Discretionary Access Control (DAC)

Nicola Zannone
Outline

Access Control

Discretionary Access Control

Safety Problem

DAC weaknesses
Access Control

Discretionary Access Control

Safety Problem

DAC weaknesses
Computer Security Objectives

- **Confidentiality**
  - Information disclosed only to principals authorized to know it
  - **Privacy**:  
    - right to be let alone (Warren and Bradeis, 1890)  
    - right of the individual to decide when, what, why and who manage his personal information (Westin, 1970)  
    - freedom from unreasonable constraints on the construction of one’s own identity (Agre, 1999)

- **Integrity**
  - Information modified only by authorized principals and in authorized ways

- **Availability**
  - Information accessible when it is needed (No denial of service)
Access Control

- **Goal**: Protect confidentiality and integrity of information
- Control what a subject can do to prevent damage to the system
- Regulate the operations that can be executed by a subject on data and resources
- Typically provided as part of operating systems and of database management systems
The basic idea of access control is that there is an active subject requiring access to a passive object to perform some specific access operation.

A reference monitor grants or denies access.
Subjects

- Active entity performing operations in the system

- Subjects can be:
  - users: single individuals connecting to the system
  - groups: sets of users
  - roles: a function or a position within an organization
  - processes: executing programs on behalf of users

- Relations may exist among the various types of subject
Objects

- Any system resource
  - file, printer, etc.

- *Protection objects*: objects controlled by access control system

- Note: not all resources managed by a system need to be protected

- *Advanced Objects*: subjects
Access Rights

- Operations that a subject can execute on protection objects
- Each type of operation corresponds to an *access right*
  - access control must be able to control the specific type of operation
- The most simple example of access rights is:
  - read: look at the contents of an object
  - write: change the contents of an object
- Other types of rights depending on the resources to be protected
  - execute, select, insert, update, delete, etc.
- **Advanced Rights**: ownership, delegate, remove
Subjects, Objects, Access Rights in Unix

- **Subjects**: users, groups
- **Objects**: files, directories
- **Access rights**: read, write, execute
  - For files
    - read: reading from a file
    - write: writing to a file
    - execute: executing a (program) file
  - For directories
    - read: list the files within the directory
    - write: create, rename, or delete files within the directory
    - execute: enter the directory
Access Control vs. Authentication

Access Control

Completely different things

Authentication:
Establishing who you are whether a user possesses a certain pseudonym/attribute or not

Access Control:
Establishing if a user has the right of doing a certain operation

Authentication is necessary for access control
Access Control vs. Authentication

- Completely different things

- Authentication:
  - Establishing who you are
    - whether a user possesses a certain pseudonym/attribute or not

- Access Control:
  - Establishing if a user has the right of doing a certain operation

- Authentication is necessary for access control
Policies, Models, Mechanisms

- **Policy**: define (high-level) guidelines and rules describing the accesses to be authorized by the system

- **Model**: formally define the access control specification and enforcement

- **Mechanism**: implement the policies via low level (software and hardware) functions
Separation between policies and mechanisms

- Discuss access requirements *independently* from their implementation
- **Compare** different access control policies as well as different mechanisms that enforce the same policy
- Design mechanisms able to **enforce multiple policies**
Access Control

Security policies

- **Access control** policies: define who can access a resource
  - Discretionary (DAC)
  - Mandatory (MAC)
  - Role-based (RBAC)
  - Attribute-based (ABAC)

- **Administrative** policies: define who can specify access control policies
  - Usually coupled with DAC, RBAC and ABAC
Access control systems should answer questions like:

1. Can Alice read file “/users/Bob/readme.txt”?
2. Can Bob open a SSH connection with “myserver.com”?

- If yes, access is granted,
- If not, access is denied.

Looks like a simple enough mechanism to implement?
Key challenges for an access control system

- **Expressiveness**: Does the access control model allow to completely express high level access requirements in terms of access control policies?

- **Efficiency**: Access requests need to be dealt with quickly.

- **Full Mediation**: How do you know you have not forgotten some checks?

- **Safety**: How do you know your access control policy captures your access requirements?
Outline

Access Control

Discretionary Access Control

Safety Problem

DAC weaknesses
Discretionary Access Control (DAC)

- **Intuitions:**
  - The owner of a resource decides who can access it
  - Access rights can be delegated to other users

- **Intended Environment:**
  - Operating systems in the late 60s
  - Users are members of the same community: objective is to protect data from mistakes of others (cd /; rm -fr *)
  - Still used in many corporate environments
The basis for DAC is that an individual user (or program on its behalf) is allowed to specify explicitly the types of access other users (or programs on their behalf) may have to information under the user’s control.

NCSC (National Computer Security Center)
Discretionary Access Control (DAC)

- **Entities:**
  - **Subjects:** who have privileges and can do operations on objects
  - **Objects:** files, resources, programs
  - **Access Rights:** what a subject can do on an object

- **Main Features**
  - **Ownership:** Users have full control over the objects they create
    - When users create objects, they own them
  - **Delegation:** Users can give certain rights to other users
    - Users can grant rights they have, to others
    - Users that have control can remove rights
Discretionary Access Control

DAC Models

- Graham-Denning (1972)
- Lampson (1974)
- Harrison-Ruzzo-Ullman (1976)
- Griffiths-Wade (1976)
- Take-grant protection model (1977)
- Originator control (1989)
Lampson: Access Matrix

- Set of subjects $S$
- Set of objects $O$
  - Subjects can be considered as objects, i.e. $S \subseteq O$
- Access Matrix $A (S \times O)$
  - Entries contain sets of rights
    - access rights: read, write, call, etc.
    - administration rights: own, control
    - delegation rights: flags “*” and “+”
### Sample Access Matrix

<table>
<thead>
<tr>
<th>Subject 1</th>
<th>Subject 2</th>
<th>Subject 3</th>
<th>File 1</th>
<th>File 2</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>own control</td>
<td>own control</td>
<td>*call</td>
<td>own</td>
<td>*read *write</td>
<td></td>
</tr>
<tr>
<td>Subject 2</td>
<td></td>
<td>call</td>
<td></td>
<td>write</td>
<td>wakeup</td>
</tr>
<tr>
<td>Subject 3</td>
<td></td>
<td>own control</td>
<td>read</td>
<td>own</td>
<td></td>
</tr>
</tbody>
</table>

- “own” can *add* any rights
- “control” is the right to *remove* rights from a subject (in Lampson’s work; in other models ownership implies control)
- “*” flag can *delegate* rights to other subjects
### Question

What they are allowed to do?

<table>
<thead>
<tr>
<th>Subject 1</th>
<th>Subject 2</th>
<th>Subject 3</th>
<th>File 1</th>
<th>File 2</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject 1</td>
<td>own control</td>
<td>own control</td>
<td>*call</td>
<td>own *read *write</td>
<td></td>
</tr>
<tr>
<td>Subject 2</td>
<td></td>
<td>call</td>
<td></td>
<td>write</td>
<td>wakeup</td>
</tr>
<tr>
<td>Subject 3</td>
<td></td>
<td>own control</td>
<td>read</td>
<td></td>
<td>own</td>
</tr>
</tbody>
</table>
Discretionary Access Control

Lampson

Answer

<table>
<thead>
<tr>
<th>Subject 1</th>
<th>Subject 2</th>
<th>Subject 3</th>
<th>File 1</th>
<th>File 2</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject 1</td>
<td>own control</td>
<td>own control</td>
<td>*call</td>
<td>own</td>
<td>own</td>
</tr>
<tr>
<td>Subject 2</td>
<td>call</td>
<td>own control</td>
<td>*read</td>
<td>write</td>
<td>wakeup</td>
</tr>
<tr>
<td>Subject 3</td>
<td>own control</td>
<td>read</td>
<td>own</td>
<td>own</td>
<td></td>
</tr>
</tbody>
</table>

- **control**: S1 can remove any right of S2
- ***read**: S1 can give any subject the right to read F1
- **own**: S3 can give any right on F2 to every subject
Matrix is generally large and sparse

- Storing the entire matrix is waste of memory space

Alternative approaches

- **Authorization table**: store table of non-null triples \((s, o, a)\)
  - used in DBMS
- **Access control lists (ACLs)**: Stored by column
- **Capability lists**: Stored by row
### Authorization Tables

<table>
<thead>
<tr>
<th>Subject</th>
<th>Access mode</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject 1</td>
<td>own</td>
<td>Subject 1</td>
</tr>
<tr>
<td>Subject 1</td>
<td>control</td>
<td>Subject 1</td>
</tr>
<tr>
<td>Subject 1</td>
<td>own</td>
<td>Subject 2</td>
</tr>
<tr>
<td>Subject 1</td>
<td>control</td>
<td>Subject 2</td>
</tr>
<tr>
<td>Subject 1</td>
<td>call</td>
<td>Subject 3</td>
</tr>
<tr>
<td>Subject 1</td>
<td>own</td>
<td>File 1</td>
</tr>
<tr>
<td>Subject 1</td>
<td>read</td>
<td>File 1</td>
</tr>
<tr>
<td>Subject 1</td>
<td>write</td>
<td>File 1</td>
</tr>
<tr>
<td>Subject 2</td>
<td>call</td>
<td>Subject 3</td>
</tr>
<tr>
<td>Subject 2</td>
<td>read</td>
<td>File 1</td>
</tr>
<tr>
<td>Subject 2</td>
<td>write</td>
<td>File 3</td>
</tr>
<tr>
<td>Subject 2</td>
<td>wakeup</td>
<td>Process 1</td>
</tr>
</tbody>
</table>
Access control lists vs. Capability lists
Access control lists vs. Capability lists

Access control lists

- Objects are the starting point (per-object basis)
- Require authentication of subjects
- Superior for access control and revocation on per-object basis

Capability lists

- Subject are the starting point (per-subject basis)
- Require control of propagation of capabilities
- Superior for access control and revocation on per-subject basis

The per-object basis usually more convenient
Most systems based on ACLs
Define authorization system

- **State** \((S, O, A)\)
  - \(S\) set of subjects
  - \(O\) set of objects
    - Subject can be considered as objects (i.e., \(S \subseteq O\))
  - \(A\) access matrix \((S \times O)\)
    - rows correspond to subjects
    - columns correspond to objects
    - \(A[s, o]\) indicates the privilege of \(s\) on \(o\)

- **State transitions** described by commands
  - commands defined as sequences of primitive operations
  - **enter** \(r\) into \(A[s, o]\), **delete** \(r\) from \(A[s, o]\), create subject \(s'\), destroy subject \(s'\), create object \(o'\), destroy object \(o'\)
### Primitive operations (1)

Current state: $(S, O, A)$

<table>
<thead>
<tr>
<th>OPERATION $(op)$</th>
<th>CONDITIONS</th>
<th>NEW STATE $(Q' = (S', O', A'))$</th>
</tr>
</thead>
</table>
| **create subject $s'$** | $s' \notin O$ | $S' = S \cup \{s'\}$  
$O' = O \cup \{s'\}$  
$A'[s, o] = A[s, o]$ $\forall s \in S, o \in O$  
$A'[s', o] = \emptyset$ $\forall o \in O'$  
$A'[s, s'] = \emptyset$ $\forall s \in S'$ |
| **create object $o'$** | $o' \notin O$ | $S' = S$  
$O' = O \cup \{o'\}$  
$A'[s, o] = A[s, o]$ $\forall s \in S, o \in O$  
$A'[s, o'] = \emptyset$ $\forall s \in S'$ |
| **enter $r$ into $A[s, o]$** | $s \in S$  
$o \in O$ | $S' = S$  
$O' = O$  
$A'[s, o] = A[s, o] \cup \{r\}$  
$A'[s, o] = A[s, o]$ $\forall (s_i, o_i) \neq (s, o)$ |
### Primitive operations (2)

**Current state:** \((S, O, A)\)

<table>
<thead>
<tr>
<th>OPERATION ((op))</th>
<th>CONDITIONS</th>
<th>NEW STATE ((Q' = (S', O', A')))</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>delete</strong> (r) from (A[s, o])</td>
<td>(s \in S)  (o \in O)</td>
<td>(S' = S)  (O' = O)  (A'[s, o] = A[s, o]\setminus{r})  (A'[s_i, o_j] = A[s_i, o_j] \ \forall (s_i, o_i) \neq (s, o))</td>
</tr>
<tr>
<td><strong>destroy subject</strong> (s')</td>
<td>(s' \in S)</td>
<td>(S' = S\setminus{s'})  (O' = O\setminus{s'})  (A'[s, o] = A[s, o] \ \forall s \in S', o \in O')</td>
</tr>
<tr>
<td><strong>destroy object</strong> (o')</td>
<td>(o' \in O)  (o' \notin S)</td>
<td>(S' = S)  (O' = O\setminus{o'})  (A'[s, o] = A[s, o] \ \forall s \in S', o \in O')</td>
</tr>
</tbody>
</table>
**Commands**

Changes to the system state modeled by commands of the form

\[ \text{command } c(x_1, \ldots, x_k) \]
\[ \text{if } r_1 \text{ in } A[x_{s_1}, x_{o_1}] \text{ and} \]
\[ r_2 \text{ in } A[x_{s_2}, x_{o_2}] \text{ and} \]
\[ \ldots \]
\[ r_m \text{ in } A[x_{s_m}, x_{o_m}] \]
\[ \text{then } op_1 \]
\[ op_2 \]
\[ \ldots \]
\[ op_n \]
\[ \text{end} \]

with \( r_1, \ldots, r_m \) rights and \( op_1, \ldots, op_n \) primitive operations
Commands – Examples

**command** `CREATE(s, o)`
- create object `o`
- enter own into $A[s, o]$ end.

**command** `CONFERENCE_{read}(s_1, s_2, o)`
- if own in $A[s_1, o]$
- then enter read into $A[s_2, o]$ end.

**command** `REVOKER_{read}(s_1, s_2, o)`
- if own in $A[s_1, o]$
- then delete read into $A[s_2, o]$ end.
Transfer of privileges

Delegation of authority by attaching flags to privileges (e.g., * copy flag, + transfer-only flag)

▶ copy flag (*): subject can transfer privilege to others

command \(\text{TRANSFER}_{\text{read}}(s_1, s_2, o)\)
  
  if *read in \(A[s_1, o]\)  
  then enter read into \(A[s_2, o]\) end.

▶ transfer-only flag (+): subject can transfer privilege to others (and the flag on it), but he loses the privilege

command \(\text{TRANSFER-ONLY}_{\text{read}}(s_1, s_2, o)\)
  
  if +read in \(A[s_1, o]\)  
  then delete +read into \(A[s_1, o]\)  
  enter +read into \(A[s_2, o]\) end.
Write a command which allows a process $p$ to create a new process $q$ where parent and child processes can signal (read/write) each other.
Solution

```plaintext
command spawn_process(p, q)
create subject q
enter own into A[p, q]
enter r into A[p, q]
enter w into A[p, q]
enter r into A[q, p]
enter w into A[q, p]
end
```
A state \( Q = (S, O, A) \) yields a state \( Q' = (S', O', A') \) under

**command** \( c(x_1, \ldots, x_k) \)

**if** \( r_1 \) in \( A[x_{s_1}, x_{o_1}] \) and \( \ldots \) \( r_m \) in \( A[x_{s_m}, x_{o_m}] \)

**then** \( o_{p1}, \ldots, o_{pn} \)

**end**

with arguments \( a_1, \ldots, a_k \), denoted \( Q \Rightarrow c(a_1, \ldots, a_k) \) \( Q' \), provided

- \( Q' = Q \) if one of the conditions of \( c \) is not satisfied
- \( Q' = Q_n \) otherwise, where there exist states \( Q_0, Q_1, \ldots, Q_n \) such that \( Q = Q_0 \) and \( Q_n = Q' \) and for each \( i \), with \( 0 \leq i \leq n \),

\[
Q_i \Rightarrow o_{p_{i+1}}[a_j/x_j] \ Q_{i+1}
\]

where \( o_{p_{i+1}}[a_j/x_j] \) denotes the primitive operation \( o_{p_{i+1}} \), substituting \( a_1, \ldots, a_k \) for variables \( x_1, \ldots, x_k \)
Compute the access matrix that results from the following initial state by executing the sequence of commands $\alpha$ defined as follows:

1. $CREATE(Alice, File3)$
2. $CONFER_{+\text{read}}(Alice, Bob, File3)$
3. $CREATE(Charlie, File3)$
4. $CONFER_{+\text{read}}(Charlie, David, File3)$
5. $TRANSFER-ONLY_{\text{read}}(Bob, David, File3)$
6. $TRANSFER_{\text{read}}(Charlie, Alice, File2)$
Solution

Compute the access matrix that results from the following initial state by executing the sequence of commands $\alpha$ defined as follows:

1. $CREATE(Alice, File3)$
2. $CONFER_{+read}(Alice, Bob, File3)$
3. $CREATE(Charlie, File3)$
4. $CONFER_{+read}(Charlie, David, File3)$
5. $TRANSFER-ONLY_{read}(Bob, David, File3)$
6. $TRANSFER_{read}(Charlie, Alice, File2)$

<table>
<thead>
<tr>
<th></th>
<th>File 1</th>
<th>File 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td>own</td>
<td></td>
</tr>
<tr>
<td>Charlie</td>
<td></td>
<td>own</td>
</tr>
<tr>
<td>David</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Compute the access matrix that results from the following initial state by executing the sequence of commands $\alpha$ defined as follows:

1. $CREATE(Alice, File3)$
2. $CONFERENCE_read(Alice, Bob, File3)$
3. $CREATE(Charlie, File3)$
4. $CONFERENCE_read(Charlie, David, File3)$
5. $TRANSFER-ONLY_read(Bob, David, File3)$
6. $TRANSFER_read(Charlie, Alice, File2)$

<table>
<thead>
<tr>
<th></th>
<th>File 1</th>
<th>File 2</th>
<th>File 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td></td>
<td></td>
<td>own</td>
</tr>
<tr>
<td>Bob</td>
<td></td>
<td>own</td>
<td></td>
</tr>
<tr>
<td>Charlie</td>
<td>own</td>
<td></td>
<td></td>
</tr>
<tr>
<td>David</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Solution

Compute the access matrix that results from the following initial state by executing the sequence of commands $\alpha$ defined as follows:

1. $CREATE(Alice, File3)$
2. $CONFER_{+\text{read}}(Alice, Bob, File3)$
3. $CREATE(Charlie, File3)$
4. $CONFER_{+\text{read}}(Charlie, David, File3)$
5. $TRANSFER-ONLY_{\text{read}}(Bob, David, File3)$
6. $TRANSFER_{\text{read}}(Charlie, Alice, File2)$

<table>
<thead>
<tr>
<th></th>
<th>File 1</th>
<th>File 2</th>
<th>File 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td></td>
<td></td>
<td>own</td>
</tr>
<tr>
<td>Bob</td>
<td>own</td>
<td></td>
<td>+read</td>
</tr>
<tr>
<td>Charlie</td>
<td></td>
<td>own</td>
<td></td>
</tr>
<tr>
<td>David</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Compute the access matrix that results from the following initial state by executing the sequence of commands $\alpha$ defined as follows:

1. $\text{CREATE}(\text{Alice}, \text{File3})$
2. $\text{CONFER}^{+\text{read}}(\text{Alice}, \text{Bob}, \text{File3})$
3. $\text{CREATE}(\text{Charlie}, \text{File3})$
4. $\text{CONFER}^{+\text{read}}(\text{Charlie}, \text{David}, \text{File3})$
5. $\text{TRANSFER-ONLY}_{\text{read}}(\text{Bob}, \text{David}, \text{File3})$
6. $\text{TRANSFER}_{\text{read}}(\text{Charlie}, \text{Alice}, \text{File2})$

<table>
<thead>
<tr>
<th></th>
<th>File 1</th>
<th>File 2</th>
<th>File 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td></td>
<td>own</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td>own</td>
<td></td>
<td>+read</td>
</tr>
<tr>
<td>Charlie</td>
<td>own</td>
<td></td>
<td></td>
</tr>
<tr>
<td>David</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Solution

Compute the access matrix that results from the following initial state by executing the sequence of commands $\alpha$ defined as follows:

1. $\text{CREATE}(\text{Alice}, \text{File}3)$
2. $\text{CONFER}_{\text{+read}}(\text{Alice}, \text{Bob}, \text{File}3)$
3. $\text{CREATE}(\text{Charlie}, \text{File}3)$
4. $\text{CONFER}_{\text{+read}}(\text{Charlie}, \text{David}, \text{File}3)$
5. $\text{TRANSFER-ONLY}_{\text{read}}(\text{Bob}, \text{David}, \text{File}3)$
6. $\text{TRANSFER}_{\text{read}}(\text{Charlie}, \text{Alice}, \text{File}2)$

<table>
<thead>
<tr>
<th></th>
<th>File 1</th>
<th>File 2</th>
<th>File 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td></td>
<td></td>
<td>own</td>
</tr>
<tr>
<td>Bob</td>
<td>own</td>
<td></td>
<td>+read</td>
</tr>
<tr>
<td>Charlie</td>
<td>own</td>
<td></td>
<td></td>
</tr>
<tr>
<td>David</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Discretionary Access Control
Harrison-Ruzzo-Ullman (HRU) Model

Solution

Compute the access matrix that results from the following initial state by executing the sequence of commands $\alpha$ defined as follows:

1. $\text{CREATE}(Alice, \text{File}3)$
2. $\text{CONFER}_{\text{read}}^+(Alice, Bob, \text{File}3)$
3. $\text{CREATE}(Charlie, \text{File}3)$
4. $\text{CONFER}_{\text{read}}^+(Charlie, David, \text{File}3)$
5. $\text{TRANSFER-ONLY}_{\text{read}}(Bob, David, \text{File}3)$
6. $\text{TRANSFER}_{\text{read}}(Charlie, Alice, \text{File}2)$

<table>
<thead>
<tr>
<th></th>
<th>File 1</th>
<th>File 2</th>
<th>File 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td></td>
<td></td>
<td>own</td>
</tr>
<tr>
<td>Bob</td>
<td>own</td>
<td></td>
<td>+read</td>
</tr>
<tr>
<td>Charlie</td>
<td>own</td>
<td></td>
<td></td>
</tr>
<tr>
<td>David</td>
<td></td>
<td></td>
<td>+read</td>
</tr>
</tbody>
</table>
Compute the access matrix that results from the following initial state by executing the sequence of commands $\alpha$ defined as follows:

1. $\text{CREATE}(Alice, \text{File3})$
2. $\text{CONFER}_{\text{read}}(Alice, Bob, \text{File3})$
3. $\text{CREATE}(Charlie, \text{File3})$
4. $\text{CONFER}_{\text{read}}(Charlie, David, \text{File3})$
5. $\text{TRANSFER-ONLY}_{\text{read}}(Bob, David, \text{File3})$
6. $\text{TRANSFER}_{\text{read}}(Charlie, Alice, \text{File2})$

Note: own does not assume all rights.
### Solution

Compute the access matrix that results from the following initial state by executing the sequence of commands $\alpha$ defined as follows:

1. $\text{CREATE}(\text{Alice, File3})$
2. $\text{CONFER}_{\text{read}}(\text{Alice, Bob, File3})$
3. $\text{CREATE}(\text{Charlie, File3})$
4. $\text{CONFER}_{\text{read}}(\text{Charlie, David, File3})$
5. $\text{TRANSFER-ONLY}_{\text{read}}(\text{Bob, David, File3})$
6. $\text{TRANSFER}_{\text{read}}(\text{Charlie, Alice, File2})$

<table>
<thead>
<tr>
<th></th>
<th>File 1</th>
<th>File 2</th>
<th>File 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td></td>
<td></td>
<td>own</td>
</tr>
<tr>
<td>Bob</td>
<td>own</td>
<td></td>
<td>+read</td>
</tr>
<tr>
<td>Charlie</td>
<td>own</td>
<td></td>
<td></td>
</tr>
<tr>
<td>David</td>
<td></td>
<td></td>
<td>+read</td>
</tr>
</tbody>
</table>

**Note:** own does not assume all rights.
Outline

Access Control

Discretionary Access Control

Safety Problem

DAC weaknesses
Fundamental Questions

- How can we determine whether a system is secure?
  - Need to define what we mean by a system being “secure”

- Is there a generic algorithm that allows us to determine whether a computer system is secure?
Safety and Liveness

- **Safety property**: nothing bad happens
  - the program will never produce a wrong result ("partial correctness")

- **Liveness property**: something good eventually happens
  - the program will produce a result ("termination")
Example: Safety and Liveness

C1: Whenever process P1 wants to enter the critical session, provided process P2 never stays in the critical session forever, P1 gets to enter eventually.

C2: It cannot happen that both processes are in their critical sessions simultaneously.
Example: Safety and Liveness

C1: Whenever process P1 wants to enter the critical session, provided process P2 never stays in the critical session forever, P1 gets to enter eventually.

Liveness

C2: It cannot happen that both processes are in their critical sessions simultaneously.

Safety
When is a system secure?

A simple definition

A system is secure if it does not allow violations of security policies.

Alternative definition based on distribution of rights

No leakage of rights.
When is a system secure?

- A simple definition
  - A system is secure if it does not allow violations of security policies

- Alternative definition based on distribution of rights
  - No leakage of rights
What is a secure system?

**Definition**

A state **leaks** a right $r$ if there is a command $c$ that enters $r$ into a position of the access matrix that previously did not contain $r$.

Intuitively, if there is $c$ such that $(S, O, A) \Rightarrow_c (S', O', A')$ where $r \notin A[s, o]$ and $r \in A'[s, o]$ for some $s$ and $o$.

**Definition**

A state is **secure** with respect to a right $r$ if no sequence of commands can transform the access matrix into a state that leaks $r$. 
Exercise

Assume a system with the following two commands:

```plaintext
command grant_execute (s₁, s₂, o)
   if own in A[s₁, o]
   then enter exec into A[s₂, o]
end

command modify_own_right (s, o)
   if exec in A[s, o]
   then enter write into A[s, o]
end
```

**Policy**: Suppose Bob has developed an application program; he wants this program to be run by other users but not modified by them.

Is the system secure? Why?
The system is NOT secure with respect to the given policy

1. Bob: grant_execute \((Bob, Tom, p_1)\)
2. Tom: modify\_own\_right \((Tom, p_1)\)

\(A[Tom, p_1]\) contains the write access right
HRU: Safety Problem

Safety problem: Given an initial state \((S, O, A)\) and a right \(r\), is there any sequence of commands leaking \(r\)?
HRU and Safety: “Desired” leaks

Leaks are not necessarily bad
- A system that allows sharing will of course have many leaks
- Subjects can intentionally transfer (leak) their rights to other trustworthy subjects

Leaks of rights to trustworthy subjects should be ignored when assessing the security of the system (i.e., ignoring “desired” leaks)
Exercise

Given initial state $Q = (S, O, A)$ with $S = \{s_1, s_2, s_3, s_4, s_5\}$, $O = \{s_1, s_2, s_3, s_4, s_5\}$ and $A = \emptyset$, and the sequence of commands $\alpha$

- CREATE$(s_3, o)$
- CONFER$_{read}$(s$_3$, s$_1$, o)$
- CONFER$_{*read}$(s$_3$, s$_4$, o)$
- TRANSFER$_{*read}$(s$_2$, s$_5$, o)$
- TRANSFER$_{read}$(s$_1$, s$_5$, o)$
- REVOKE$_{read}$(s$_1$, s$_4$, o)$
- TRANSFER$_{read}$(s$_4$, s$_5$, o)$
- REVOKE$_{read}$(s$_3$, s$_5$, o)$

Determine the state $Q'$ after the execution of $\alpha$.

Does $\alpha$ leak access privileges? (Consider only $s_5$ to be untrusted)

Justify the answer.
The solution to the safety problem involves the following actions:

- \( \text{CREATE}(s_3, o) \)
- \( \text{CONFER}_{\text{read}}(s_3, s_1, o) \)
- \( \text{CONFER}_{\ast, \text{read}}(s_3, s_4, o) \)
- \( \text{TRANSFER}_{\text{read}}(s_2, s_5, o) \)
- \( \text{TRANSFER}_{\text{read}}(s_1, s_5, o) \)
- \( \text{REVOKE}_{\text{read}}(s_1, s_4, o) \)
- \( \text{TRANSFER}_{\text{read}}(s_4, s_5, o) \)
- \( \text{REVOKE}_{\text{read}}(s_3, s_5, o) \)

The table below shows the access rights:

<table>
<thead>
<tr>
<th></th>
<th>S_1</th>
<th>S_2</th>
<th>S_3</th>
<th>S_4</th>
<th>S_5</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>own</td>
</tr>
<tr>
<td>S_4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S_5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Solution**

CREATE\((s_3, o)\)

CONFER\_{read}(s_3, s_1, o)

CONFER\_{*read}(s_3, s_4, o)

TRANSFER\_{read}(s_2, s_5, o)

TRANSFER\_{read}(s_1, s_5, o)

REVOKE\_{read}(s_1, s_4, o)

TRANSFER\_{read}(s_4, s_5, o)

REVOKE\_{read}(s_3, s_5, o)

\[
\begin{array}{cccccc}
 & S_1 & S_2 & S_3 & S_4 & S_5 & O \\
S_1 &   &   &   &   &   & \text{read} \\
S_2 &   &   &   &   &   &   \\
S_3 &   &   &   &   &   & \text{own} \\
S_4 &   &   &   &   &   &   \\
S_5 &   &   &   &   &   &   \\
\end{array}
\]

Does \(\alpha\) leak access privileges? (Consider only \(s_5\) to be untrusted)

Yes, \(\alpha\) leaks access privileges. At a certain point, \(s_5\) had read privilege.
\text{CREATE}(s_3, o) \\
\text{CONFER}_{\text{read}}(s_3, s_1, o) \\
\text{CONFER}_{\ast \text{read}}(s_3, s_4, o) \\
\text{TRANSFER}_{\text{read}}(s_2, s_5, o) \\
\text{TRANSFER}_{\text{read}}(s_1, s_5, o) \\
\text{REVOKE}_{\text{read}}(s_1, s_4, o) \\
\text{TRANSFER}_{\text{read}}(s_4, s_5, o) \\
\text{REVOKE}_{\text{read}}(s_3, s_5, o)

\begin{array}{|c|c|c|c|c|c|}
\hline
& S_1 & S_2 & S_3 & S_4 & S_5 & O \\
\hline
S_1 & & & & & & \text{read} \\
\hline
S_2 & & & & & & \text{own} \\
\hline
S_3 & & & & & & \ast \text{read} \\
\hline
S_4 & & & & & & \ast \text{read} \\
\hline
S_5 & & & & & & \text{read} \\
\hline
\end{array}

\text{Does } \alpha \text{ leak access privileges? (Consider only } s_5 \text{ to be untrusted)} \\
\text{Yes, } \alpha \text{ leaks access privileges. At a certain point, } s_5 \text{ had read privilege.}
### Solution

```plaintext
CREATE(s_3, o)  
CONFER_{read}(s_3, s_1, o)  
CONFER_{*read}(s_3, s_4, o)  
TRANSFER_{read}(s_2, s_5, o)  
TRANSFER_{read}(s_1, s_5, o)  
REVOKE_{read}(s_1, s_4, o)  
TRANSFER_{read}(s_4, s_5, o)  
REVOKE_{read}(s_3, s_5, o)
```

<table>
<thead>
<tr>
<th></th>
<th>S_1</th>
<th>S_2</th>
<th>S_3</th>
<th>S_4</th>
<th>S_5</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>read</td>
</tr>
<tr>
<td>S_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>own</td>
</tr>
<tr>
<td>S_4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*read</td>
</tr>
<tr>
<td>S_5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Does α leak access privileges? (Consider only s_5 to be untrusted)

Yes, α leaks access privileges. At a certain point, s_5 had read privilege.
Solution

\[\text{CREATE}(s_3, o)\]
\[\text{CONFER}_{\text{read}}(s_3, s_1, o)\]
\[\text{CONFER}_{\text{*read}}(s_3, s_4, o)\]
\[\text{TRANSFER}_{\text{read}}(s_2, s_5, o)\]
\[\text{TRANSFER}_{\text{read}}(s_1, s_5, o)\]
\[\text{REVOKE}_{\text{read}}(s_1, s_4, o)\]
\[\text{TRANSFER}_{\text{read}}(s_4, s_5, o)\]
\[\text{REVOKE}_{\text{read}}(s_3, s_5, o)\]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
 & s_1 & s_2 & s_3 & s_4 & s_5 & o \\
\hline
s_1 &  &  &  &  &  & \text{read} \\
\hline
s_2 &  &  &  &  &  & \text{own} \\
\hline
s_3 &  &  &  &  &  & \text{*read} \\
\hline
s_4 &  &  &  &  &  & \text{own} \\
\hline
s_5 &  &  &  &  &  & \text{read} \\
\hline
\end{array}
\]

\text{Does} \alpha \text{ leak access privileges? (Consider only } s_5 \text{ to be untrusted)}
\text{Yes, } \alpha \text{ leaks access privileges. At a certain point, } s_5 \text{ had read privilege.}
Safety Problem

Solution

CREATE($s_3, o$)
CONFER_{read}($s_3, s_1, o$)
CONFER_{*read}($s_3, s_4, o$)
TRANSFER_{read}($s_2, s_5, o$)
TRANSFER_{read}($s_1, s_5, o$)
REVOKE_{read}($s_1, s_4, o$)
TRANSFER_{read}($s_4, s_5, o$)
REVOKE_{read}($s_3, s_5, o$)

<table>
<thead>
<tr>
<th></th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
<th>$s_5$</th>
<th>$o$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>read</td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>own</td>
<td></td>
</tr>
<tr>
<td>$s_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*read</td>
</tr>
<tr>
<td>$s_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Does $\alpha$ leak access privileges? (Consider only $s_5$ to be untrusted)

Yes, $\alpha$ leaks access privileges. At a certain point, $s_5$ had read privilege.
Safety Problem

Solution

```
CREATE(s_3, o)
CONFER_{read}(s_3, s_1, o)
CONFER_{*read}(s_3, s_4, o)
TRANSFER_{read}(s_2, s_5, o)
TRANSFER_{read}(s_1, s_5, o)
REVOKE_{read}(s_1, s_4, o)
TRANSFER_{read}(s_4, s_5, o)
REVOKE_{read}(s_3, s_5, o)
```

<table>
<thead>
<tr>
<th></th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
<th>$S_5$</th>
<th>$O$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>read</td>
<td></td>
</tr>
<tr>
<td>$S_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>own</td>
<td></td>
</tr>
<tr>
<td>$S_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*read</td>
<td></td>
</tr>
<tr>
<td>$S_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>read</td>
<td></td>
</tr>
</tbody>
</table>

Does α leak access privileges? (Consider only $s_5$ to be untrusted)
Yes, α leaks access privileges. At a certain point, $s_5$ had read privilege.
Solution

CREATE\((s_3, o)\)
CONFER_{read}(s_3, s_1, o)
CONFER_{*read}(s_3, s_4, o)
TRANSFER_{read}(s_2, s_5, o)
TRANSFER_{read}(s_1, s_5, o)
REVOKE_{read}(s_1, s_4, o)
TRANSFER_{read}(s_4, s_5, o)
REVOKE_{read}(s_3, s_5, o)

<table>
<thead>
<tr>
<th></th>
<th>(S_1)</th>
<th>(S_2)</th>
<th>(S_3)</th>
<th>(S_4)</th>
<th>(S_5)</th>
<th>(O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S_1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>read</td>
</tr>
<tr>
<td>(S_2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(S_3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>own</td>
</tr>
<tr>
<td>(S_4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*read</td>
</tr>
<tr>
<td>(S_5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Does \(\alpha\) leak access privileges? (Consider only \(S_5\) to be untrusted)

Yes, \(\alpha\) leaks access privileges. At a certain point, \(S_5\) had read privilege.
Solution

\[ CREATE(s_3, o) \]
\[ CONFER_{read}(s_3, s_1, o) \]
\[ CONFER_{*read}(s_3, s_4, o) \]
\[ TRANSFER_{read}(s_2, s_5, o) \]
\[ TRANSFER_{read}(s_1, s_5, o) \]
\[ REVOKE_{read}(s_1, s_4, o) \]
\[ TRANSFER_{read}(s_4, s_5, o) \]
\[ REVOKE_{read}(s_3, s_5, o) \]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
 & s_1 & s_2 & s_3 & s_4 & s_5 & o \\
\hline
s_1 & & & & & & read \\
\hline
s_2 & & & & & & \\
\hline
s_3 & & & & & & own \\
\hline
s_4 & & & & & & *read \\
\hline
s_5 & & & & & & \\
\hline
\end{array}
\]

Does \( \alpha \) leak access privileges? (Consider only \( s_5 \) to be untrusted)
CREATE\((s_3, o)\)
CONFER_{\text{read}}(s_3, s_1, o)
CONFER_{\ast\text{read}}(s_3, s_4, o)
TRANSFER_{\text{read}}(s_2, s_5, o)
TRANSFER_{\text{read}}(s_1, s_5, o)
REVOKE_{\text{read}}(s_1, s_4, o)
TRANSFER_{\text{read}}(s_4, s_5, o)
REVOKE_{\text{read}}(s_3, s_5, o)

<table>
<thead>
<tr>
<th></th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
<th>$s_5$</th>
<th>$o$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>read</td>
</tr>
<tr>
<td>$s_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>own</td>
</tr>
<tr>
<td>$s_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*read</td>
</tr>
<tr>
<td>$s_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>read</td>
</tr>
</tbody>
</table>

Does $\alpha$ leak access privileges? (Consider only $s_5$ to be untrusted)

Yes, $\alpha$ leaks access privileges. At a certain point, $s_5$ had read privilege.
Safety problem: Decidability

Safety problem: Given an initial state \((S, O, A)\) and a right \(r\), is there any sequence of commands leaking \(r\)?

Or does transfer of \(r\) violate security policies of the system?

Is there a generic algorithm that allows us to determine whether a computer system is secure?
Safety Problem

Safety problem: Some bad news

Theorem

*Verifying the security of an access matrix $A$ with respect to a right $r$ is undecidable.*

The safety problem can be reduced to the halting problem of a Turing machine
A mono-operational system is a system in which every command consists of a single operation

**Theorem**

Verifying the security of an access matrix $A$ with respect to a right $r$ is decidable in a mono-operational system.

**Bad news**

- Mono-operational systems are not useful
- If you create an object you cannot own it $\implies$ you cannot modify it

**Remark**: Decidability can also be proved for systems with a finite number of subjects and objects, or for systems where it is not possible to create new subjects and objects.
The more expressive the security model is, the more difficult it is to verify security.

- Is it better to have a very expressive policy language or a simple language where it is easier to check the impact of policy decision?
- In an expressive language, it is easier to capture the intended policy but more difficult to verify that the policy actually does what is supposed to do.
- In a simple language, it may not be possible to capture access requirements precisely, but there are effective ways to check what the policy does.
Outline

Access Control

Discretionary Access Control

Safety Problem

DAC weaknesses
DAC weaknesses

- DAC imposes constraints only on direct access
- No control on the information once released
  \[ \Rightarrow \text{DAC is vulnerable to Trojan horses} \]
  - exploiting access privileges of calling subject

**Trojan horses**: a program (apparently harmless) which contains malicious or harmful code
Trojan horses – Example

contacts

<table>
<thead>
<tr>
<th>Alice</th>
<th>06-12345678</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>06-23456781</td>
</tr>
</tbody>
</table>

(Alice, owner, contacts)
(Bob, read, contacts)
Trojan horses – Example

Bob invokes Application

contacts

Alice  06-12345678
---
Bob    06-23456781
---

(Alice, owner, contacts)
(Bob, read, contacts)
Trojan horses – Example

Bob invokes Application

contacts

Alice 06-12345678
Bob 06-23456781

(Alice, owner, contacts)
(Bob, read, contacts)

stolen

(Bob, write, stolen)
(Charlie, read, stolen)
Trojan horses – Example

Bob invokes Application

- read contacts
- write stolen

contacts
- Alice 06-12345678
- Bob 06-23456781

stolen

(Alice, owner, contacts)
(Bob, read, contacts)

(Bob, write, stolen)
(Charlie, read, stolen)
Trojan horses – Example

Bob invokes Application

- read contacts
- write stolen

Malicious code

Contacts

- Alice 06-12345678
- Bob 06-23456781

(Alice, owner, contacts)
(Bob, read, contacts)

Stolen

(Charlie, read, stolen)
Trojan horses – Example

Application

Bob invokes

read contacts
write stolen

contacts

(Alice, owner, contacts)
(Bob, read, contacts)

stolen

(Alice 06-12345678)
(Bob 06-23456781)

malicious code

Alice 06-12345678
Bob 06-23456781
Access control: regulates the operations that can be executed on data and resources to be protected

DAC: users can regulate the access to their resources
- Lampson
- HRU model

Safety Problem: Given a configuration $Q$ and a right $r$, is there any sequence of commands that leaks $r$?

DAC vulnerabilities: Trojan horses
References

Homework (Deadline: 10/9/2019)

1. Compute the access matrix that results from the following initial state

<table>
<thead>
<tr>
<th>Alice</th>
<th>File 1</th>
<th>File 2</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td>own</td>
<td>own</td>
<td></td>
</tr>
<tr>
<td>Charlie</td>
<td>own</td>
<td>*read</td>
<td></td>
</tr>
<tr>
<td>David</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

by executing the sequence of commands $\alpha$ defined as follows:

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$CONFER_{\ast \text{read}}(\text{Charlie, Alice, File1})$</td>
</tr>
<tr>
<td>2</td>
<td>$CONFER_{\text{exec}}(\text{Bob, Alice, Process1})$</td>
</tr>
<tr>
<td>3</td>
<td>$CONFER_{\text{write}}(\text{Charlie, Alice, File1})$</td>
</tr>
<tr>
<td>4</td>
<td>$CONFER_{\text{read}}(\text{Bob, Bob, File2})$</td>
</tr>
<tr>
<td>5</td>
<td>$CONFER_{\text{exec}}(\text{Bob, Charlie, Process1})$</td>
</tr>
<tr>
<td>6</td>
<td>$TRANSFER_{\text{exec}}(\text{Alice, Charlie, Process1})$</td>
</tr>
<tr>
<td>7</td>
<td>$CONFER_{\ast \text{write}}(\text{Charlie, Bob, File1})$</td>
</tr>
<tr>
<td>8</td>
<td>$REVOKE_{\text{read}}(\text{Bob, Charlie, File2})$</td>
</tr>
<tr>
<td>9</td>
<td>$REVOKE_{\text{read}}(\text{Alice, Alice, File1})$</td>
</tr>
<tr>
<td>10</td>
<td>$TRANSFER_{\text{read}}(\text{Alice, David, File1})$</td>
</tr>
<tr>
<td>11</td>
<td>$REVOKE_{\text{read}}(\text{Charlie, David, File1})$</td>
</tr>
<tr>
<td>12</td>
<td>$CREATE(\text{Charlie, File3})$</td>
</tr>
<tr>
<td>13</td>
<td>$CONFER_{\ast \text{read}}(\text{Charlie, Bob, File3})$</td>
</tr>
<tr>
<td>14</td>
<td>$TRANSFER_{\text{read}}(\text{Bob, Alice, File3})$</td>
</tr>
<tr>
<td>15</td>
<td>$TRANSFER_{\text{read}}(\text{Charlie, Bob, File2})$</td>
</tr>
<tr>
<td>16</td>
<td>$REVOKE_{\text{read}}(\text{Charlie, Bob, File3})$</td>
</tr>
<tr>
<td>17</td>
<td>$TRANSFER_{\text{read}}(\text{Charlie, David, File2})$</td>
</tr>
<tr>
<td>18</td>
<td>$REVOKE_{\text{read}}(\text{Bob, David, File2})$</td>
</tr>
<tr>
<td>19</td>
<td>$CONFER_{\ast \text{read}}(\text{Bob, Charlie, File2})$</td>
</tr>
<tr>
<td>20</td>
<td>$TRANSFER_{\text{write}}(\text{Charlie, Alice, File1})$</td>
</tr>
</tbody>
</table>

Hints:

- Command $CONFER_{\ast \text{read}}$ is equal to $CONFER_{\text{read}}$ but grants $\ast \text{read}$ instead of $\text{read}$. Similar principle applies to $CONFER_{\ast \text{exec}}$ and $CONFER_{\ast \text{write}}$.
- Command $REVOKE_{\text{read}}$ removes both $\text{read}$ and $\ast \text{read}$. Similar principle applies to $REVOKE_{\text{exec}}$ and $REVOKE_{\text{write}}$.

2. Is $\alpha$ leaking access privileges? (Consider only David to be untrusted) Justify the answer.