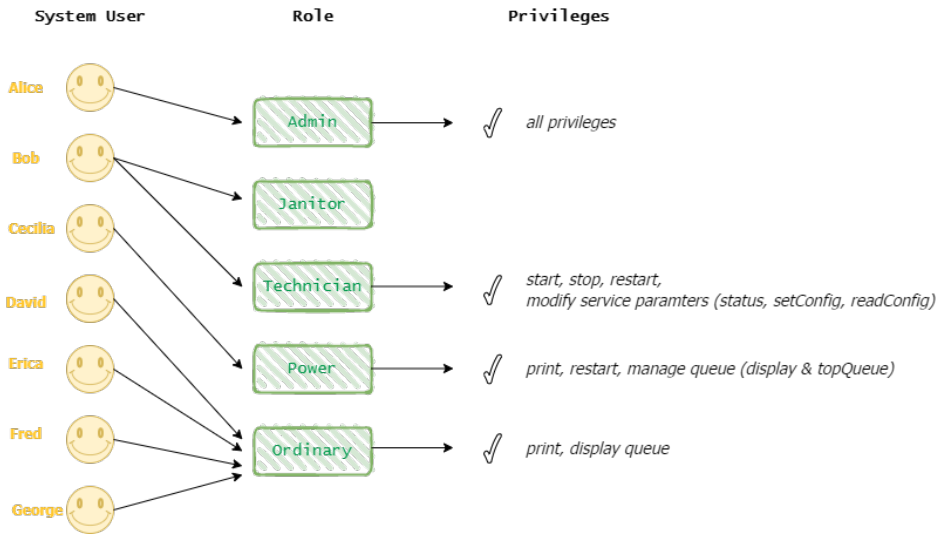


Figure 4 – Roles in print server prototype

From the diagram the advantages in terms of updating access rights for a certain user should also be clear. For instance, if George's role in the system changes (as will be presented later), he should simply be mapped to the corresponding change role/responsibility from which he will be granted the corresponding access rights.

3.1 RBAC Prototype Implementation

As was the case with ACL, the actual implementation of RBAC is an extended version of our authentication control prototype in lab 2. As described in lab 2, we decided on a public password file that can be read by everyone but only edited by the print server.
For this reason, we have implemented methods to read/write a map to a text file. In order to enforce RBAC, we create a mapping from user to a list of roles, which is written to a public readable (only readable!) text file, *roleFile*.

Whenever a method is invoked, the role file is re-read and compared to the roles that should be able to invoke the given method. The aforementioned roles that should be able to invoke a given method is determined by the *roleAccessFile*, which is a text file that

stores a map that maps from methods name to list of roles. This *roleAccessFile* is only read upon initialisation of the print server and is kept as a map in the object. The advantage of keeping this information in a external file instead of having it coded directly into the code, is that the privileges of each role, i.e., which methods they are able to invoke, can be changed without changing the source code.

Whenever a user attempts to invoke a method two check are performed by the RBAC mechanism. Both of these checks are implemented in the private method (only visible to the print server itself) *hasAccess*. The method first performs an authentication procedure checking if user/password is correct, i.e., the user exists in the system and the password is correct (done as described in lab 2).
Secondly, if the user is authenticated then the *hasAccess* method checks whether any of the roles assigned to the user (*roleFile*), are allowed to invoke the desired method (determined by *roleAccessFile*).

Listing 1 shows a code snippit containing the code for the *print* method. Notice that the *hasAccess* method is invoked with the roles from the map containing information about which roles should be able to invoke the method. Notice also that the methods serves as a wrapper for the private method print, that does not contain any checks, since this is done either on a single request basis, as shown here, or as part of a system wide authentication process.

Listing 1 – Role based access control method

```
1 public String print(String filename, String printer, String usr,
     String pswd) throws RemoteException {
2         if (hasAccess(usr, pswd, roleAccess.get("print") )) {
3             return print(filename, printer);
4         }   return "Access Denied.";
5     }
```

4 Evaluation

The mechanism should be able to create and delete users according to changes of users and their roles within the system. The creation of new users is implemented assuming that a new user changes an initially generated password by the server to a personal secret password.

4.1 Access control list

Before adding the access control list, we had a function *hasAccess* that would be called at the start of each method. This function would check if the user was in the database and if so they would be granted access otherwise they would not. We tested this by using unit tests that would test both when the authentication was successful and when it was not, manual testing was also used. The function *hasAccess* has both an implementation for authentication via user/password and via session ID.

Now that we have an access control list, we also need to check if the logged-in user should have access to the requested method. To do so, we modified the *hasAccess* to also check if the user should have access to the specified method. The implementation is shown in Figure 5, in which a parameter is added to *hasAccess* that should specify the requested method. The *hasAccess* method will then check if the user appears in the list of allowed users for this specified method. Again, this is implemented for both authentications via user/password and session ID.

```
/**
 * This method is used to check if the user has access to a method with authentication using password.
 *
 * @param usr Username of the user
 * @param pswd Password of the user
 * @param methodName Name of the method
 * @return Returns true if the user has access else returns false.
 */
14 usages  remi +1
private Boolean hasAccess(String usr, String pswd, String methodName){
    var database = (HashMap<String, String>) ApplicationServerListBased.hashMapFromTextFile(ApplicationServerListBased.pswdFile);

    if(!database.containsKey(usr)) return false;
    if(!database.get(usr).equals(ApplicationServerListBased.hash(usr,pswd))) return false;
    if(methodToUser.get("all").contains(usr) || methodName.equals("authenticate")) return true;

    if(methodToUser == null) readAccessFile();
    if(!methodToUser.containsKey(methodName)) return false;

    return methodToUser.get(methodName).contains(usr);
}
```

Figure 5 – Implementation of *hasAccess* for access control list

This was tested in the *LBACTest* file, we tested the *print* method with both Alice and Bob. Alice has access to it but not Bob, and as expected the testing shows the same results.

A few more features were added to help update the access control list and for quality-of-life improvements. First of all, we have added the possibility to have admins which have access to everything, this is done by adding a line in the *.access* file named all as shown in Figure 6.

```
print:Cecilia, David, Erica, Fred, George
queue:Cecilia, David, Erica, Fred, George
all:Alice
```

Figure 6 – .access file

Secondly, we added methods to edit the access control list.

- *editUsers* this method updates the *.access* file by removing the old list of users and adding a new list of users for a given method
- *addUsers* this method updates the *.access* file by adding a list of users to the old list of users for a given method
- *removeUsers* this method updates the *.access* file by removing a list of users from the old list of users for a given method
- *removeAllUsers* this method updates the *.access* file by removing all users for a given method
- *changeUsers* this method updates the *.access* file by for each of the old user, removing the old user and adding the new user

As required the access control mechanisms should support updates and change in user privileges. For ACL this can either be done by manually modifying the *.access* file or by using the above helper methods. All those methods got unit and manually tested to be sure that only authorised users are granted access. The result of the final task is shown in Figure 7.

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

(a) Before

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

(b) After

Figure 7 – *.access* file when updating user database.

4.2 Role Based Access Control

Firstly, we must ensure that the specified role based access control is actually enforced. In previous sections it was demonstrated how the *hasAccess* method is always called before doing anything, but this does not prove that it works.
junit-tests have been used in order to test whether the access control works as intended for the RBAC prototype. Specifically, we make sure that we can perform login with both Alice and Bob. After successful login for both parties, it is attempted to invoke the *queue* method from both users. Notice that Alice should be able to do so and Bob should not (refer to Figure 4 for the role diagram). The tests shows that Alice is not denied access while Bob is, i.e., the system works as intended.

Secondly, the prototype must be able to make changes to the users and their roles. Therefore, we have implemented extra methods in printer server (*addRole*, *removeRole*, *addUser*, *removeUser*), all of which only is callable from an *admin* user (Alice).
Likewise, junit-tests are performed in order to check that these additional methods work as intended. Specifically, we attempt to let Bob invoke *queue* which yields "Access denied." Afterwards, we add the role "power" to Bob (only Alice is allowed to do so) and let him try to invoke *queue* again. We see that Bob now has access as expected for all power users. This shows that the mechanism indeed is able to add roles to existing users.

In the assignment, we are asked to remove and add certain users and roles. Figure 8 present the user/password file before and after the updates to the users and roles. Notice that Bob is no longer present, i.e., not a user in the system anymore. Notice also that Ida and Henry have both been added as users.

```
Cecilia:f0877341a0a3949787ad8c9ae310e13bdc5d0345dfcd37dd8e7d59b39a7799c2
Bob:4733645adcf9fcd010ec1f4069a7f9262c6541a2b50b07675bf98880e54095d1
George:97ca0e50352241fe419639c171373700fcaa48792e40ba9590fa97e6991a1de1
Alice:97b63ad3b594fd4166d2d7c3574281f71dcb16746a6048825ef4b73029b87b8c
David:f28afed8115f25e3e5760adce9242ecea98d438e2fd28e0579cd5a327dbfa52b
Fred:0a1b9d6c10eec0969013f938eae0d0b5eb2729fde1e713841608d5aead12a88b
Erica:c1601fcc2fb90027971d61c261a2f0f7a1122583346bd5bf0d9547c0b0388bdc
```

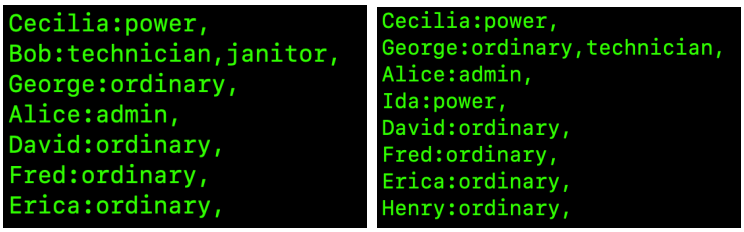
(a) Before

```
Cecilia:f0877341a0a3949787ad8c9ae310e13bdc5d0345dfcd37dd8e7d59b39a7799c2
George:97ca0e50352241fe419639c171373700fcaa48792e40ba9590fa97e6991a1de1
Alice:97b63ad3b594fd4166d2d7c3574281f71dcb16746a6048825ef4b73029b87b8c
Ida:72d7139021f535f3b3d30c1dbea953e145ce0ffb75dc326dae229c7e8bebadb2
David:f28afed8115f25e3e5760adce9242ecea98d438e2fd28e0579cd5a327dbfa52b
Fred:0a1b9d6c10eec0969013f938eae0d0b5eb2729fde1e713841608d5aead12a88b
Erica:c1601fcc2fb90027971d61c261a2f0f7a1122583346bd5bf0d9547c0b0388bdc
Henry:a39f7ae9d4eb88ac3681641f3cb2efb7d36e8812d1c9d74acd2541b8c931367e
```

(b) After

Figure 8 – Password file when updating user database.

Figure 9 shows the role file before and after update to the users. Notice that George has been given the extra role as "technician", Ida is a "power" user, and Henry is an "ordinary" user.



(a) Before **(b)** After

Figure 9 – Role file when updating user database.

Overall, we have demonstrated to our prototype enforces the role based access control policy as intended, and that we are able to update/change both the users and their respective roles. Thereby, we satisfy all requirements for RBAC, with no missing functionalities.

5 Discussion

The implementation of two different access control approaches, *access control list* (ACL) and *role based access control* (RBAC), for a print server has made different observations of advantages and disadvantages become apparent.

In general, ACLs are custom made accessibility lists for each method. As such, the admin has very good control on which users are able to invoke any given method. In our case, we do not differentiate between levels of rights - either a user has access or they do not. Therefore, all accessibility data can be stored in one file, and no handling for writing/reading privileges are needed.
As mentioned, the rights for each method are defined in lists of users. When a new user is added, the admin must update each list for the methods the user should have access to. Depending on the number of methods, this might be cumbersome. Additionally, whenever a user is removed, it is necessary to go through all lists for all methods and remove the user if present.

For the RBAC mechanism access rights are linked to what role(s) any user has. Consequently, more structure and a stronger hierarchy is present compared to ACL. This lab's implementation maintains two access files in order to enforce this type of access control. One file that links users to their respective role(s) and one file that links the roles to granted method accesses. This process ease the processes of removing/adding a user - you simply have to remove the row that defines their role(s). Additionally, if the scope of a role change, it is simple to amend the privileges of the role, and it will without further action update for the affected users.

Unfortunately, the mentioned advantages do not come without a cost. As stated, we had to maintain two access files to enforce the RBAC, which naturally comes with the added complexity of having multiple sources of potential errors. Furthermore, if any user does not fall into one of the existing roles, it might be necessary to make an entirely new role (if their access rights cannot be defined as a union of existing roles). Hence, role definitions should be carefully considered when setting up the entire system in order to keep confidentiality, integrity and availability at sufficient level without creating a way too complex role structure.

To summarise, when using the access control lists policy it is cumbersome to add/remove users, since everything is done manually. However, this also has the advantage that no extra work is needed if a user's access rights are quite custom, which means that confidentiality can be managed with higher granularity. This approach is well suited in a system with few users, or a system where each user have very personalised access rights and generally have access to few methods. The role based access policy makes it easier to add/remove users, but has the drawback that it relies on the users to fit into roles. In many systems many users fit naturally into roles and have access rights based on this, therefore this approach is widely used in larger real life systems.

6 Conclusion

In any IT system it is crucial to be able to control who can access what and in what ways in order to avoid fraud and ensure confidentiality, integrity and availability of information. This lab aims at enforcing a specified access control policy in a print-server setting using two different approaches: Access Control Lists and Role-Based Access Control.

Both prototypes satisfy that all users are identified through a password authentication process before any further actions. Next, any user of the system who attempts to invoke a method will be checked for the required privileges. If they are not granted access to an object as per the access control policy, both ACL and RBAC rejects the request. On the contrary, the request is accepted thus the method successfully invoked if the user has been granted access.
Furthermore, the two implementations enable amendments in the user privileges structure. Specifically, new users can be added to the system, current access rights can be changed for any user and lastly, users can be removed entirely from the system. All access right updates in the system can only be performed by authorised subjects.

Depending on the complexity and criticality level of a system environment, there are advantages of choosing one access control mechanism over others. In larger domains the RBAC mechanism advances in the ease of updating privileges for either one specific subject or an entire group of subjects. Though, RBAC in such system requires that the information/methods needed for each group is carefully considered such that availability is not overly prioritised at the expense of breaching required confidentiality and integrity goals. On the other side, the advantage of RBAC is lost if too many role-groups are defined.

As an alternative, ACLs provide a good overview of which users have access to one specific method, as each method is linked directly to the subjects with granted access. ACLs allows customised access rights which is a force in balancing availability, confidentiality and integrity and for a complex environment structure. However, it comes at cost of efficiency; each list (number of methods) must be updated whenever a new user is added to the system as well as whenever privileges are amended for a single user. It might also be hard to identify all privileges of one subject and thereby check if a user by mistake is granted access to an undesired object.

The entire access control mechanism is based on the assumption that only the intended subjects are able to amend access rights. All security goals are therefore at high risk of being attacked if intruders are able to authenticate as administrators who are allowed to implement changes in access rights. Therefore, a more complex authentication process should be prioritised for these subjects.

References

[1] C. P. Pfleeger, S. L. Pfleeger, and J. Margulies, *Security in Computing*. Prentice Hall, 2015.