CS745: Assignment 1 Sample Solution

Q. 7.1

1. **Trojan Horse:**
   Suppose there are two subjects S1 and S2 and one object O1. S1 has read and write privilege for O1. Suppose S2 put a program (trojan horse) on S1's system which will start a process to access O1. Since the process is running on S1's system then the access permission for that will be that of S1's, i.e., read and write over O1. This process can send the content read from O1 to S2. In this way S2 can access O1. Since the program is running on S1's system, so it will be allowed by the DAC model (which it should not).

2. **Information Flow:**
   Once the particular information is acquired by a process, and then DAC do not have any control on the flow of information. Suppose subject S1 has read privilege on document d1. S1 will read d1 and produce the exact copy of d1 as d2. Now the owner of d2 is S1 and he can give any permission to anyone or store it for its personal use. Here also DAC is failed to stop the duplicacy of secret info.

3. **Limited Negative Authorization Power:**
   Suppose there is an organization and it has certain Object O1. To restrict particular user to access O1, we have to grant access to O1 for remaining users, which is very complex thing to implement. This become more complex when there is frequent change of privileges.

Q. 7.2

(a)

Suppose there is an object name O designated to an object Obj but is later recycled to designate object Obj', then by holding <O,priv> long enough, a user eventually gets privilege to access Obj' (even though access to Obj was what had been authorized). And if an object name O designated object Obj in one user's address space but Obj' in another's, then transferring <O,priv> unwittingly gives the recipient privilege to Obj' (even though a capability for Obj was being transferred).

(b)

The main problem is different name to one object. So one approach to solve this problem is to name object such that single, global virtual address space is used. Suppose an object that

occupies *len* bytes and starts at virtual address *v* should be given name <v,len>. This will be unique in today's processor because virtual address space is large enough (64 bits) for distinct objects each to be assigned distinct virtual address. The object's length is being incorporated into its name to ensure that distinct but overlapping objects nevertheless have different names.

Another approach to solve this problem is to use equivalence class of each object. Different names of object Obj will belong to same equivalence class. While checking for any privilege, check whether the name of the object belong to any equivalence class of the object, and then check whether that subject have the privilege to that equivalence class or not.

Q. 7.3

In DAC policy, the initial assignment and subsequent propagation of all privileges associated with an object are controller by the owner of that object and/or other users whose authority can be traced back to the owner.

An authorization relation Auth <P,O,op> is a relation is principal (user) P hold privilege op for object O. any DAC policy can be circumvented if principals are permitted to make arbitrary changes to Auth. To characterize permitted changes to Aut, a DAC policy includes commands. Each command specifies a parameter list: a Boolean precondition, and an action. If the command is invoked and the precondition holds, then the action is executed. The precondition and action are permitted to name constants, Aut, and the formal parameters.

In DAC, network administrator permits the subjects holding resources to decide on accepting or rejecting the access to the resources at their sole discretion. This access model is based on the data owner and subject has required authority with certain constraints to determine which objects are accessible.

As long as these conditions are satisfied by Authorization relation Auth and set of commands C, the access policy is DAC.

Q. 7.4

(a)

At most one principal has privilege for accessing each given object. Given the DAC table, as long as the set of subjects and set of objects are available (static), one can say whether at most one principal has privilege for each given object. It can be determined by considering all the possible indirect flows. Here, in general, only owner will get privileges over the objects. In case of transferring to other owner, check ACL it should have only 1 element.

(b)

Any principal granted privilege w for an object is also granted privilege r. This can be enforced by access control policies as follows:

WriteConst (P, O, op): command

Pre: invoker(P) ^ (<P,O,owner> ∈ Auth) ^ (O = [particular object O]) ^ (op = w)

Action: Auth = Auth U {<P,O,w>, <P,O,r>}

(c)

Principal Po is never granted privilege r for object Objo. There is no such access policy which stores the previous list of constraints. So there is no way of tracking down whether Po has r permission on Objo or not.

But this problem can be solved by using log_files. For each grant or revoke command, log that entry into the log_file. Now to check whether Po had r on Objo or not just refer to this file.

(d)

No principal is ever granted a privilege that would subsequently allow principal Po to be granted a privilege r for object Objo. The drawback of DAC is, it has problem in implementing the negative access rule. The above constraint is also a negative constraint. For this to enforce, the system has to check each of the principal which is related to Objo, and based on this data it has to decide whether Po can be granted r on Objo. Since we know that privilege propagation is undecidable problem, so above constraint is hard to implement.

Q. 7.6

(a)

Since every user U of a file system has a separate directory Du. So if any user tries to get into another user's directory it won't be allowed (condition for 'link' is defined later).

Each file in the directory contains the r or rw privilege according to permission given to that user. Also it contain the list of users authorized to 'link' that file. If any user other than U tries to access the file inside directory Du, it is allowed only to access that memory segment in which that file is stored and restrict any other memory access by the other user's. This

restriction will help to prevent malicious user which can get the location of current file and then read other memory blocks where file is not stored.

Following commands can be implemented in this fashion:

Create (u, priv, file_name)

Pre: $u \in U$

Command: 1. Create file in Du with 'file_name' name.

2. Assign priv in privileges of 'file_name'.

3. Make 'link' as null.

Delete (u, file_name)

Pre: $u \in U$ ^ file_name is in Du

Command: 1. Delete file_name

Read (u, file_name)

Pre: $(u \in U) \vee (u \in$ [users in 'link' of file_name] ^ priv(u) $\supseteq r)$

Command: Allow reading

Write (u, file_name)

Pre: $(u \in U) \vee (u \in$ [users in 'link' of file_name] ^ priv(u) $\supseteq$ rw)

Command: Allow writing

Link (u, file_name, u')

Pre: $(u \in U) \vee$ (file_name is in Du)

Command: Append u' in access list of file_name.

Unlink (u, file_name, u')

Pre: $(u \in U) \vee$ (file_name is in Du) ^ $(u' \in$ access_list of file_name)

Command: Remove u' from access list of file_name.

(b)

Create_res_note (user, amount)

Pre: (if user is valid) ^ (amount is valid)

Command: 1. Add amount in user's account.

2. Create token for 'amount' for 'user'.

Delete_res_note (user, amount)

Pre: (if user is valid) ^ (amount is valid) ^ (token for <user,amount> exist)

Command: 1. Delete token.

2. Deduce amount from user's account.

Transfer_res_note (user1, amount, user2)

Pre: (if user1 and user2 are valid)^(amount is valid)^(<user1,amount> is valid token)

Command: 1. Delete_res_note (user1, amount)

2. Create_res_note (user2, amount)

(c)

Submit_solution (s, soln)

Pre: (s ∈ student) ^ (soln not empty)

Command: Create file under folder s.

Read_ans_key (s)

Pre: (s ∈ student) ⋁ (s ∈ grader) ⋁ (s ∈ professor)

Command: Read access granted.

Write_ans_key (s)

Pre: (s ∈ professor)

Command: Write access granted.

Read_grade (s, s')

Pre: ((s ∈ student) ^ (s = s')) v (s ∈ professor)

Command: Show grades of s'.

Assign_grade (s, s', grade)

Pre: (s ∈ grader) ^ (s' ∈ student) ^ (grade not assigned yet)

Command: Assign grade to s'.

Write_grade (s, s', grade)

Pre: (s ∈ professor) ^ (s' ∈ student)

Command: write 'grade' to s' grade.

Read&Annotate_soln (s, s')

Pre: (s ∈ grader) ^ (s' ∈ student)

Command: Give access to read and annotate the solution of s'.

Post_assignment (s)

Pre: s ∈ professor

Command: Allow posting the assignment.

This system can be implemented as DAC.

Q. 7.10

(a)

Since system does not include support for groups, then at max each principal will have every privilege. Therefore, the longest possible access list will have $n \times m$ length.

(b)

1. System support for group comprising subsets of the original n principals.
   ∴ Maximum group possible = Max. Subset possible with at least one principal
   $= 2^n - 1$
   ∴ Length of longest access control list = $(2^n - 1) \times m$
2. If our concern is with access authorization but not with review of priviledge or with changes to group compositions or to the privilege granted to each principal, there is no need to maintain the privilege corresponding to the group. Assigning that particular privilege to that principal will be sufficient.

(c)

Same as (b).

Q. 7.13

(a)

Consider the following access control matrix:

	User 1	User 2
File 1	r,w	r
File 2	r	w

Capability list corresponds to a user, what right he has, whereas access control list corresponds to the user which tells which user has what right on it.

For example, capability list for User 1 says that it has <r,w> rights on File 1 and <r> right on File 2. Access control list for File 2 says that User 1 have <r> and User 2 has <w> rights on it.

A capability list can be transformed into the ACL by creating such access control matrix or otherwise. And it has shown that ACL is preferred over capability list.

(b)

Encryption can be used to scramble an input and produce enciphered text that is hard to decipher on the key. Suppose that the kernel has a key k that is secret. Assume that the kernel can compute

$$[object \,||\, right]_k$$

The object name concatenated with access right and encrypted using the key k. The kernel can then return this encrypted string as the capability. However, we also need to ensure that random processing by a subject won't also produce something meaningful. To do this, a capability is

$$object : \; [object \,||\, right]_k$$

Once capabilities are implemented in this way, it is only necessary for the kernel to maintain the secret key k in storage.

Q. 7.16

Suppose send and receive routines might be implemented to pass capabilities from one process to another (whether or not those processes share a credit). Specifically, a process P could invoke send, giving a destination process P' and identifying some capabilities that could then be buffered at P' for receipt; by invoking receive, P' would cause any buffered capabilities to be moved to a specified c_list and would obtain the indices where they are stored.

This capability passing is similar to the message passing from processes to another. This supported by the kernel. This may use some flat but set at processes memory in kernel that will say some message is coming or the system can trigger some interrupt that will notify the process about the message. It depends on system to system.