isec
INFORMATION SECURITY
lectures by Ferucio Laurentiu Tiplea

Access Control

Discretionary Access Control

Prof.dr. Ferucio Laurențiu Țiplea

Fall 2023

Department of Computer Science
"Alexandru Ioan Cuza" University of Iași
Iași 700506, Romania

e-mail: ferucio.tiplea@uaic.ro

Outline

Introduction to DAC

Discretionary Access Control

Basic features:

- DAC models enforce access control based on the identity of requesters;
- DAC models are called "discretionary" as users can be given the ability of passing on their privileges to other users;
- DAC mechanisms usually include a concept of object ownership;
- DAC models are state-transition systems:
  - A state specifies the rights that subjects have over objects at a given time;
  - A transition shows how the state changes when a command involving subjects, objects, or rights is executed.

The access control matrix model

The access control matrix model

The most general DAC model is the access control matrix (ACM) model, also called the access matrix model or the HRU model, introduced by Harrison et al. (1976).

Basic features:

- It is a state-transition system;
- States are matrices where each row corresponds to a subject, each column corresponds to an object, and a cell specifies the rights a subject has over an object;
- Transitions between states are performed by commands;
- Subjects are objects too.

States

In what follows, R denotes a non-empty finite set of rights.

Definition 1

A state over R is a triple $Q = (S, O, A)$, where S and O are non-empty finite sets of subjects and objects, respectively, and A is an $|S| \times |O|$-matrix whose elements are subsets of R.

Example 2

Let $S = \{process_1, process_2\}$, $O = S \cup \{file\}$, and A given by:

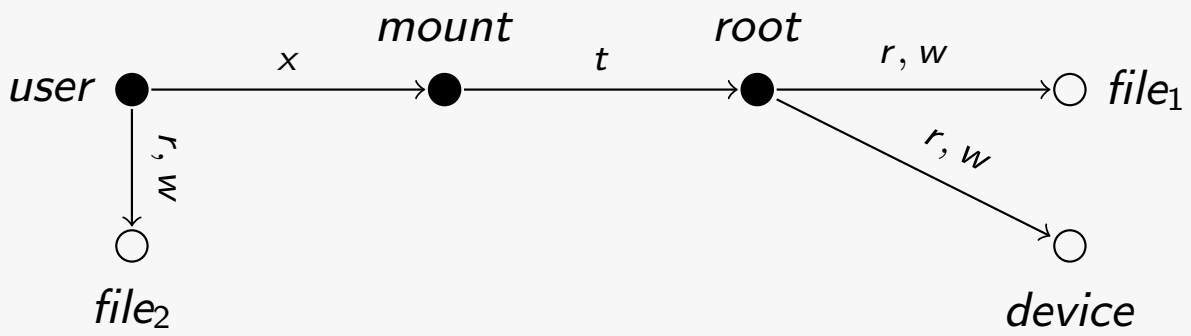
$$\begin{array}{c} \\ process_1 \\ process_2 \end{array} \begin{array}{c} \begin{array}{ccc} process_1 & process_2 & file \end{array} \\ \begin{pmatrix} & r & r, w \\ r, x & & r \end{pmatrix} \end{array}$$

The triple (S, O, A) is a state over $R = \{r, w, x\}$. $A(process_1, file)$ lists the rights $process_1$ has over $file$, namely r and w.

States

For more readability, one may represent states by directed graphs. This representation is specific to the Take-Grant model (Lipton and Snyder (1977)), a sub-model of the ACM model.

Example 3



Dark vertices stand for subjects and open vertices stand for objects. A labeled arc from x to y specifies the rights x has over y.

Primitive operations

Changing rights in the system is based on precisely defined rules, formally specified by commands that are made up of primitive operations.

Let $\mathcal{V}_s$ and $\mathcal{V}_o$ be sets of subject-type and object-type variables, resp.

Definition 4

A primitive operation over R is a construct of the one of the following types ($r \in R$, $X_s \in \mathcal{V}_s$, and $X_o \in \mathcal{V}_o$):

1. *enter* r *into* (X_s, X_o)
2. *delete* r *from* (X_s, X_o)
3. *create subject* X_s
4. *create object* X_o
5. *destroy subject* X_s
6. *destroy object* X_o

Commands

Definition 5

A command over R is a construct of the form:

$$\begin{array}{l} \textit{command } \alpha(X_1, \ldots, X_k) \\ \quad op_1, \ldots, op_n \\ \textit{end} \end{array} \quad \Bigg| \quad \begin{array}{l} \textit{command } \alpha(X_1, \ldots, X_k) \\ \quad \textit{if } \; r_1 \textit{ in } (X_{s_1}, X_{o_1}) \textit{ and} \\ \qquad \ldots \\ \qquad r_m \textit{ in } (X_{s_m}, X_{o_m}) \\ \quad \textit{then } op_1, \ldots, op_n \\ \textit{end} \end{array}$$

where $m, n \geq 1$, $\{X_1, \ldots, X_k\} \subseteq \mathcal{V}_s \cup \mathcal{V}_o$, $X_{s_i} \in \mathcal{V}_s \cap \{X_1, \ldots, X_k\}$, $X_{o_i} \in \mathcal{V}_o \cap \{X_1, \ldots, X_k\}$, $r_i \in R$, and $op_1, \ldots, op_n$ are operations over R with variables in $\{X_1, \ldots, X_k\}$.

Commands do not check the lack of rights but only their existence!

Examples of commands

Example 6 (Harrison et al. (1976))

command $CREATE(process, file)$
    create object $file$
    enter own into $(process, file)$
end

Example 7 (Harrison et al. (1976))

command $CONFER\_READ(owner, friend, file)$
    if own in $(owner, file)$
    then
        enter r *into* $(friend, file)$
end

Examples of commands

Example 8 (Harrison et al. (1976))

command $REMOVE\_READ(owner, exfriend, file)$
    if own *in* $(owner, file)$ *and*
        r *in* $(exfriend, file)$
    then
        delete r *from* $(exfriend, file)$
end

Example 9 (Samarati and de Capitani di Vimercati (2001))

command $TRANSFER\_READ(subj, friend, file)$
    if r *in* $(subj, file)$
    then
        enter r *into* $(friend, file)$
end

Substitution

To execute a command, its formal parameters must be replaced with actual parameters. This is done by substitutions that assign values to variables according to their types:

- subjects to subject-type variables, and
- objects to object-type variables.

Substitutions can homomorphically be applied to primitive operations and commands.

Example 10

If $\sigma(X) = \mathit{Alice} \in S$ and $\sigma(X') = \mathit{file} \in O$, then

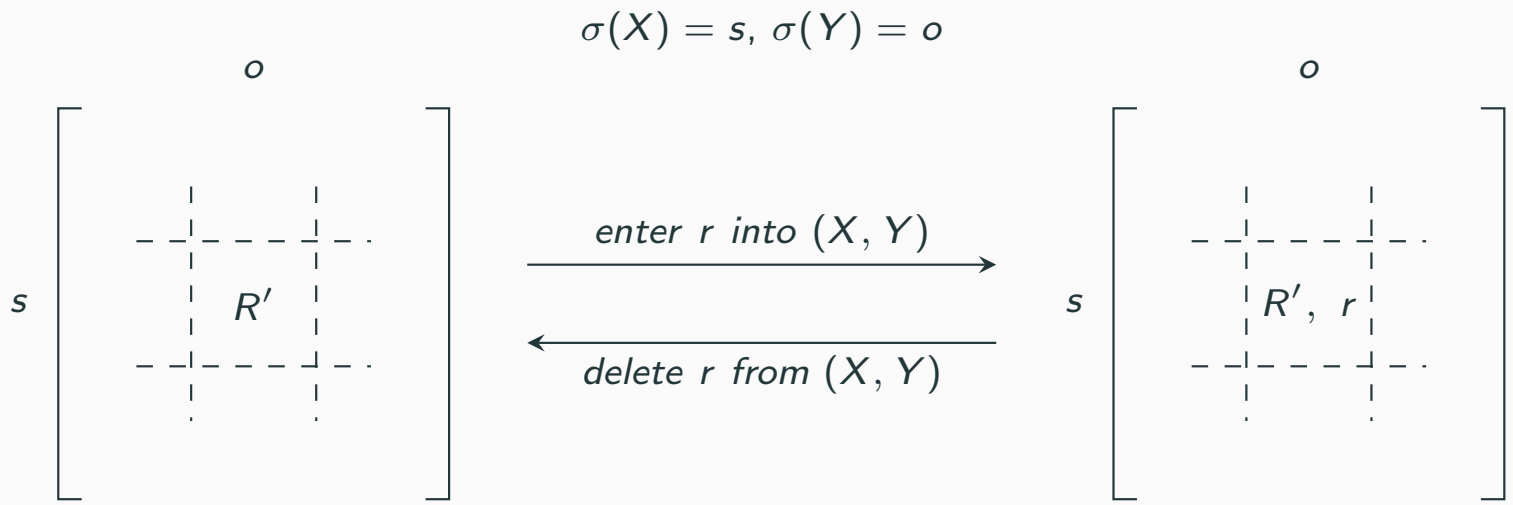
$$\sigma(\mathit{enter}\ r\ \mathit{into}\ (X, X')) = \mathit{enter}\ r\ \mathit{into}\ (\mathit{Alice}, \mathit{file})$$

Transition relation

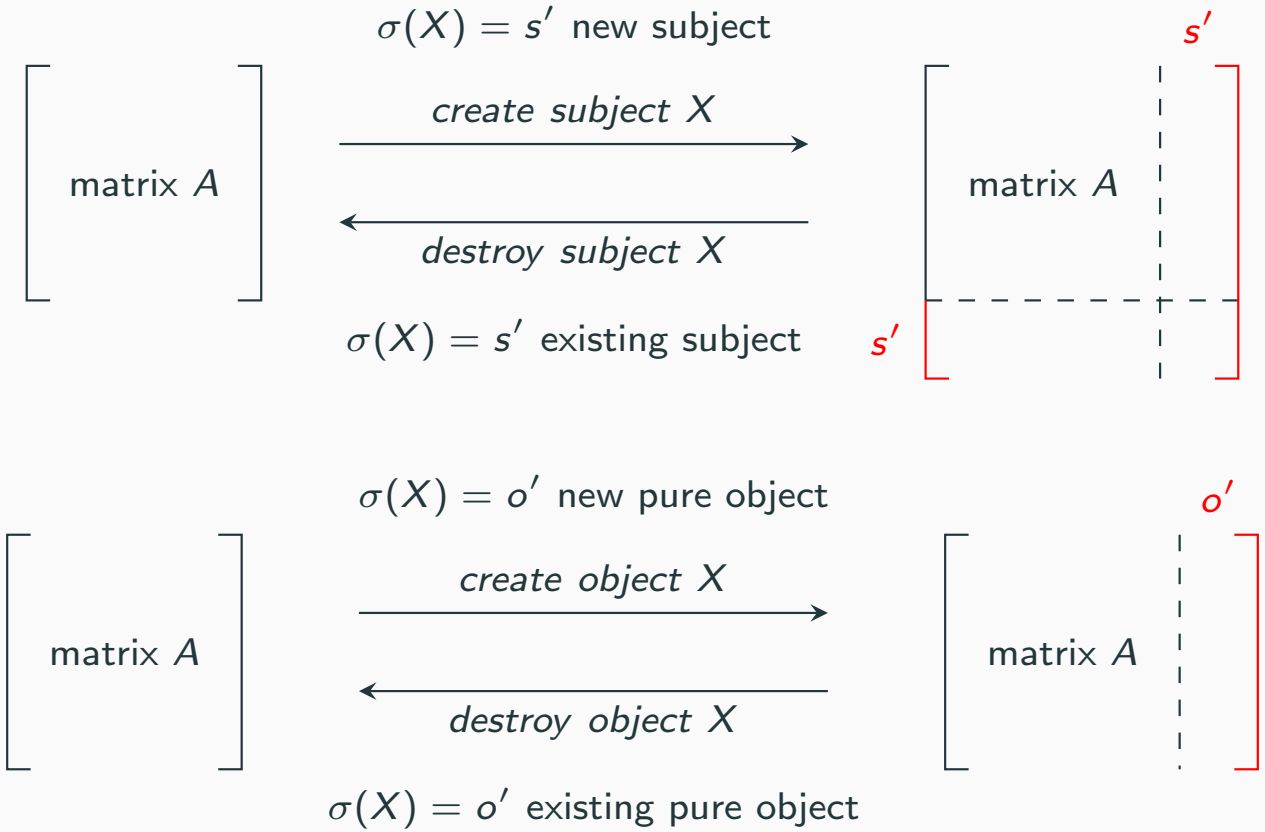
Given an operation *op* and a substitution σ, define the binary relation $\Rightarrow_{\sigma(op)}$ on states by

$$(S, O, A) \Rightarrow_{\sigma(op)} (S', O', A')$$

if and only if one of the following properties holds:



Transition relation



Then,

$$(S, O, A) \Rightarrow_{op} (S', O', A') \quad \Leftrightarrow \quad \exists \sigma : \ (S, O, A) \Rightarrow_{\sigma(op)} (S', O', A')$$

Transition relation

Given a command α and a substitution σ, define the binary relation $\Rightarrow_{\sigma(\alpha)}$ on states by $(S, O, A) \Rightarrow_{\sigma(\alpha)} (S', O', A')$ iff:

1. if the test of $\sigma(\alpha)$ is not satisfied at (S, O, A), then $(S', O', A') = (S, O, A)$;
2. if the test of $\sigma(\alpha)$ is satisfied at (S, O, A), then there exist $Q_0, Q_1, \ldots, Q_n$ such that

$$(S, O, A) = Q_0 \Rightarrow_{\sigma(op_1)} Q_1 \Rightarrow_{\sigma(op_2)} \cdots \Rightarrow_{\sigma(op_n)} Q_n = (S', O', A')$$

where $op_1, \ldots, op_n$ is the body of α.

Define then:

$$(S, O, A) \Rightarrow_{\alpha} (S', O', A') \quad \Leftrightarrow \quad \exists \sigma : \ (S, O, A) \Rightarrow_{\sigma(\alpha)} (S', O', A')$$

$$(S, O, A) \Rightarrow (S', O', A') \quad \Leftrightarrow \quad \exists \alpha : \ (S, O, A) \Rightarrow_{\alpha} (S', O', A')$$

Transition relation

Example 11 (Transition by *CONFER_READ*)

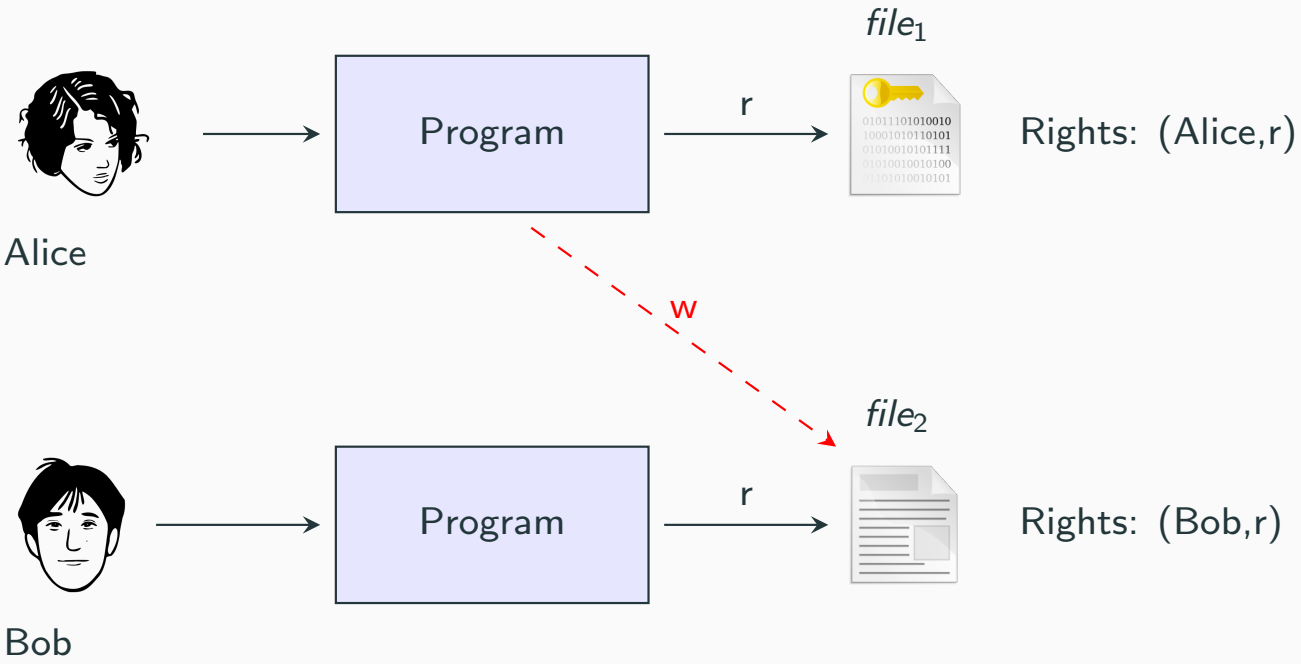
$$\begin{array}{c} \\ Alice \\ Bob \end{array} \begin{array}{c} \begin{array}{cccc} Alice & Bob & file_1 & file_2 \end{array} \\ \begin{pmatrix} & & own, r, w & r, w \\ & & & r \end{pmatrix} \end{array}$$

$$CONFER\_READ(Alice, Bob, file_1)$$

$$\begin{array}{c} \\ Alice \\ Bob \end{array} \begin{array}{c} \begin{array}{cccc} Alice & Bob & file_1 & file_2 \end{array} \\ \begin{pmatrix} & & own, r, w & r, w \\ & & r & r \end{pmatrix} \end{array}$$

The need for safety

In the example below, Alice only has the right to read $file_1$, and Bob only $file_2$.



If at a time, Alice gets the right of writing in $file_2$, it can write $file_1$ in $file_2$, thus giving Bob's ability to read, indirectly, $file_1$. Can we decide this?

Safety

Definition 12

A protection system over R is a finite set $\mathcal{C}$ of commands over R.

A protection system specifies the rules governing the rights granting in a system. Despite the name, it does not mean that it provides protection. About this, we have to talk further.

Definition 13

Let $\mathcal{C}$ be a protection system over R, Q a state of $\mathcal{C}$, $r \in R$, and $\alpha \in \mathcal{C}$. We say that α leaks r from Q if there exists a substitution σ such that:

1. the test of $\sigma(\alpha)$ is satisfied at Q;
2. there exist $Q_0, Q_1, \ldots, Q_i$ such that:
   - $Q_0 = (S_0, O_0, A_0) \Rightarrow_{\sigma(op_1)} \cdots \Rightarrow_{\sigma(op_i)} Q_i = (S_i, O_i, A_i)$;
   - $r \in A_i(s, o) - A_{i-1}(s, o)$ for some s and o,

   where $op_1, \ldots, op_i, \ldots, op_n$ is the body of α and $1 \leq i \leq n$.

Safety

Definition 14

Let $\mathcal{C}$ be a protection system over R, Q a state of $\mathcal{C}$, and $r \in R$. We say that $\mathcal{C}$ leaks r from Q if there exists a command of $\mathcal{C}$ that leaks r from Q.

Definition 15

Let $\mathcal{C}$ be a protection system over R, Q a state of $\mathcal{C}$, and $r \in R$. We say that Q is unsafe for r if there exists a reachable state Q' from Q such that $\mathcal{C}$ leaks r from Q'.

We say that Q is safe for r if it is not unsafe for r.

Remark 16

Leaks are not necessarily bad! Any interesting protection system has commands that leak some rights. However, these leaks should not occur at unauthorized states.

Undecidability of safety

The discussion so far shows us that it is imperative to be able to decide whether a protection system is safe or not for its rights.

Safety problem for protection systems

Instance: Protection system $\mathcal{C}$ over some set R of rights, state Q of $\mathcal{C}$, and $r \in R$

Question: Is Q safe for r?

Theorem 17 (Harrison et al. (1976))

The safety problem for bi-conditional (i.e., at most two conditions) monotonic (i.e., without delete and destroy operations) protection systems is undecidable.

Deciding safety

Even when decidable, the safety problem for practical protection systems is very complex.

Theorem 18 (Harrison et al. (1976))

1. *The safety problem for mono-operational (i.e., each command has exactly one operation) protection systems is NP-complete.*
2. *The safety problem for state-bounded (i.e., without create operations) protection systems is PSPACE-complete.*
3. *The safety problem for mono-conditional protection systems without destroy-operations is decidable.*

Most practical systems require multi-conditional commands!

Open Problem: safety for mono-conditional protection system.

Implementation

ACM implementations do not scale well: a bank with 50,000 staff and 300 applications would have an ACM of 15 million entries!

We need compact ways of storing and managing ACMs.

There are three approaches to implementing the ACM in a practical way:

1. Authorization tables – similar to relational tables in the database management;
2. Access control lists (ACL) – these are columns of the ACM;
3. Capability lists – these are rows of the ACM.

Moreover, we can use groups of subjects to manage their privileges simultaneously (e.g., Employees, Programmers, etc.). Groups need not be disjoint and a single authorization granted to a group can be enjoyed by all its members.

Authorization tables

An authorization table has three columns corresponding to subjects, privileges, and objects. Somehow, it unfolds the non-empty access control matrix cells.

Subject	Access	Object
$process_1$	r	$process_2$
$process_1$	r	*file*
$process_1$	w	*file*
$process_2$	r	*file*

The authorization table approach is generally used in database management systems, where authorizations are stored as catalogs (relational tables) of the database.

Access control lists

An access control list (ACL) is a column of the ACM (therefore, associated to an object - the ACL associated to o is denoted ACL_o, and it is stored along with o).

Advantages and disadvantages of ACLs:

- Simple to implement;
- Suited to environments where users manage their own file security;
- Less suited where the user population is large and constantly changing;
- Less suited where users want to be able to delegate their authority to run a particular program to another user for some set period of time;

Access control lists

Advantages and disadvantages of ACLs (continued):

- Security checking at runtime is difficult (usually, the operating system knows which user is running a particular program, rather than which files it has been authorized to access);
- Tedious to find all the files to which a user has access;
- Tedious to run system wide checks, such as verifying that no files have been left world-writable by users whose access was revoked.

Access control lists in Unix

- Every file or folder has associated access permissions. There are three types of permissions:
  - read access
  - write access
  - execute access
- Permissions are defined for three types of users:
  - the owner of the file
  - the group that the owner belongs to
  - anyone else (world)

  Each permission type has exactly two values, allowed or denied, specified by a bit.

Access control lists in Unix

Example 19

ACL for a file:

```
-rw-r----- Alice Accounts
```

The first bit specifies that the *ACL* is for a file, the next three bits give the access rights for the owner, the next three bits for the group, and the last three bits for anyone else. It follows then the owner name and the group name.

Example 20

ACL for a folder:

```
drwxrwxrwx Alice Accounts
```

The first bit specifies that the *ACL* is for a folder (directory); the next bits have the same meaning as above.

Access control lists in Unix

Unix / Linux offers special tools (such as programs) which enable unprivileged users to be able to accomplish tasks that require privileges (for instance, to run passwd programs):

- `suid` (set user id)
  - the owner of the program mark the program as `suid` (the bit `x` in owner *ACL* is set to `s` meaning both `x` and `suid`, or to `S` meaning only `suid`);
  - then, the program is placed in some folder where some user Bob has access;
  - Bob can run the program with the privilege of its owner;
- `sgid` (set group id) – things are similar, but the bit `s` occurs in the group area of bits.

This method may lead to serious security breaches!

Access control lists in Windows

- Permissions may be granted to users and groups (other than the resource's owner), with three values each: denied, allowed, and audit;
- Objects include files, folders, printers, registry keys, and Active Directory Domain Services (AD DS) objects;
- The permissions attached to an object depend on the type of object;
- Six types of permissions in Windows NT: read, write, execute, delete, change permissions (i.e., modify the *ACL*), take ownership (make current account the new owner);
- Types of permissions in Windows 10: read, modify, change owner, delete, full control, read & execute, write, list folder contents.

*ACL*s are lists of entries of the form

$$\cdots \texttt{ (user/group,permissions) } \cdots$$

For more details please see Microsoft (2021) docs.

Capability lists

An capability list (C-list) is a row of the ACM (therefore, associated to a subject - the C-list associated to s is denoted C_s, and it is stored along with s).

In practice, it is more convenient to store a C-list C_s as a list of pairs (o, r), where o is an object and r is a right (permission). Such a pair will be called a capability; then, C_s becomes a list of capabilities. Each capability acts like a ticket for s to access o with permission r. Therefore, capabilities are authentication tags.

This technique is used in EROS (Extremely Reliable Operating System), Hydra operating system (CMU), IBM System/38 and AS/400, Amoeba distributed operating system etc.

Capability lists

Problems with capabilities:

- How to represent object o in capability (o, r)?
  The use of o's address might not be a good idea if the address changes. A solution would be to use random bit strings, hash tables, and translation techniques (naming schemes);
- How to make capabilities unforgeable? There are a number of possibilities:
  - Hardware tags: 1-bit tag associated to the capability, showing that the capability can/cannot be changed or copied;
  - Protected address space: store capabilities in parts of memory that are not accessible to programs;
  - Language-based security: use of a programming language to enforce restrictions on access and modification to capabilities;
  - Cryptography: use encryption.

ACLs vs capabilities

	ACLs	Capabilities
Authentication	require subject auth and ACL integrity	require integrity and propagation control
Access review	suited on object-basis	suited on subject-basis
Revocation	suited on object-basis	suited on subject-basis
Least privilege control w.r.t. subjects	–	provide for finer-grained, especially dynamic short-lived subjects created for specific tasks

ACLs are preferred to capabilities: most systems use ACLs.

Other related DAC models

The take-grant model

It was proposed by Lipton and Snyder (1977). Basic features:

- It is a state-transition system;
- Subjects are not objects;
- States are directed graphs whose nodes are subjects and objects, and whose arcs are labeled by sets of rights;
- There are two special rights: take (t) and grant (g):
  - if x has the right t for y, then x can "borrow" from y all his rights;
  - if x has the right g for y, then x can "lend" to y all his rights;
- In this model, safety can be checked in polynomial time in the size of the initial graph (state);
- Main drawback: the model is a too little expressive, although one can use it to check certain security properties in computer networks (Shahriari and Jalili (2007)).

The Schematic Model

It was proposed by Sandhu (1988) to fill the gap between the richness in expressive power of the HRU model and its intractability w.r.t. the safety question as compared with the limited applicability of the take-grant model but efficient decidability of safety.

The schematic model subsumes several well-known protection models in terms of expressive power and safety analysis (Sandhu (1992)).

The model is of more theoretical than practical interest.

Adding new features to the DAC model

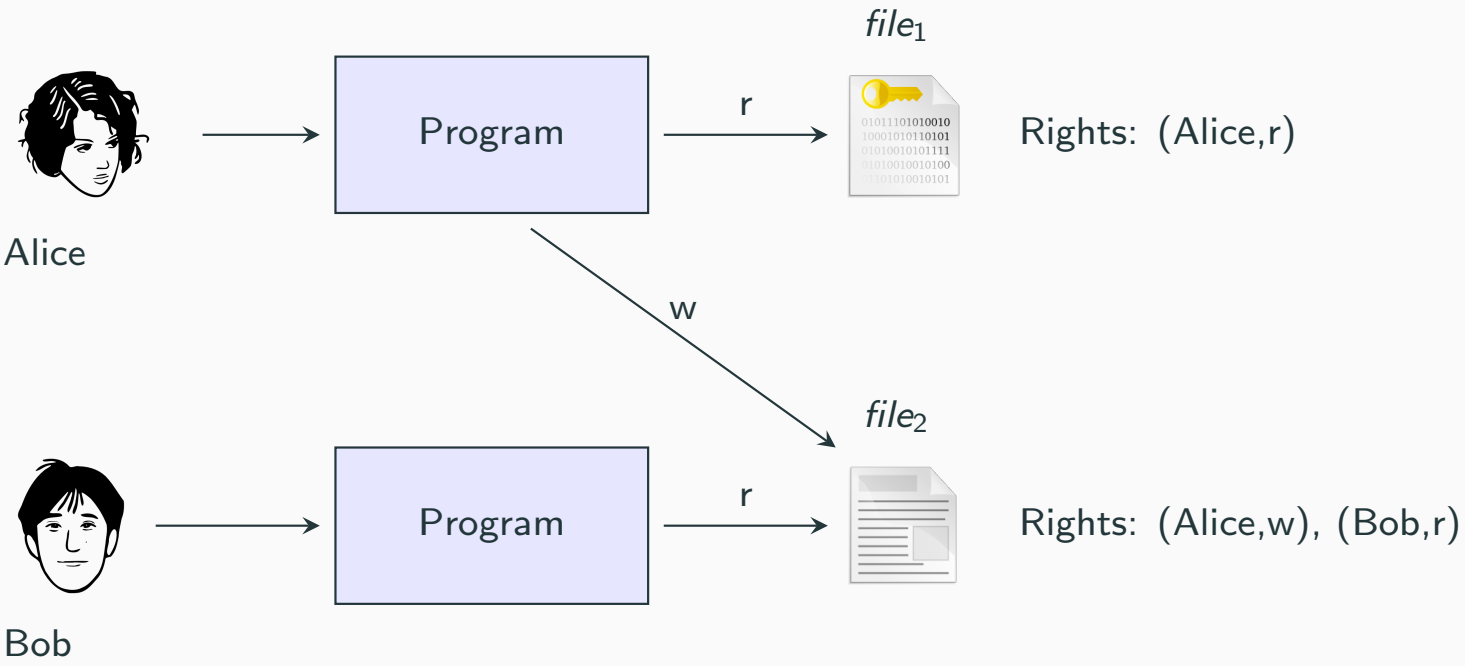
- Positive and negative permissions – their combination is a convenient way to express exceptions: a positive right to a group (shared thus by all its members) and a negative right to a member of the group would exclude that member from getting access. Example of use: Windows;
- Weak and strong permissions – weak permissions override each other, while strong permissions override weak permissions (no matter their specificity) and cannot be overridden;
- Implicit and explicit permissions – implicit permissions can be derived in the system;
- Context-based permissions;
- Content-dependent permissions.

For more details please see Samarati and de Capitani di Vimercati (2001).

A major weakness of DAC models

We may trust users ...

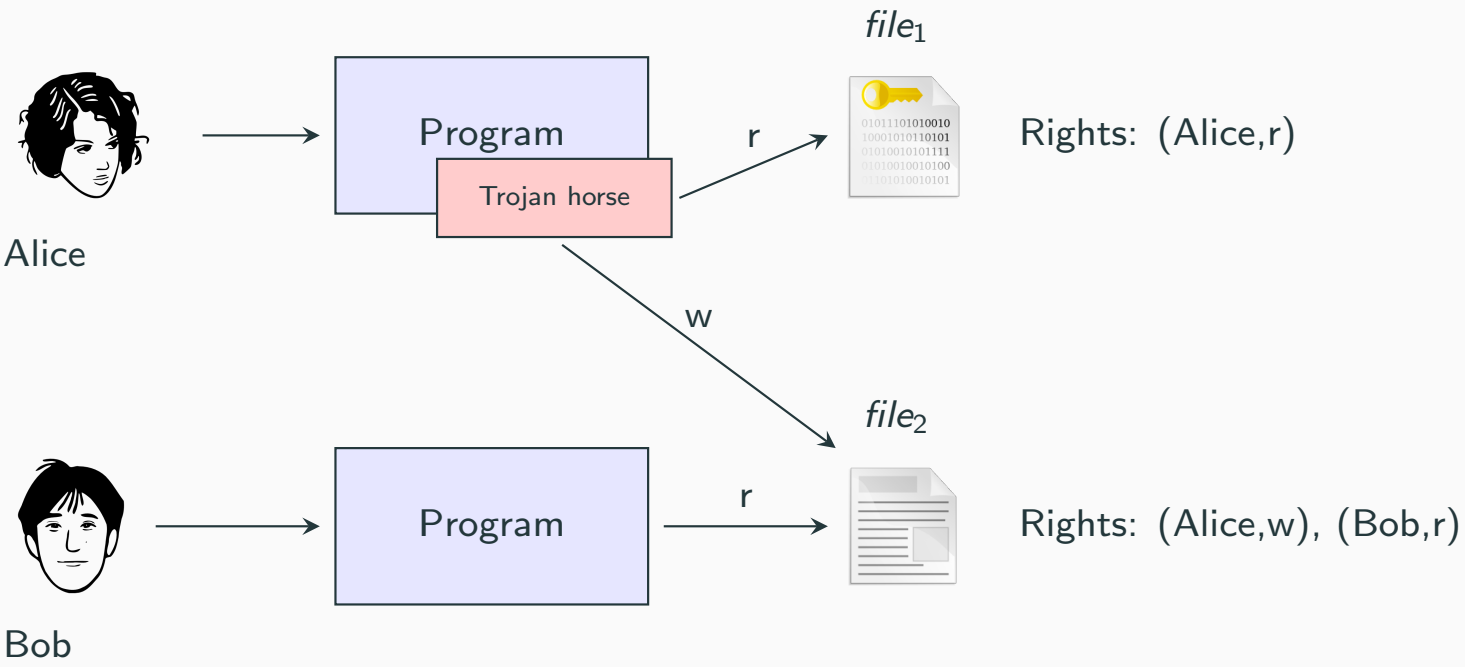
We assume that Alice and Bob are trustworthy, and even if Alice has the right to write in $file_2$, she will not write $file_1$ in $file_2$ to allow Bob to read $file_1$.



Assuming the users obey the access restrictions, the subjects operating on their behalf do the same?

... but not subjects!

A Trojan horse is a rogue software that may perform illegitimate actions unknown to the caller, exploiting access privileges of the caller.



The Trojan horse takes Alice's privileges, reads $file_1$, and copies it in $file_2$, allowing Bob to read it!

Concluding remarks on DAC models

Concluding Remarks

- DAC policies enforce access control on the basis of the identity of the requester and explicit access rules;
- DAC policies ignore the distinction between users and subjects and evaluate all requests submitted by a process (subject) running on behalf of some user against the authorizations of the user;
- DAC policies do not offer protection against processes that execute malicious programs (such as Trojan Horses) exploiting the privileges of the user on behalf of whom they are executing;
- DAC policies do not enforce any control on the flow of information once this information is acquired by a process.

Two things that emerge from our discussion: the need for separating users from subjects and controlling the flow of information!

Readings

In addition to the materials indicated so far, I additionally recommend:

- Chapter 6 of Conrad et al. (2016);
- Chapters 3 and 4 of Andress (2014);
- Chapter 11 of Collins (2014);
- Chapter 23 of Bertino (2012);
- Samarati and de Capitani di Vimercati (2001).

References

Andress, J. (2014). *The Basics of Information Security. Understanding the Fundamentals of Infosec in Theory and Practice*. Syngress, Elsevier, Boston, 2nd edition.

Bertino, E. (2012). Chapter 23 - Policies, access control, and formal methods. In Das, S. K., Kant, K., and Zhang, N., editors, *Handbook on Securing Cyber-Physical Critical Infrastructure*, pages 573–594. Morgan Kaufmann, Boston.

Collins, L. (2014). Chapter 11 - Access controls. In Vacca, J. R., editor, *Cyber Security and IT Infrastructure Protection*, pages 269–280. Syngress, Boston.

Conrad, E., Misenar, S., and Feldman, J. (2016). *CISSP Study Guide*. Singress, Elsevier, 3rd edition.

Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. (1976). Protection in operating systems. *Commun. ACM*, 19(8):461–471.

Lipton, R. J. and Snyder, L. (1977). A linear time algorithm for deciding subject security. *J. ACM*, 24(3):455–464.

Microsoft (2021). Windows security. Technical report, Microsoft.

References (cont.)

Samarati, P. and de Capitani di Vimercati, S. (2001). Access control: Policies, models, and mechanisms. In Focardi, R. and Gorrieri, R., editors, *Foundations of Security Analysis and Design*, pages 137–196, Berlin, Heidelberg. Springer Berlin Heidelberg.

Sandhu, R. S. (1988). The schematic protection model: Its definition and analysis for acyclic attenuating schemes. *J. ACM*, 35(2):404–432.

Sandhu, R. S. (1992). Expressive power of the schematic protection model. *J. Comput. Secur.*, 1(1):59–98.

Shahriari, H. R. and Jalili, R. (2007). Vulnerability Take Grant (VTG): An efficient approach to analyze network vulnerabilities. *Computers & Security*, 26:349–360.