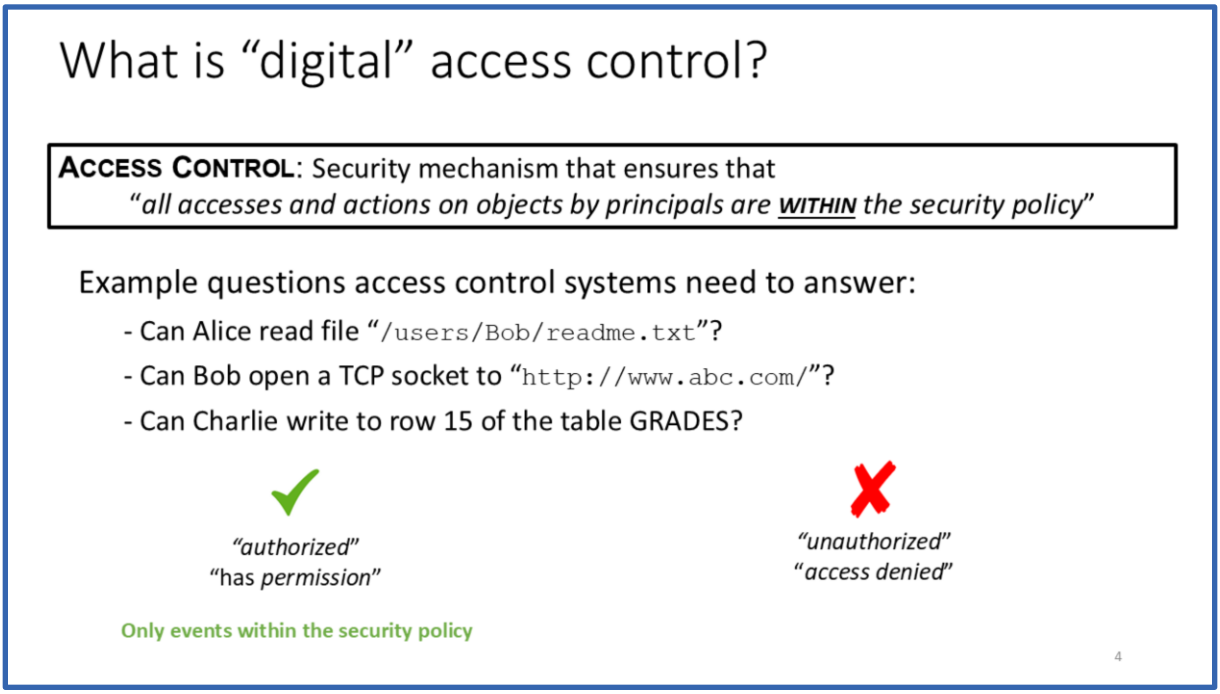
Computer Security and Privacy
Access Control

Slides made by Carmela Troncoso, SPRING Lab, additions by Thomas Bourgeat
Some slides/ideas adapted from: Philippe Oechslin, George Danezis, Emiliano de Cristofaro, Gianluca Stringhini

EPFL

Computer Security and Privacy
Access control Introduction

Access control is a concept that we have in the physical world, where we often have resources to which not everyone should have access.
In the physical world, we have many means to restrict access to resources (e.g., keys to access homes or cars). Sometimes these physical means rely on digital support (e.g., access via CAMIPRO card).



The most basic security mechanism is **access control**. This mechanism’s goal is to ensure that every action in a system (reading a file, writing on a socket, accessing an entry on a database, …) always respects the security policy.

If every action respects the security policy, then the system is secure (of course, within the threat model considered by the security policy).

Given a tuple “principal-object-action”, an access control mechanism returns one of two outcomes:
- action *authorized*, meaning that the principal has permission to execute the action.
- action *unauthorized*, meaning that the principal has been denied access to the object in question.

Why is so important to learn about access control?

Access control is the **first line of defense**. Thus, **it is used everywhere**

Applications

Online Social Networks, Email server, Cloud storage

Middleware

Databases Management Systems (DBMS)

Operating System

control access to files, directories, ports,…

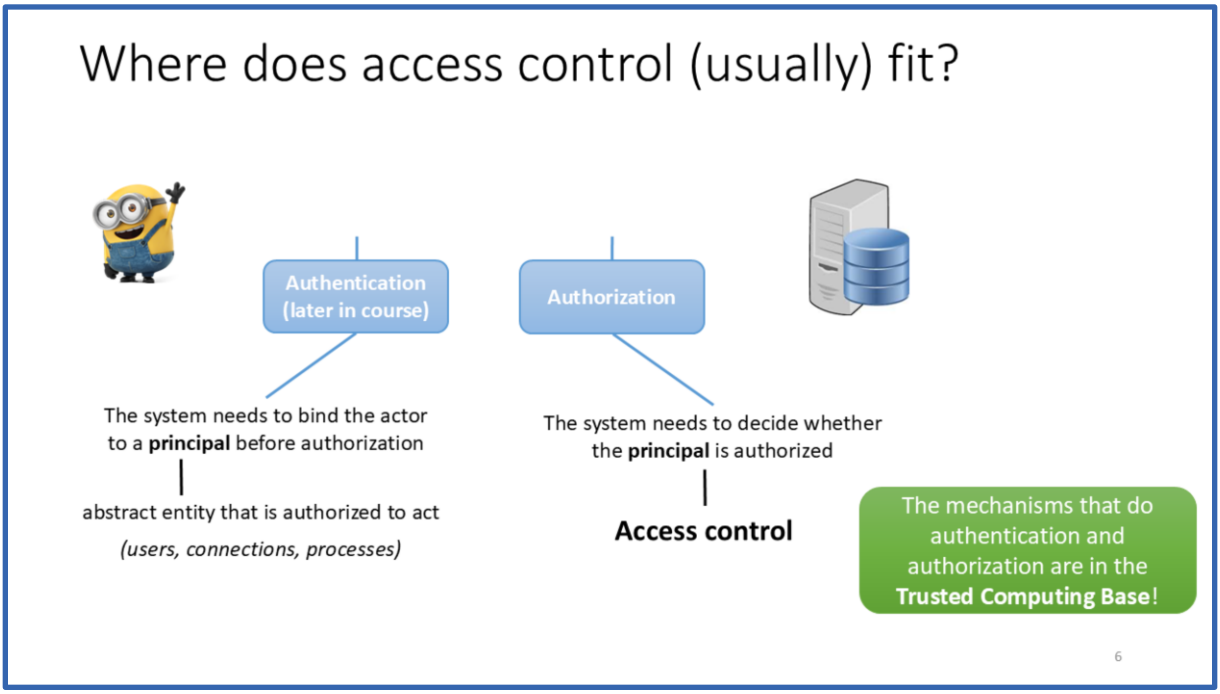
Hardware

Memory, register, privileges

5

Access control is also the most pervasive security mechanism. You (or your computer, or your phone) use it everyday. From the application level, exposed to the user, where the system decides whether a user has access to a resource, to lower levels: middleware supporting applications (e.g., deciding which database tables can be accessed), operative systems (e.g., deciding which ports), and even hardware that controls which parts of the memory can be accessed.

Access control is a cornerstone of computer security, and it is very important to get it correct when engineering security into systems.



Access control helps the system decide about whether to *authorize* a principal (in the example the minion Bob) to access a resource to execute a particular action (in the example write on a database).

Authorization refers to a tuple principal-access-object. As such, it needs to know who is the principal! The security mechanism to bind a principal to the actor that wants to perform the access is called **Authentication** and we study it later in the course.

Both authentication and access control are key mechanisms to enforce the security policy. If authentication or access control fails, the security policy is breached. Thus, these mechanisms **must** be inside the TCB.

Implementing access control

What NOT to do: "Checks soup"

- All over the program, add checks
    - implementing the decision in-line based on the policy

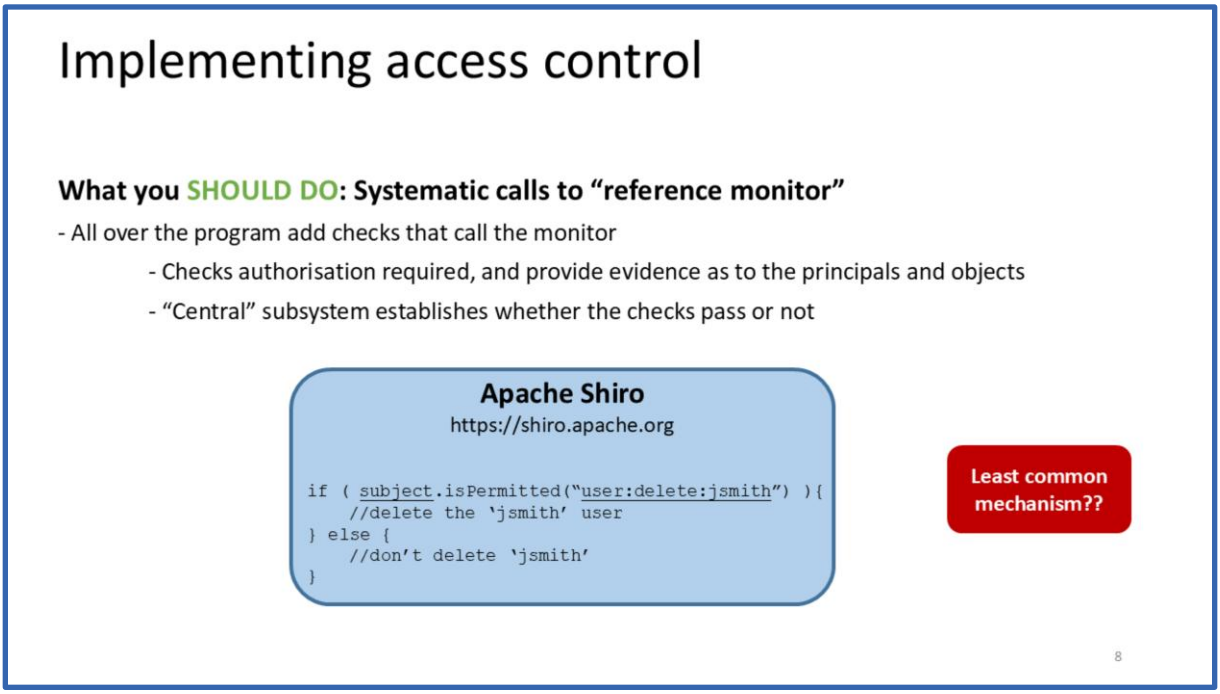
```
#some code that needs to access file3.txt

if (action == read) and ((userID == Alice) or (userID == Bob) :
        open(file3.txt, 'r')
elif (action == write) or (userID == Bob) then:
        open(file3.txt,'w')
else:
        print("The user does not have access to file3.txt")
```

7

Some advice on implementing access control:

Do not check everywhere in a program!

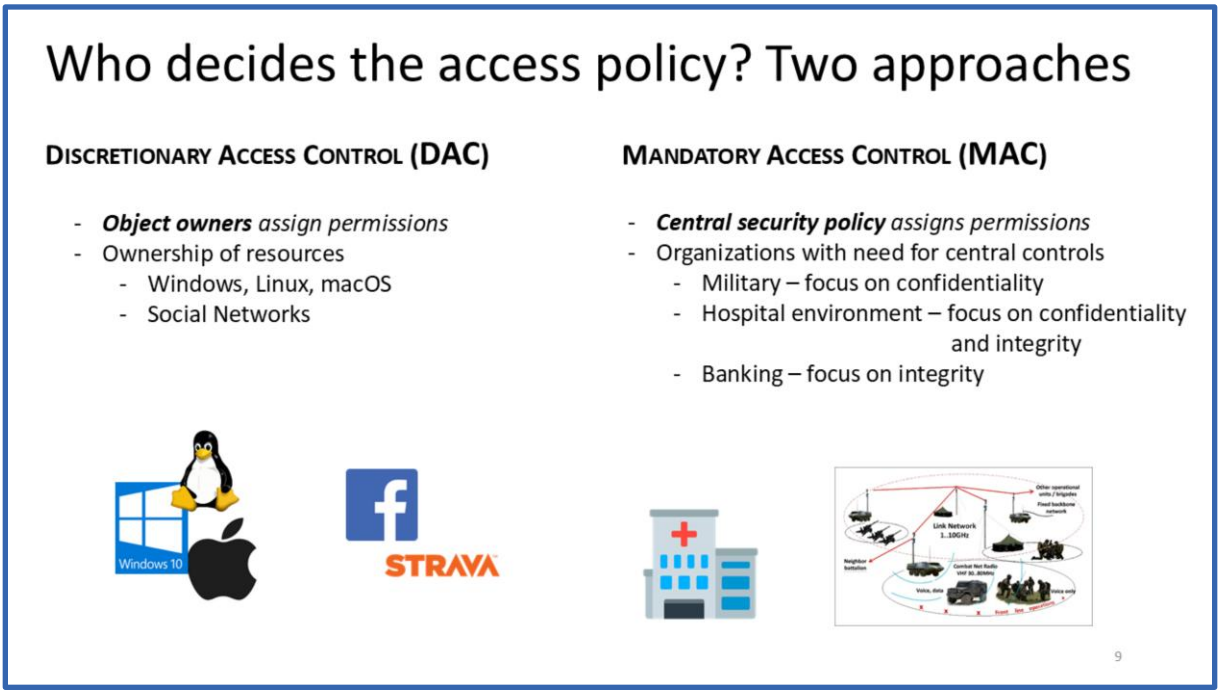
1) It is prone to errors. What if you missed any permission? How to check correctness or debug?
2) It is very hard to change. If anything changes in the policy one has to go in the code and find every place the check is made. What if we have thousands of lines of code?



What one should do is to have **one** only place where everything is checked. This central system is called the **Reference monitor.**

The reference monitor indeed violates the least common mechanism principle. If the reference monitor fails or is compromised, the security policy can be breached. On the other hand it fulfils economy of mechanism, as it keeps the access control in one place where it is easy to check. It also supports the principle of complete mediation, as ideally should check that subjects have permissions to perform all actions.

This is one example of a case where not all principles can be followed at the same time, and the one that is better for security is used.



An important question regarding access control is who decides which principals are authorized to have which access to which objects. There are two main alternatives:

In **Discretionary Access Control (DAC)** the permissions of one object is decided by the owner of that object. Not all objects are owned by the same entity, i.e., there is not one central authority. Objects are typically owned by users, who decide the permissions on the object. As systems tend to have a very large amount of objects, it is common that instead of allowing owners fine-grained access control decisions, they can choose from a pre-defined set of options.

- This is the access control mode for operative systems, such as Windows or Linux, which we revisit later in this lecture.
- This is also the access control mode for social networks, where users decide which other users have access to the content they publish on the network.

In **Mandatory Access Control (MAC)** the decision is made by a *central authority*. This authority defines the permissions assigned to each principal with respect to each object. Mandatory access control are typical to *organizations* in which decisions need to be made centrally and affect *all* the individuals in the organization (it is a *system-wide* policy). Depending on the needs of the organization, the MAC policies focus on one or more

security properties:

- Military organizations have strong focus on confidentiality. In these organizations the most important property is that secret information is not accessible to lower rank members. Integrity of course is also important, but on a lesser level. It is better that the battle plans are corrupted or deleted, than they are leaked to the enemy.
- Hospitals have strong needs for both integrity (keeping the information unchanged, and not modified by unauthorized parties), and confidential (ensuring that patient medical records are only accessed with those with need to do so in order to deliver medical treatment).
- Banks most important requirement is integrity. The accounting must not be tampered with by any unauthorized party! Confidentiality of course is also important, people should not learn about others earnings and expenses, but the leakage of this information is less critical than a modification of the bank records redistributing money among users.

Modern Access Control systems are a hybrid between both.

EPFL

Computer Security (COM-301)

Discretionary Access control

Protection mechanisms in computers

B. Lampson. Protection. Proc. 5th Princeton Conf. on Information Sciences and Systems, Princeton, 1971. Reprinted in ACM Operating Systems Rev. 8, 1 (Jan. 1974), pp 18-24.

The original motivation for putting protection mechanisms into computer systems was to keep one user's malice or error from harming other users. Harm can be inflicted in several ways:

a) by destroying or modifying another user's data;
b) by reading or copying another user's data without permission;
c) by degrading the service another user gets, e.g. using up all the disk space or getting more than a fair share of the processing time. An extreme case is a malicious act or accident which crashes the system - this might be considered the ultimate degradation.

More recently it has been realized that all of the above reasons for wanting protection are just as strong if the word 'user' is replaced by 'program'.

Implementing Discretionary Access Control

- ***Object owners*** *assign permissions*
- Ownership of resources
  - Windows, Linux
  - Social Networks

How??

Permissions establish which subjects can access which objects

But in a system there are many subjects and objects.

There can be many subjects, many objects, and many combinations of permissions combining subjects and objects, how can we handle?

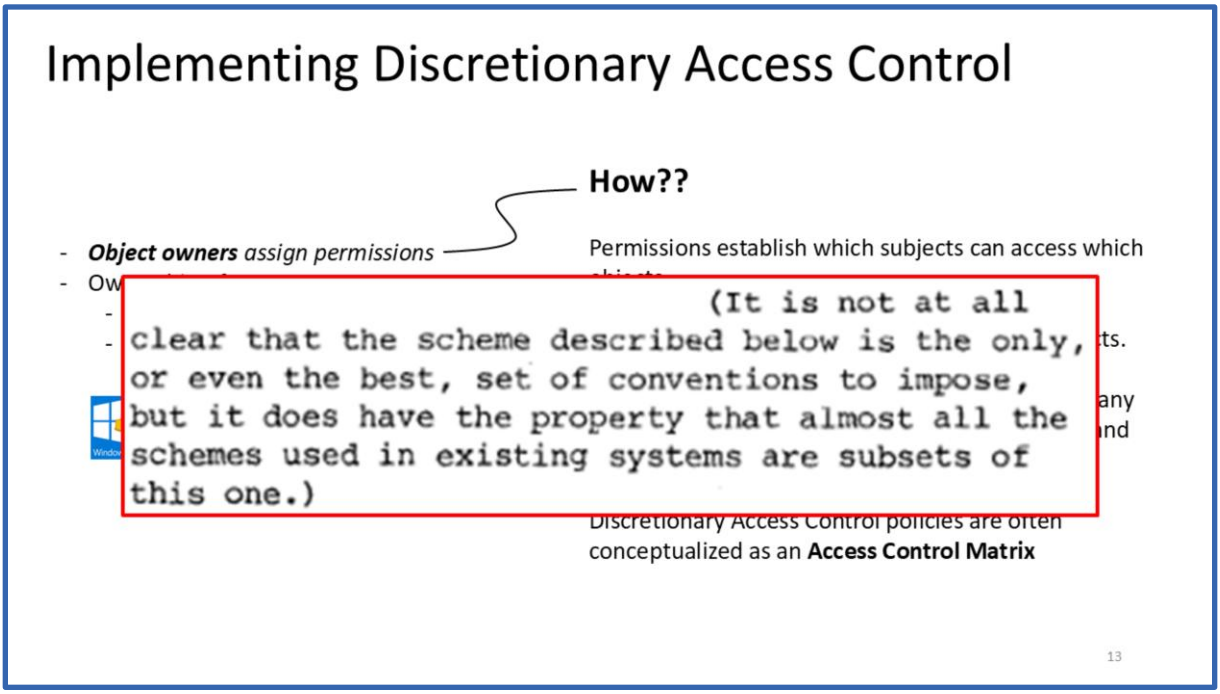
Discretionary Access Control policies are often conceptualized as an **Access Control Matrix**

12

In **Discretionary Access Control (DAC)** the permissions of one object is decided by the owner of that object. Not all objects are owned by the same entity, i.e., there is not one central authority. Objects are typically owned by users, who decide the permissions on the object. As systems tend to have a very large amount of objects, it is common that instead of allowing owners fine-grained access control decisions, they can choose from a pre-defined set of options.

- This is the access control mode for operative systems, such as Windows or Linux, which we revisit later in this lecture.
- This is also the access control mode for social networks, where users decide which other users have access to the content they publish on the network.

DAC policies are typically conceptualized as an **Access Control Mac**



In **Discretionary Access Control (DAC)** the permissions of one object is decided by the owner of that object. Not all objects are owned by the same entity, i.e., there is not one central authority. Objects are typically owned by users, who decide the permissions on the object. As systems tend to have a very large amount of objects, it is common that instead of allowing owners fine-grained access control decisions, they can choose from a pre-defined set of options.

- This is the access control mode for operative systems, such as Windows or Linux, which we revisit later in this lecture.
- This is also the access control mode for social networks, where users decide which other users have access to the content they publish on the network.

DAC policies are typically conceptualized as an **Access Control Mac**

The Access Control Matrix

ACCESS CONTROL MATRIX: an *abstract* representation of all **permitted** triplets of (subject, object, access right) within a system

Subjects (principals): entity within an IT system

a user, a process, a service

Objects(assets): resources that (some) subject may access or use

a file, a folder, a row in a database, the system's memory, a machine in the network, a printer, a page in a website

Operations: in abstract, subjects can observe and/or alter objects

read, write, append, execute

B. Lampson. Protection. Proc. 5th Princeton Conf. on Information Sciences and Systems, Princeton, 1971. Reprinted in ACM Operating Systems Rev. 8, 1 (Jan. 1974), pp 18-24.

14

Access control is complex. The number of principals and objects in a system is huge. To reason about the permitted triples, in 1974, Lampson introduced the access control matrix. An access control matrix **M** has **|S|** rows, one per subject in the system; and **|O|** columns, one per object in the system. Every element $\mathbf{M_{so}}$ of the matrix is an access operation (read, write, execute, replace, delete, etc). It specifies if the subject in row **s** is authorized to perform the given operation on the object in row **o**.

Access Control Matrix - Example

S … Alice, Bob

O … file1, file2, file3

A … read, write

Can Alice read file1?
Can Bob write file1?
Can file3 be written by Alice?

Access control matrix:

	file1	file2	file3
Alice	read write		read
Bob		read write	read write

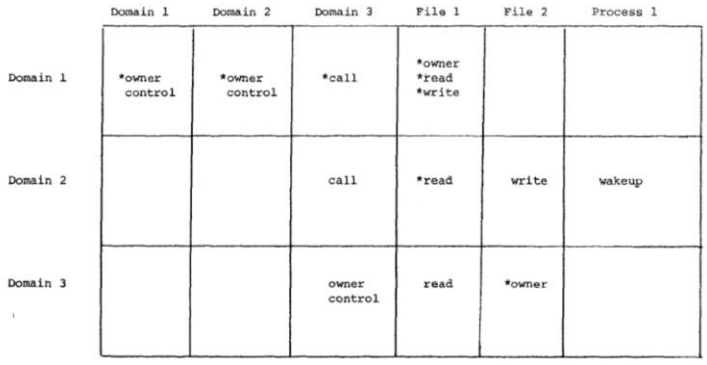
15

Here we have an example with:
- Two **subjects**: Alice and Bob
- Three **objects**: file1, file2 and file3
- Two possible **access operations**: read and write

The access control matrix says that:
- Alice can read and write on file1, and read file3, Alice *cannot* do any action on file2
- Bob can read and write file2, and read and write file3 , Bob *cannot* do any action on file1

Answering the questions:
- Alice *can* read file1
- Bob *cannot* write file1
- Alice *cannot write* on file3

Access Control Matrix – Original Example



	Domain 1	Domain 2	Domain 3	File 1	File 2	Process 1
Domain 1	*owner control	*owner control	*call	*owner *read *write		
Domain 2			call	*read	write	wakeup
Domain 3			owner control	read	*owner	

*copy flag set

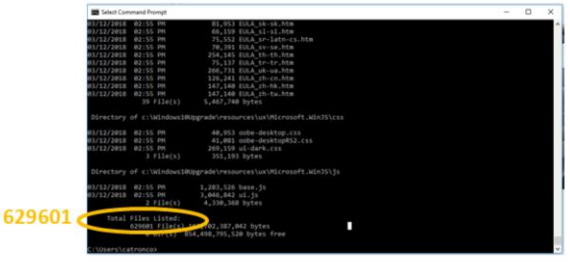
Figure 1: Portion of an access matrix

The access control matrix is more general than just for users accessing files. In the original paper, the example given was actually already more sophisticated.

The Access Control Matrix **is an abstract concept**

Not for direct implementation

what if there are thousands of files or hundreds of users?



O(f·u)

1 bit per file, 1 user	78KB
3 bits per file, 1 user	236KB
3 bits per file, 10 users	2.36MB

More issues:

usually very sparse – wasteful

inefficient to access – need to keep the matrix in fast memory?

extensibility – add a file? add a user?

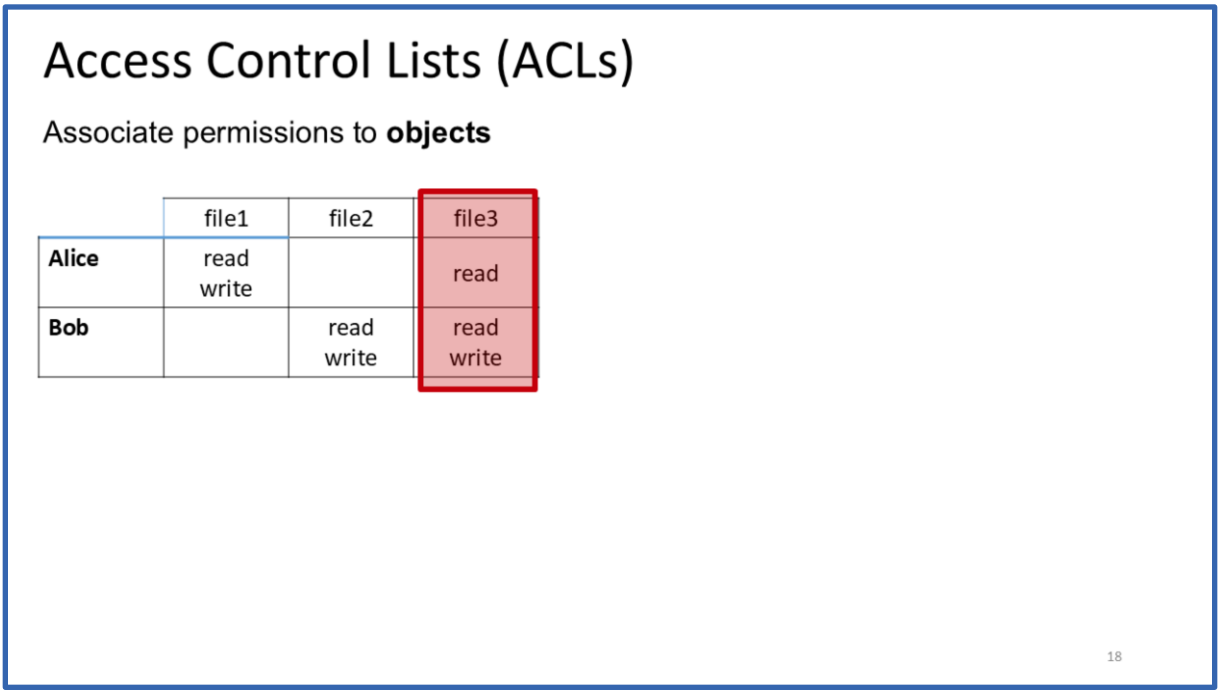
17

The Access Control Matrix is a great abstract way to reason about permissions, but storing it as in the definition is impractical. The size of the matrix is in the order of f*u, where **f** is the number of files in the system, and **u** is the number of users.

There are more issues that hinder the implementation of the matrix:

1) It is very **sparse**: typically each user has access to a "handful" of files (a small percentage of the total files in the system). Thus, most of the matrix is full of empty entries indicating that the user does NOT have permission to access a file. Thus, most of the huge size of the matrix is dedicated to non useful information.
2) The matrix would need to be kept in fast memory to be checked all the time
3) Extensibility: What to do when adding a user or when adding a file? Problem of *incrementality*

We will see two ways of implementing the matrix: associating permissions to objects (follow the matrix columns) or associate permissions to subjects (follow the matrix' rows)



A first option is to store the matrix taking information in columns, i.e., associate the permissions to **objects**. This is called an object's **Access Control List (ACL)**. For each object, the ACL contains a tuple for each subject expressing the permissions of that subject for the object.

For file1, the ACL contains `file1: {(Alice,read/write)}` indicating that Alice has read and write permissions on file1.

Note that, as Bob has no permissions the ACL for file1 *contains **no** information about Bob!*

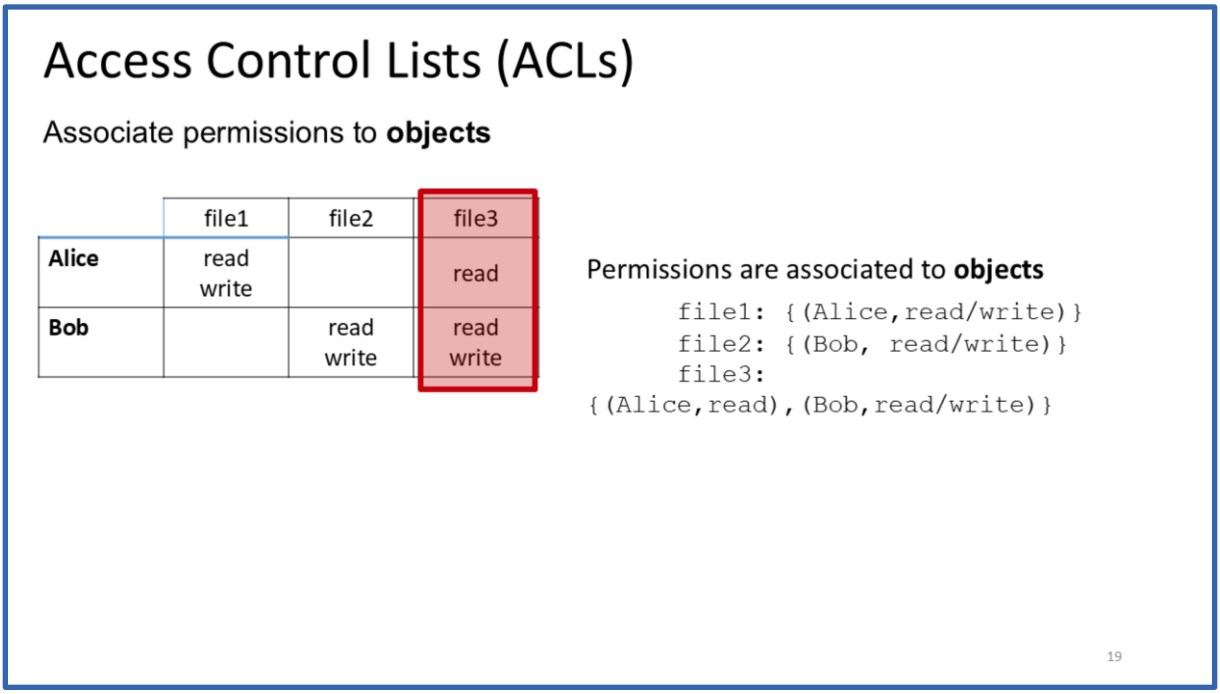
ACLs have the following advantages. They can be stored with the resource (and sent with the resource), and thus it is easy to quickly determine who has access to the resource. It is also easy to revoke permissions by resource: one finds the ACL close to the resource and eliminate the permission.

ACLs also have drawbacks:

- They are difficult to check at runtime, as in many cases it is not clear who is the subject accessing (see confused deputy below)
- It is difficult to audit all rights of a user (one has to go file by file checking if the user has some rights! This grows linearly with the number of files). This makes also very difficult to remove *all* the permissions of a user (error-prone process). If this needs to be done, may be

easier to delete the user as a whole, e.g., by revoking his authentication credentials (see Authentication lecture).

- It is difficult to delegate permissions. As it is hard to associate a permission to a user, it is also difficult find out whether a user has permission to delegate, and then difficult to temporarily assign a permission to other user (as it is difficult to remove this permission later)



A first option is to store the matrix taking information in columns, i.e., associate the permissions to **objects**. This is called an object's **Access Control List (ACL)**. For each object, the ACL contains a tuple for each subject expressing the permissions of that subject for the object.

For file1, the ACL contains `file1: {(Alice,read/write)}` indicating that Alice has read and write permissions on file1.

Note that, as Bob has no permissions the ACL for file1 *contains **no** information about Bob!*

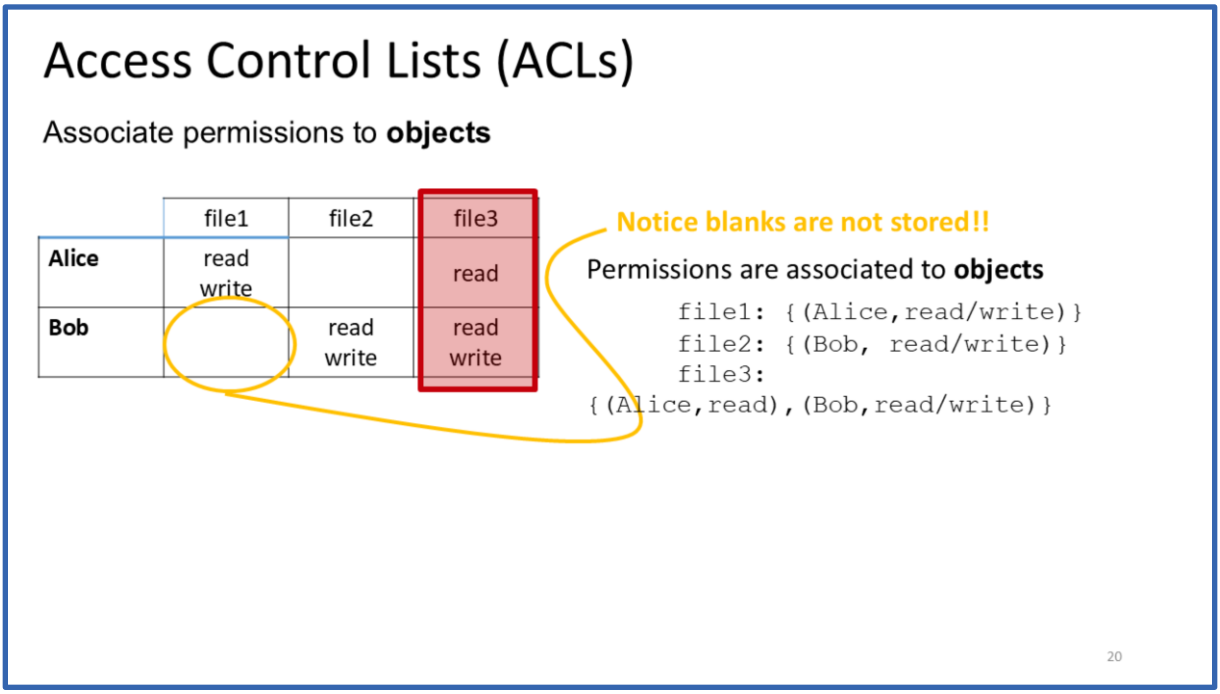
ACLs have the following advantages. They can be stored with the resource (and sent with the resource), and thus it is easy to quickly determine who has access to the resource. It is also easy to revoke permissions by resource: one finds the ACL close to the resource and eliminate the permission.

ACLs also have drawbacks:

- They are difficult to check at runtime, as in many cases it is not clear who is the subject accessing (see confused deputy below)
- It is difficult to audit all rights of a user (one has to go file by file checking if the user has some rights! This grows linearly with the number of files). This makes also very difficult to remove *all* the permissions of a user (error-prone process). If this needs to be done, may be

easier to delete the user as a whole, e.g., by revoking his authentication credentials (see Authentication lecture).

- It is difficult to delegate permissions. As it is hard to associate a permission to a user, it is also difficult find out whether a user has permission to delegate, and then difficult to temporarily assign a permission to other user (as it is difficult to remove this permission later)



A first option is to store the matrix taking information in columns, i.e., associate the permissions to **objects**. This is called an object's **Access Control List (ACL)**. For each object, the ACL contains a tuple for each subject expressing the permissions of that subject for the object.

For file1, the ACL contains `file1: {(Alice,read/write)}` indicating that Alice has read and write permissions on file1.

Note that, as Bob has no permissions the ACL for file1 *contains **no** information about Bob!*

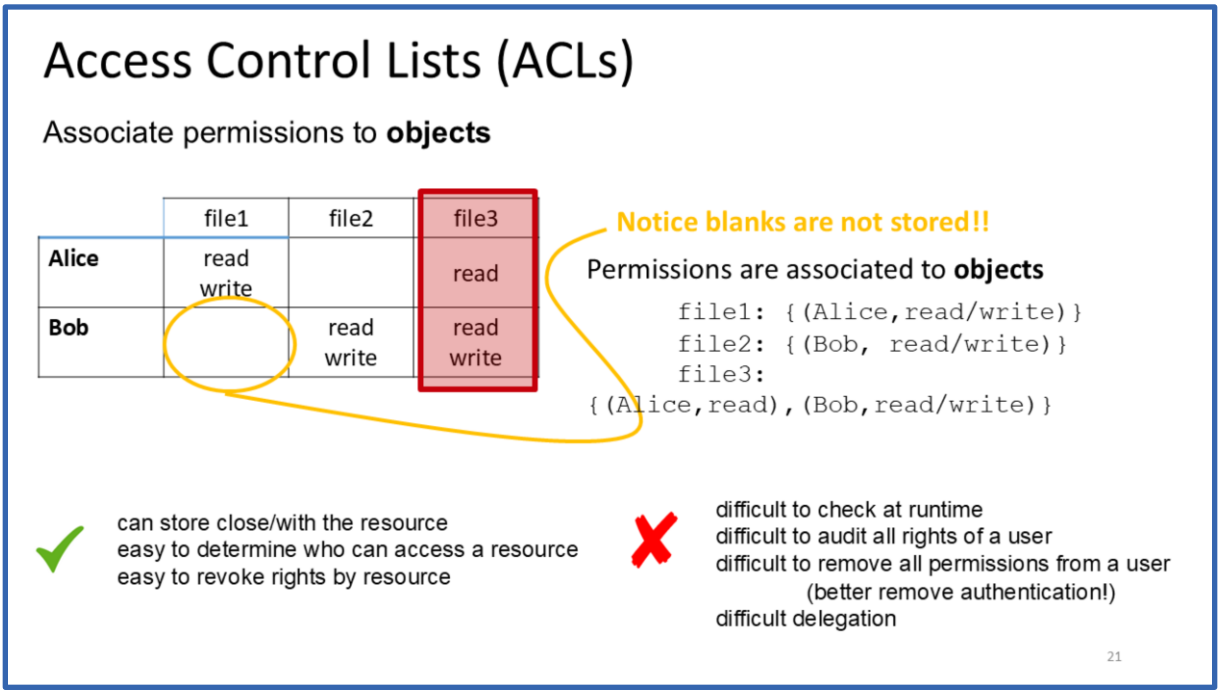
ACLs have the following advantages. They can be stored with the resource (and sent with the resource), and thus it is easy to quickly determine who has access to the resource. It is also easy to revoke permissions by resource: one finds the ACL close to the resource and eliminate the permission.

ACLs also have drawbacks:

- They are difficult to check at runtime, as in many cases it is not clear who is the subject accessing (see confused deputy below)
- It is difficult to audit all rights of a user (one has to go file by file checking if the user has some rights! This grows linearly with the number of files). This makes also very difficult to remove *all* the permissions of a user (error-prone process). If this needs to be done, may be

easier to delete the user as a whole, e.g., by revoking his authentication credentials (see Authentication lecture).

- It is difficult to delegate permissions. As it is hard to associate a permission to a user, it is also difficult find out whether a user has permission to delegate, and then difficult to temporarily assign a permission to other user (as it is difficult to remove this permission later)



A first option is to store the matrix taking information in columns, i.e., associate the permissions to **objects**. This is called an object's **Access Control List (ACL)**. For each object, the ACL contains a tuple for each subject expressing the permissions of that subject for the object.

For file1, the ACL contains `file1: {(Alice,read/write)}` indicating that Alice has read and write permissions on file1.

Note that, as Bob has no permissions the ACL for file1 *contains **no** information about Bob!*

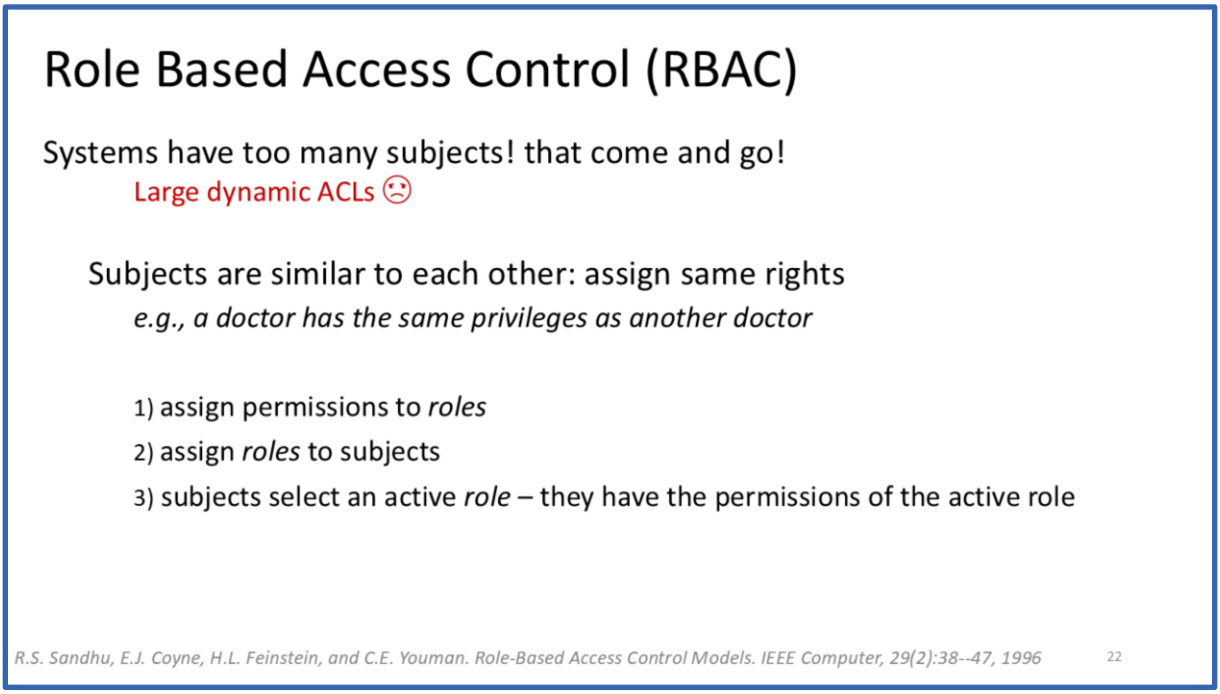
ACLs have the following advantages. They can be stored with the resource (and sent with the resource), and thus it is easy to quickly determine who has access to the resource. It is also easy to revoke permissions by resource: one finds the ACL close to the resource and eliminate the permission.

ACLs also have drawbacks:

- They are difficult to check at runtime, as in many cases it is not clear who is the subject accessing (see confused deputy below)
- It is difficult to audit all rights of a user (one has to go file by file checking if the user has some rights! This grows linearly with the number of files). This makes also very difficult to remove *all* the permissions of a user (error-prone process). If this needs to be done, may be

easier to delete the user as a whole, e.g., by revoking his authentication credentials (see Authentication lecture).

- It is difficult to delegate permissions. As it is hard to associate a permission to a user, it is also difficult find out whether a user has permission to delegate, and then difficult to temporarily assign a permission to other user (as it is difficult to remove this permission later)



Real systems may have a lot of users, whose permissions change over time (imagine Facebook's ACLs), so we end up with very large ACLs that are unmanageable.

To make the management easier, a first alternative is what is called **Role Based Access Control (RBAC)**.

Under the observation that subjects are similar in terms of permission (e.g., all doctors have the same privileges, all generals have the same privileges), one can group permissions by ***role.***

RBAC systems work as follows:

One creates roles, and assigns permissions to roles.

Each subject is assigned one or more roles. For instance, Alice the surgeon can have both Doctor and Nurse roles so that she can see all the information from her patients.

At each point in time, subjects selects one role and acts with the permissions of that role. This prevents users from misusing privileges (or enforce least privilege). For instance, when Alice is acting as a Nurse, she can read the patient's treatment but she cannot write the patient's treatment. If she changes her role as Doctor,

then she can do both.

RBAC Problems

Problem 1: Role Explosion

- Temptation to create fine grained roles, denying benefits of RBAC

Problem 2: Simple RBAC has limited expressiveness

- Problems with implementing least privilege
- Some roles are relative: “Alice's Doctor” vs. “Any Doctor”

Problem 3: Difficult to implement separation of duty

- “Two doctors are needed to authorize a procedure”
- RBAC Mechanism needs to ensure they are distinct!

Problem 1: In order to accommodate all requirements of a system, and small differences between different subjects, one may be tempted to create new roles all the time. When there are too many roles and they end up being almost one role per each user, then the gain in having roles is lost. We still have a very complex system, almost equivalent to an ACL.

Problem 2: If RBAC keeps its advantage (a small number of roles), then it is not very expressive and may lose nuanced cases. For instance, though all doctors have similar privileges regarding patients, they only should be able to see medical data of the patients they treat.

Problem 3: It makes it difficult to implement the separation of privilege principle. As all subjects assigned to a role are the same to the system, the system does not have the means to see if there are one or two users to enforce the separation of privilege.

Group Based Access Control

Systems have too many subjects! that come and go!

Large dynamic ACLs ☹

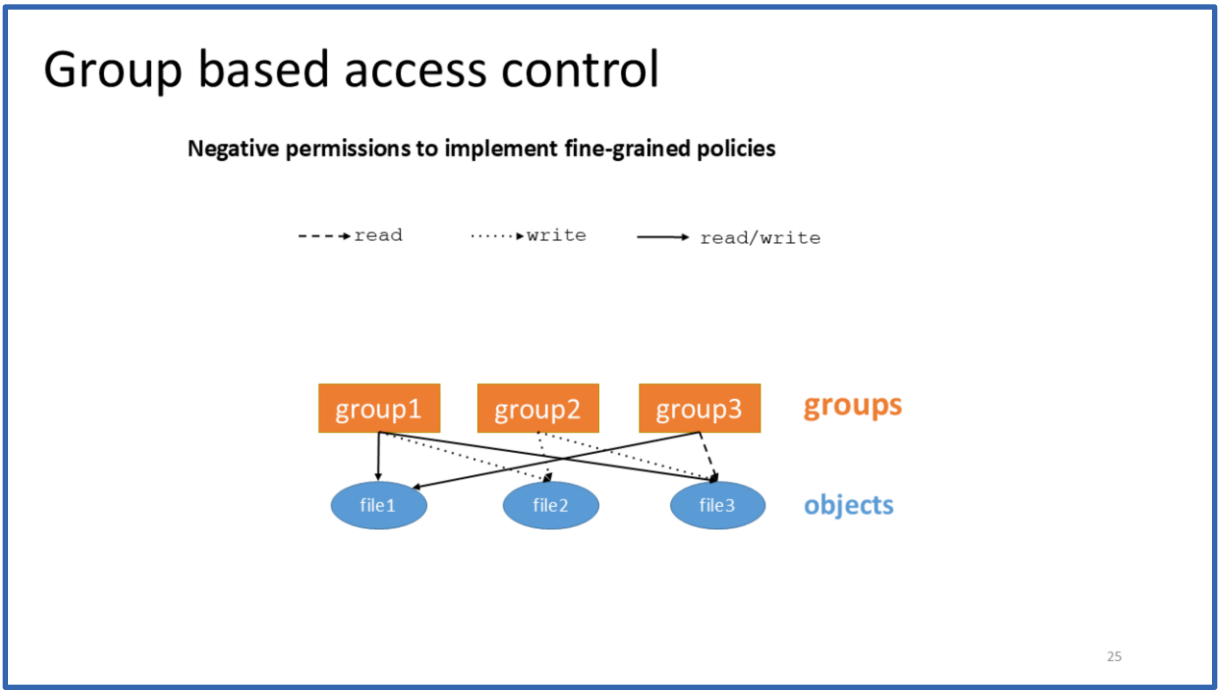
Observation: Some permissions are always needed together

e.g., access to sockets and network interface always go hand in hand

1) assign permissions to access objects to *groups*
2) assign subjects to *groups*
3) subjects have the permissions of *all* their groups

R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-Based Access Control Models. IEEE Computer, 29(2):38--47, 1996

Another option, instead of looking at similarity between users, is to look at of permissions that are often needed together to run the system. These permissions are then put together in so-called **groups**.

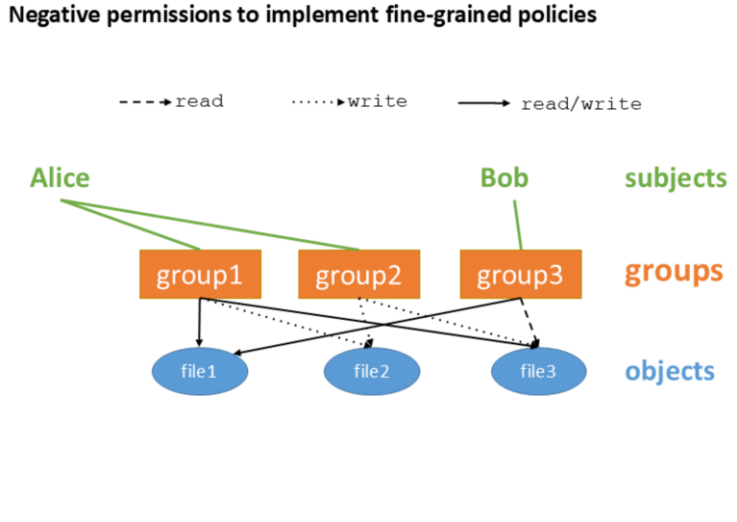


Sometimes it makes sense for a subject to belong to a group, but this subject may not be allowed to access one of the resources in the group by the security policy.

In this case we can implement what are called **negative permissions**, which indicate that a particular subject **does not** have a particular permission on an object. For instance in the example Alice needs access to file2 and file3, so group1 makes sense for her; but she should not read or write on file1. Instead of creating a new group, or breaching the security policy, we can add a negative permission that indicates that

Negative permissions should always be tested first. If there is a negative permission, there is no need to check anything else. It guarantees that there is no error when checking and obtaining some positive permission (fail safe: if something is incorrect, the subject cannot access).

Group based access control



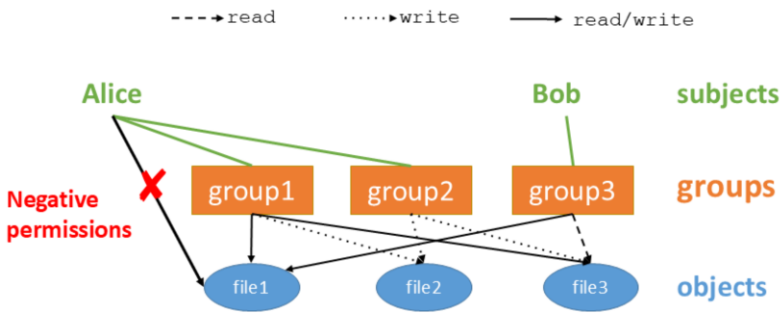
Sometimes it makes sense for a subject to belong to a group, but this subject may not be allowed to access one of the resources in the group by the security policy.

In this case we can implement what are called **negative permissions**, which indicate that a particular subject **does not** have a particular permission on an object. For instance in the example Alice needs access to file2 and file3, so group1 makes sense for her; but she should not read or write on file1. Instead of creating a new group, or breaching the security policy, we can add a negative permission that indicates that

Negative permissions should always be tested first. If there is a negative permission, there is no need to check anything else. It guarantees that there is no error when checking and obtaining some positive permission (fail safe: if something is incorrect, the subject cannot access).

Group based access control

Negative permissions to implement fine-grained policies



What would you check first: Negative permissions or group permissions?

27

Sometimes it makes sense for a subject to belong to a group, but this subject may not be allowed to access one of the resources in the group by the security policy.

In this case we can implement what are called **negative permissions**, which indicate that a particular subject **does not** have a particular permission on an object. For instance in the example Alice needs access to file2 and file3, so group1 makes sense for her; but she should not read or write on file1. Instead of creating a new group, or breaching the security policy, we can add a negative permission that indicates that

Negative permissions should always be tested first. If there is a negative permission, there is no need to check anything else. It guarantees that there is no error when checking and obtaining some positive permission (fail safe: if something is incorrect, the subject cannot access).

Capabilities

Associate permissions to **subjects**

	file1	file2	file3
Alice	read write		read
Bob		read write	read write

Mark S. Miller , Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003

A second option is to store the matrix taking information in rows, i.e., associate the permissions to **subjects**. This is called a subject's **Capabilities**. For each subject, the Capabilities list contains a tuple for each object expressing the permissions for that object of the subject.

For Alice, the Capabilities contains `Alice:` `{(file1,read/write),(file3,read)}` indicating that Alice has read and write permissions on file1, and read permissions on file3.

Note that, as Alice has no permissions on file2, Alice's capabilities *contains **no** information about file2!*

Capabilities have the following advantages. They can be stored with the subject, and thus are "portable" to the subject – this is an advantage, for instance, in distributed systems where not all objects are on the same place. The subject can show her credential to any subsystem or node and prove her permissions. In this case it is easy to know all the permissions associated to a subject. Also, delegating is simpler. As the subject has the capability list it suffices with giving it to other subject to transfer the permission. This of course has also problems (see below)

Capabilities seems to have drawbacks:

- Revoking just one permission is hard. If the objects are unstructured, the search for the

permission is linear with the number of objects of a user.
- Transferrability makes delegation easy, but how to ensure that transfers are only made in legitimate ways? What if Alice transfer to Bob her permission to read file1, which is not in the ACM. This would break the security policy.
- When the permissions reside with the subject that shows them, how do we check authenticity (this is similar to the authenticity problem of concert tickets).

The paper “Capability myths demolished” argues that these limitations are addressable.

Capabilities

Associate permissions to **subjects**

	file1	file2	file3
Alice	read write		read
Bob		read write	read write

Permissions associated to **subjects**

```
Alice:{(file1,read/write),(file3,read
Bob:{(file2,read/write),(file3,write)
```

Mark S. Miller , Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003

29

A second option is to store the matrix taking information in rows, i.e., associate the permissions to **subjects**. This is called a subject's **Capabilities**. For each subject, the Capabilities list contains a tuple for each object expressing the permissions for that object of the subject.

For Alice, the Capabilities contains `Alice:{(file1,read/write),(file3,read)}` indicating that Alice has read and write permissions on file1, and read permissions on file3.

Note that, as Alice has no permissions on file2, Alice's capabilities *contains **no** information about file2!*

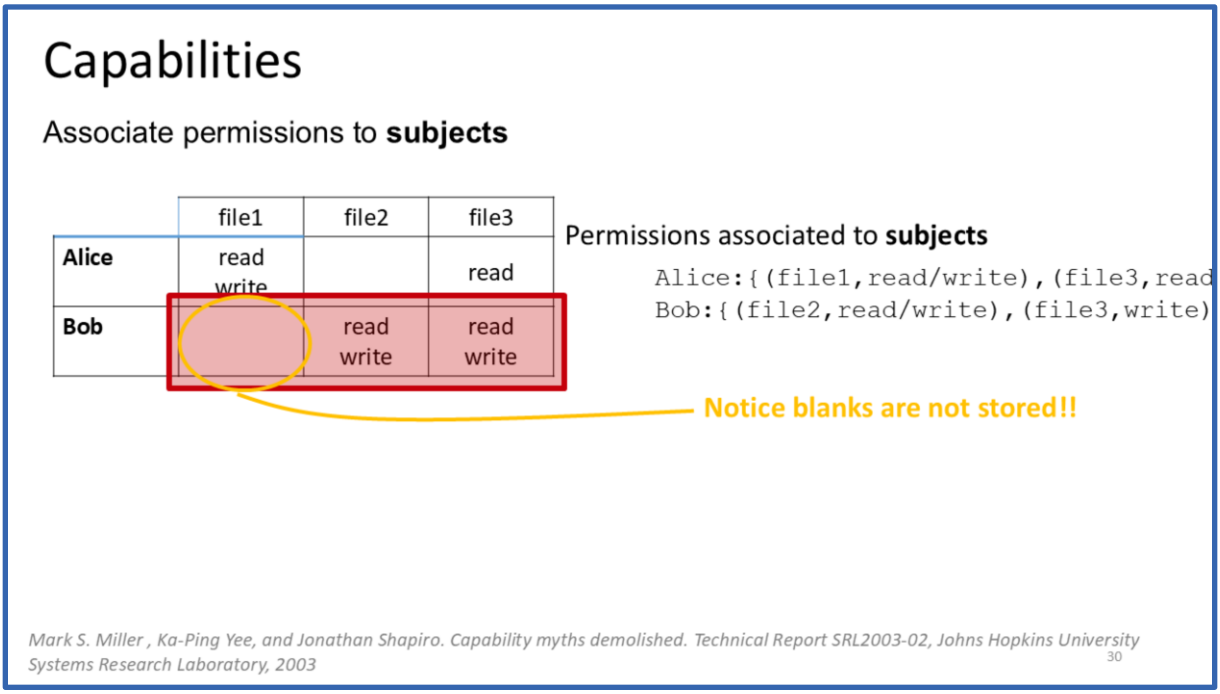
Capabilities have the following advantages. They can be stored with the subject, and thus are "portable" to the subject – this is an advantage, for instance, in distributed systems where not all objects are on the same place. The subject can show her credential to any subsystem or node and prove her permissions. In this case it is easy to know all the permissions associated to a subject. Also, delegating is simpler. As the subject has the capability list it suffices with giving it to other subject to transfer the permission. This of course has also problems (see below)

Capabilities seems to have drawbacks:

- Revoking just one permission is hard. If the objects are unstructured, the search for the

permission is linear with the number of objects of a user.
- Transferrability makes delegation easy, but how to ensure that transfers are only made in legitimate ways? What if Alice transfer to Bob her permission to read file1, which is not in the ACM. This would break the security policy.
- When the permissions reside with the subject that shows them, how do we check authenticity (this is similar to the authenticity problem of concert tickets).

The paper “Capability myths demolished” argues that these limitations are addressable.



A second option is to store the matrix taking information in rows, i.e., associate the permissions to **subjects**. This is called a subject's **Capabilities**. For each subject, the Capabilities list contains a tuple for each object expressing the permissions for that object of the subject.

For Alice, the Capabilities contains `Alice: {(file1,read/write),(file3,read)}` indicating that Alice has read and write permissions on file1, and read permissions on file3.

Note that, as Alice has no permissions on file2, Alice's capabilities *contains **no** information about file2!*

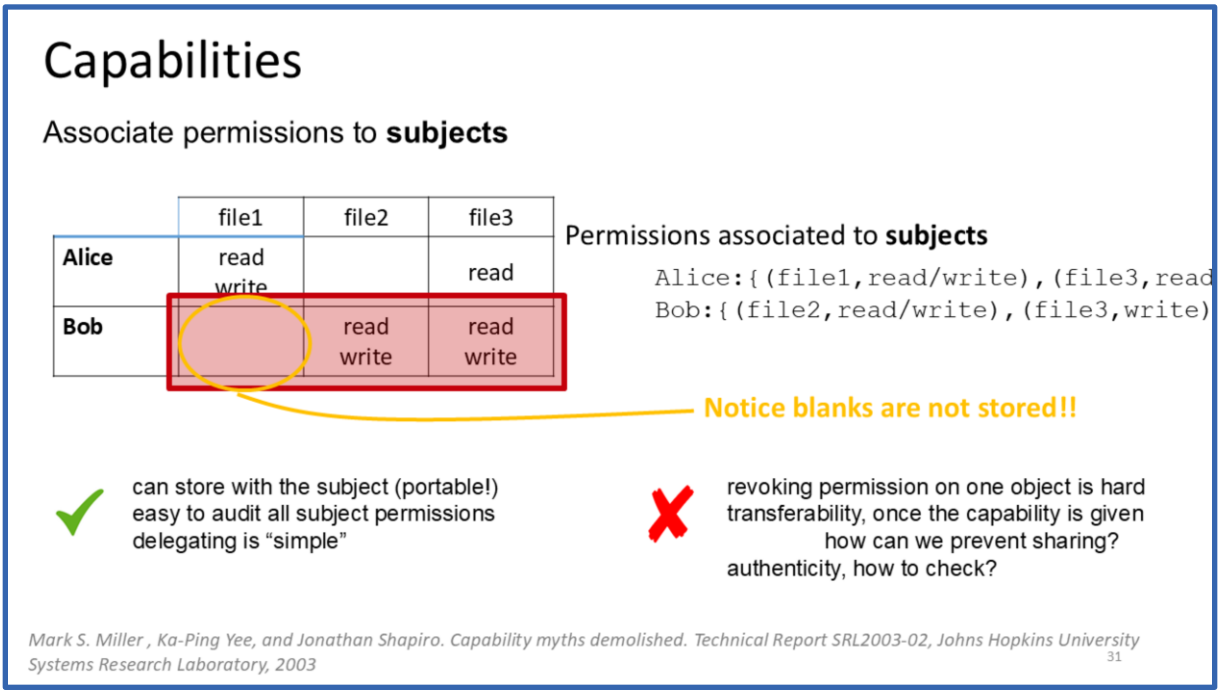
Capabilities have the following advantages. They can be stored with the subject, and thus are "portable" to the subject – this is an advantage, for instance, in distributed systems where not all objects are on the same place. The subject can show her credential to any subsystem or node and prove her permissions. In this case it is easy to know all the permissions associated to a subject. Also, delegating is simpler. As the subject has the capability list it suffices with giving it to other subject to transfer the permission. This of course has also problems (see below)

Capabilities seems to have drawbacks:

- Revoking just one permission is hard. If the objects are unstructured, the search for the

permission is linear with the number of objects of a user.
- Transferrability makes delegation easy, but how to ensure that transfers are only made in legitimate ways? What if Alice transfer to Bob her permission to read file1, which is not in the ACM. This would break the security policy.
- When the permissions reside with the subject that shows them, how do we check authenticity (this is similar to the authenticity problem of concert tickets).

The paper “Capability myths demolished” argues that these limitations are addressable.



A second option is to store the matrix taking information in rows, i.e., associate the permissions to **subjects**. This is called a subject's **Capabilities**. For each subject, the Capabilities list contains a tuple for each object expressing the permissions for that object of the subject.

For Alice, the Capabilities contains `Alice: {(file1,read/write),(file3,read)}` indicating that Alice has read and write permissions on file1, and read permissions on file3.

Note that, as Alice has no permissions on file2, Alice's capabilities *contains **no** information about file2!*

Capabilities have the following advantages. They can be stored with the subject, and thus are "portable" to the subject – this is an advantage, for instance, in distributed systems where not all objects are on the same place. The subject can show her credential to any subsystem or node and prove her permissions. In this case it is easy to know all the permissions associated to a subject. Also, delegating is simpler. As the subject has the capability list it suffices with giving it to other subject to transfer the permission. This of course has also problems (see below)

Capabilities seems to have drawbacks:

- Revoking just one permission is hard. If the objects are unstructured, the search for the

permission is linear with the number of objects of a user.
- Transferrability makes delegation easy, but how to ensure that transfers are only made in legitimate ways? What if Alice transfer to Bob her permission to read file1, which is not in the ACM. This would break the security policy.
- When the permissions reside with the subject that shows them, how do we check authenticity (this is similar to the authenticity problem of concert tickets).

The paper “Capability myths demolished” argues that these limitations are addressable.

A recurrent problem in access control

AMBIENT AUTHORITY is used by a subject if for an action to succeed it **only** needs to specify the **operation** and the **names** of the involved object(s)

In these cases the **subject** (with authority) is implicit

```
open("file1", "rw")
```

(the subject is missing, it is understood it is the process owner)

32

The concept of **Ambient Authority** refers to the situations in which when performing an action in a system, we *do not* specify the subject, but only the operation and the object on which it is performed. The subject, which is needed to decide whether the operation on the object is permitted or not is *implicit*, usually from environmental conditions. For instance, when we writing a file, one does no specify which user is opening the file, just "open", "file", and for what operations "rw". When the program is run by a user, it has the permission of this user. *The user in this case is the Ambient Authority.*

The use of ambient authority simplifies program design and usability, by not having to associate a user to the instructions and not having to repeat it for every instruction.

However, the lack of subject is troublesome. Since sometimes the authority is not clear, it is hard to enforce least privilege.

A recurrent problem in access control

AMBIENT AUTHORITY is used by a subject if for an action to succeed it **only** needs to specify the **operation** and the **names** of the involved object(s)

In these cases the **subject** (with authority) is implicit

```
open("file1", "rw")
```

The program cannot check permissions!

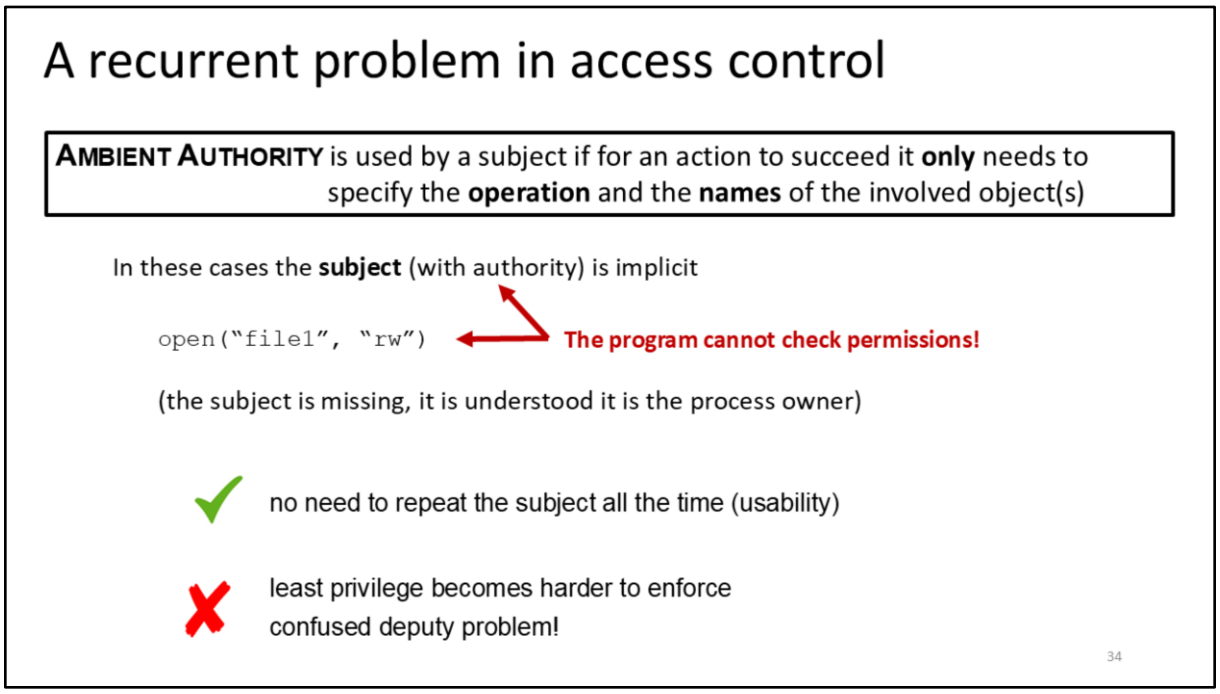
(the subject is missing, it is understood it is the process owner)

33

The concept of **Ambient Authority** refers to the situations in which when performing an action in a system, we *do not* specify the subject, but only the operation and the object on which it is performed. The subject, which is needed to decide whether the operation on the object is permitted or not is *implicit*, usually from environmental conditions. For instance, when we writing a file, one does no specify which user is opening the file, just "open", "file", and for what operations "rw". When the program is run by a user, it has the permission of this user. *The user in this case is the Ambient Authority.*

The use of ambient authority simplifies program design and usability, by not having to associate a user to the instructions and not having to repeat it for every instruction.

However, the lack of subject is troublesome. Since sometimes the authority is not clear, it is hard to enforce least privilege.



The concept of **Ambient Authority** refers to the situations in which when performing an action in a system, we *do not* specify the subject, but only the operation and the object on which it is performed. The subject, which is needed to decide whether the operation on the object is permitted or not is *implicit*, usually from environmental conditions. For instance, when we writing a file, one does no specify which user is opening the file, just "open", "file", and for what operations "rw". When the program is run by a user, it has the permission of this user. *The user in this case is the Ambient Authority.*

The use of ambient authority simplifies program design and usability, by not having to associate a user to the instructions and not having to repeat it for every instruction.

However, the lack of subject is troublesome. Since sometimes the authority is not clear, it is hard to enforce least privilege.

The confused deputy problem

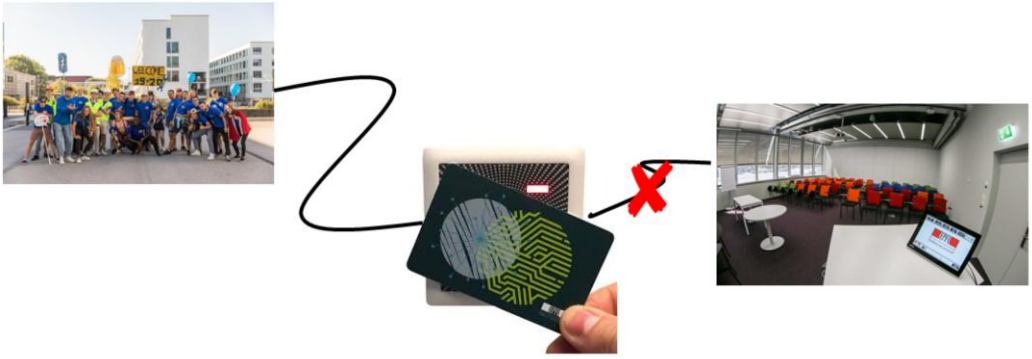
Problem with ambient authority:
A privileged program can be tricked to misuse its authority
(Confused deputy problem)

ACL generally considers ambient authority, since permissions are usually checked for the user running the program (the ambient authority)

In **Capabilities** the capability itself contains the identity of the principal. Thus, there is no ambient authority.

A **confused deputy** is a privileged principal that is used by another less privileged principal to improve her access to the system

The confused deputy problem



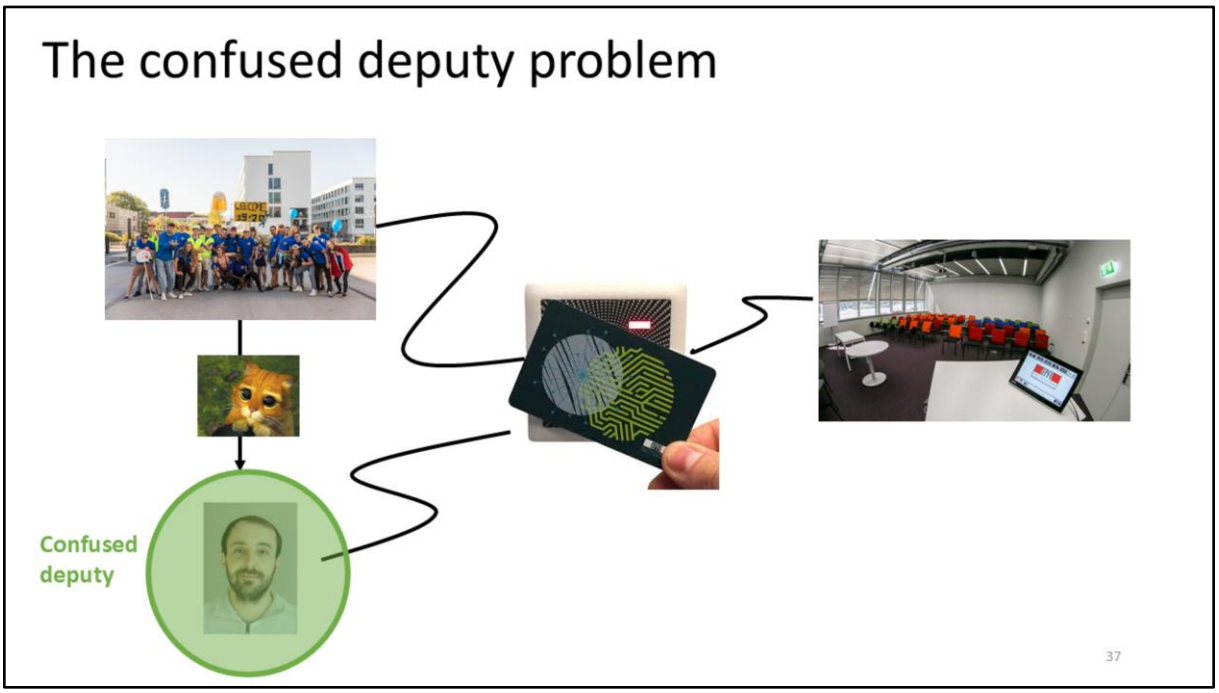
36

A group of students want to enter a classroom but their CAMIPRO cards do not have access

They can try to break the lock, or break EPFL central services to get access. These are difficult.

Alternatively, they can try to get a new professor that does not know very well who has access to what to open the door with their very privileged badge.

In this case the professor is the **Confused Deputy**

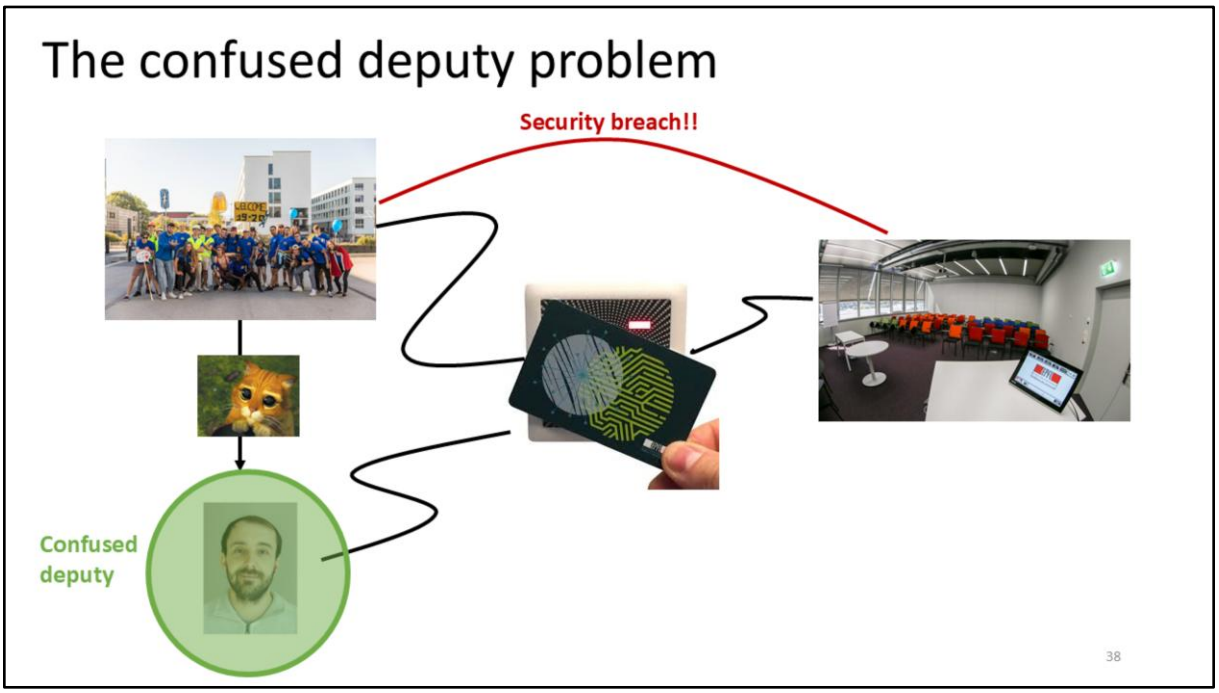


A group of students want to enter a classroom but their CAMIPRO cards do not have access

They can try to break the lock, or break EPFL central services to get access. These are difficult.

Alternatively, they can try to get a new professor that does not know very well who has access to what to open the door with their very privileged badge.

In this case the professor is the **Confused Deputy**



A group of students want to enter a classroom but their CAMIPRO cards do not have access

They can try to break the lock, or break EPFL central services to get access. These are difficult.

Alternatively, they can try to get a new professor that does not know very well who has access to what to open the door with their very privileged badge.

In this case the professor is the **Confused Deputy**

The confused deputy

PAY-PER-USE COMPILER

- Compiler receives (`input`, `output`)
- Compiles the program `input` and:
  writes usage into `bill`
  writes errors into `output`

	input	output	bill
Alice	write	read	read
Compiler	read	read write	read write

ACL

```
input: {(Alice, write), (Compiler, read)}
output: {(Alice, read), (Compiler, read/write)}
bill: {(Alice, read), (Compiler, read/write)}
```

CAN ALICE CHANGE THE `bill`?

AND AVOID PAYING?

39

In a computer environment.

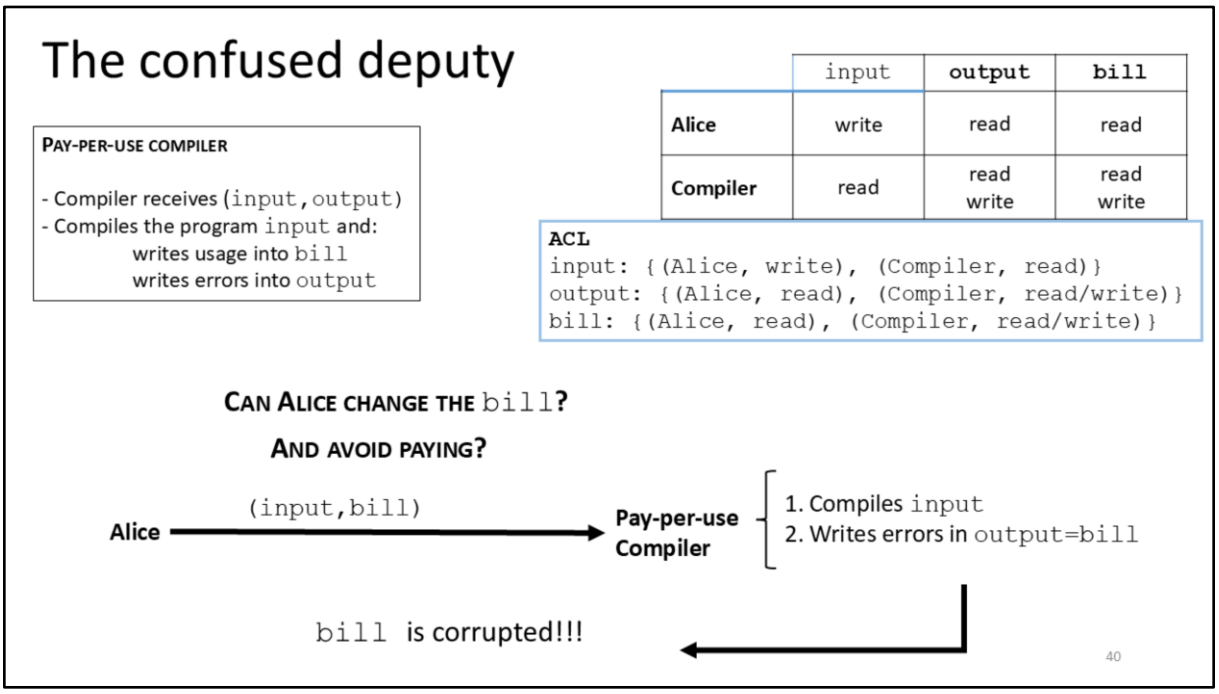
Assume a compiler as a service, where users pay per use. The compiler works as follows:
1) it receives two files:
`input`: the file to be compiled
`output`: a file where compiling errors are written to help the user to fix programming errors
2) and then it:
compiles the file `input`
writes a record of the compile process in a file called `bill` for billing purposes
writes errors in the file `output`

Alice wants to compile without paying. Given the access control matrix, however, she cannot change `bill` (she only has read permission). But what she can do is to try to corrupt `bill` so that no billing can be done.

For this, she uses the fact that the compiler does have write access on `bill`. So, instead of giving an empty file to collect the errors, she provides `bill`. The compiler does its job and while compiling overwrites `bill`. Once `bill` is corrupted, Alice cannot be billed

anymore as there is no record of her usage of the compiler.

In this scenario, the compiler acts as confused deputy. Its authority to write on `bill` is abused by Alice to avoid paying.



In a computer environment.

Assume a compiler as a service, where users pay per use. The compiler works as follows:
1) it receives two files:
`input`: the file to be compiled
`output`: a file where compiling errors are written to help the user to fix programming errors
2) and then it:
compiles the file `input`
writes a record of the compile process in a file called `bill` for billing purposes
writes errors in the file `output`

Alice wants to compile without paying. Given the access control matrix, however, she cannot change `bill` (she only has read permission). But what she can do is to try to corrupt `bill` so that no billing can be done.

For this, she uses the fact that the compiler does have write access on `bill`. So, instead of giving an empty file to collect the errors, she provides `bill`. The compiler does its job and while compiling overwrites `bill`. Once `bill` is corrupted, Alice cannot be billed

anymore as there is no record of her usage of the compiler.

In this scenario, the compiler acts as confused deputy. Its authority to write on `bill` is abused by Alice to avoid paying.

How to avoid confused deputies

Real problem. Ambient authority is used for convenience in many real systems, OS services, web servers,…

Solutions:

1) Re-implement access control in the privileged process
2) Let privileged process check authorization for Alice.
3) Capabilities can help!

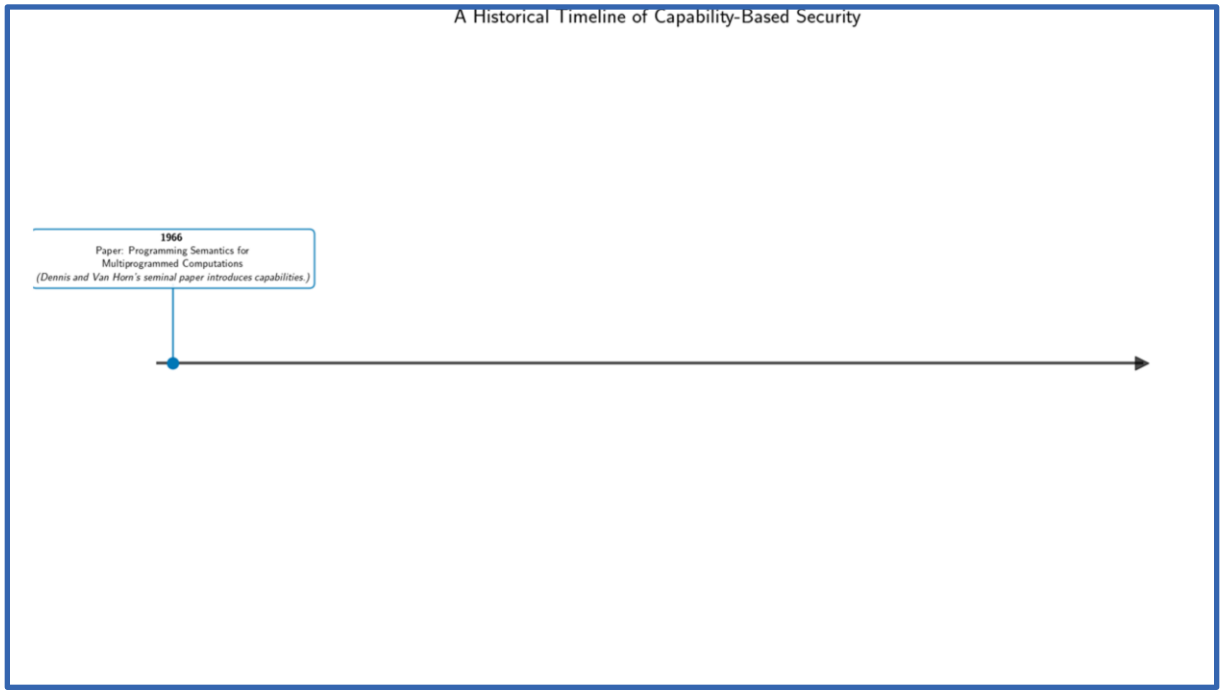
41

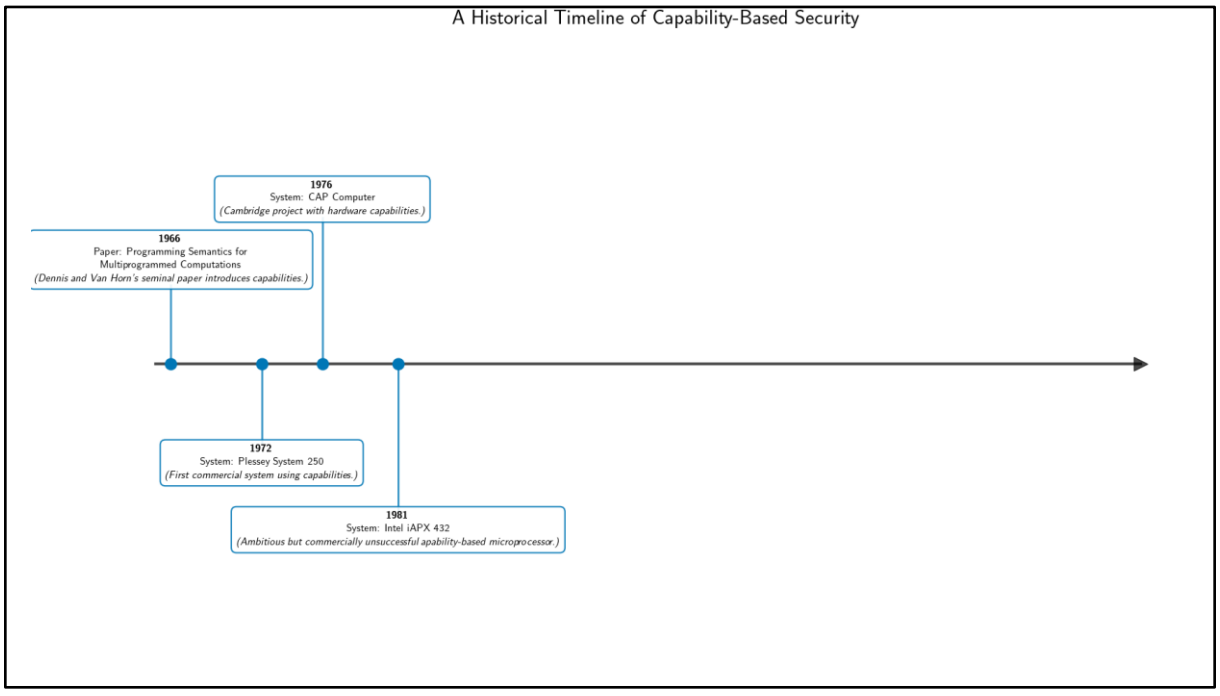
The confused deputy, as we will see along the course, is a real problem any time there is ambient authority (and this happens in many real servers).

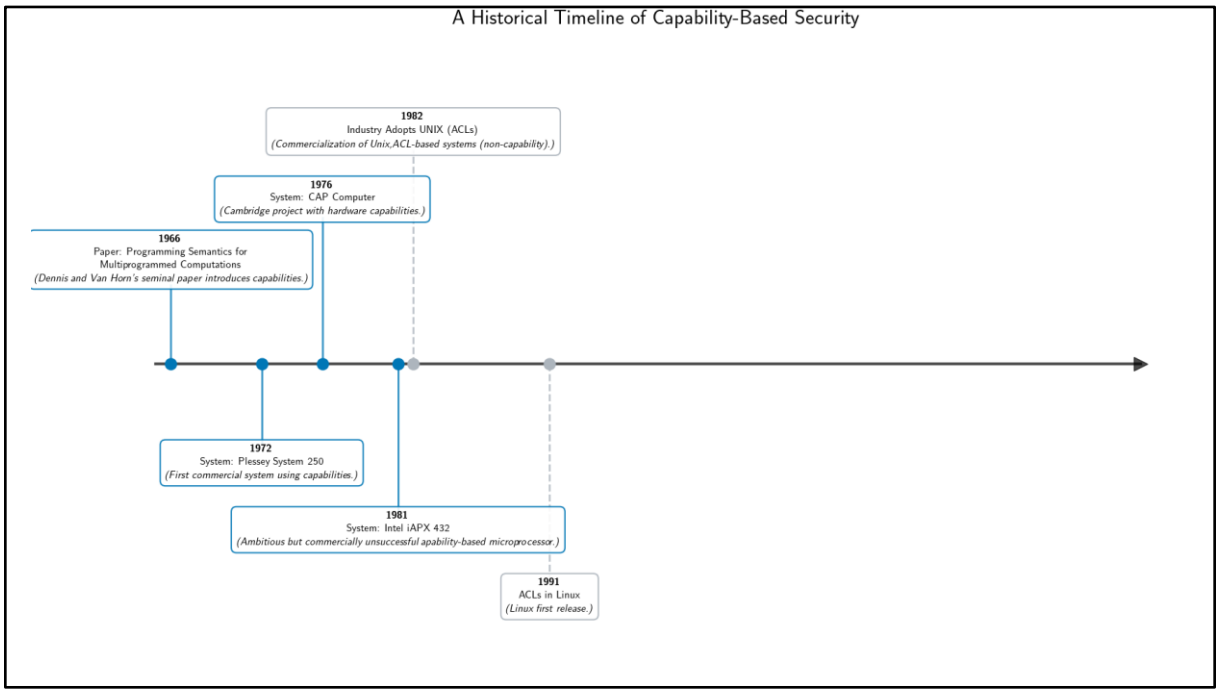
How to avoid it:

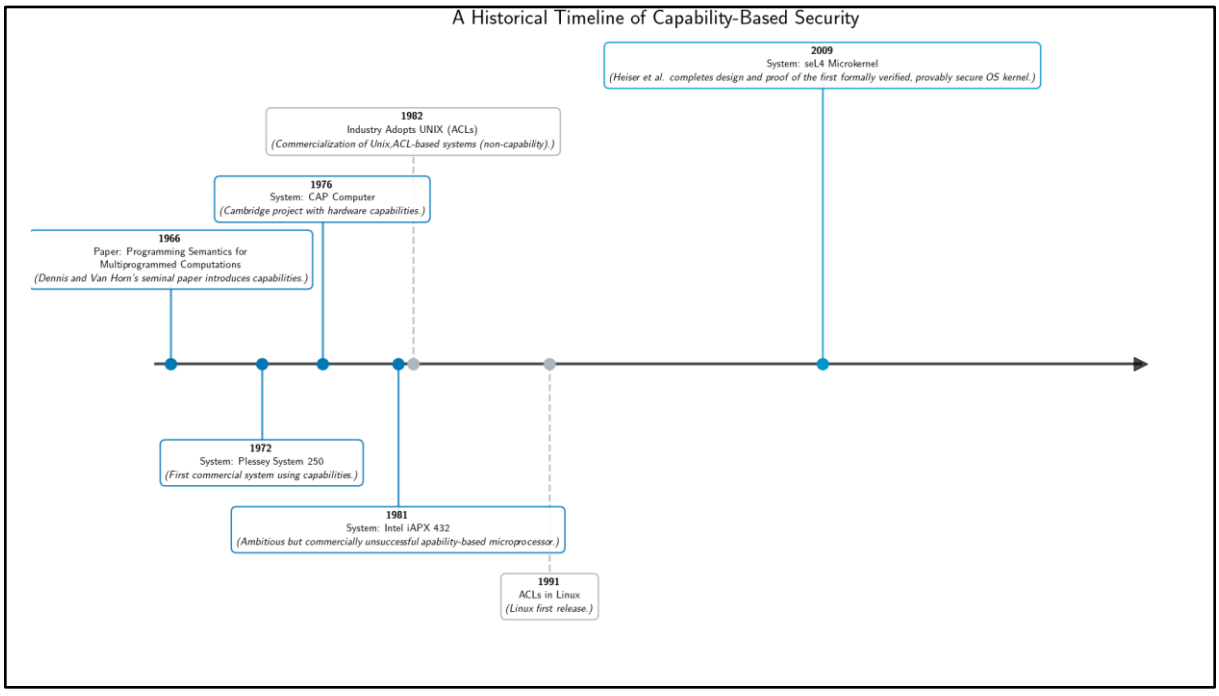
1) Have access control inside the privileged process: avoid the ambient authority. This is very expensive and cumbersome., thus prone to errors.
2) Give the privileged process the means to check Alice's permissions. There is ambient authority (the process) but this authority can check whether the user invoking the authority has the same permissions.
3) Use capabilities. When using capabilities instead of an ACL, there is no ambient authority anymore. The permissions are explicit in the capability.

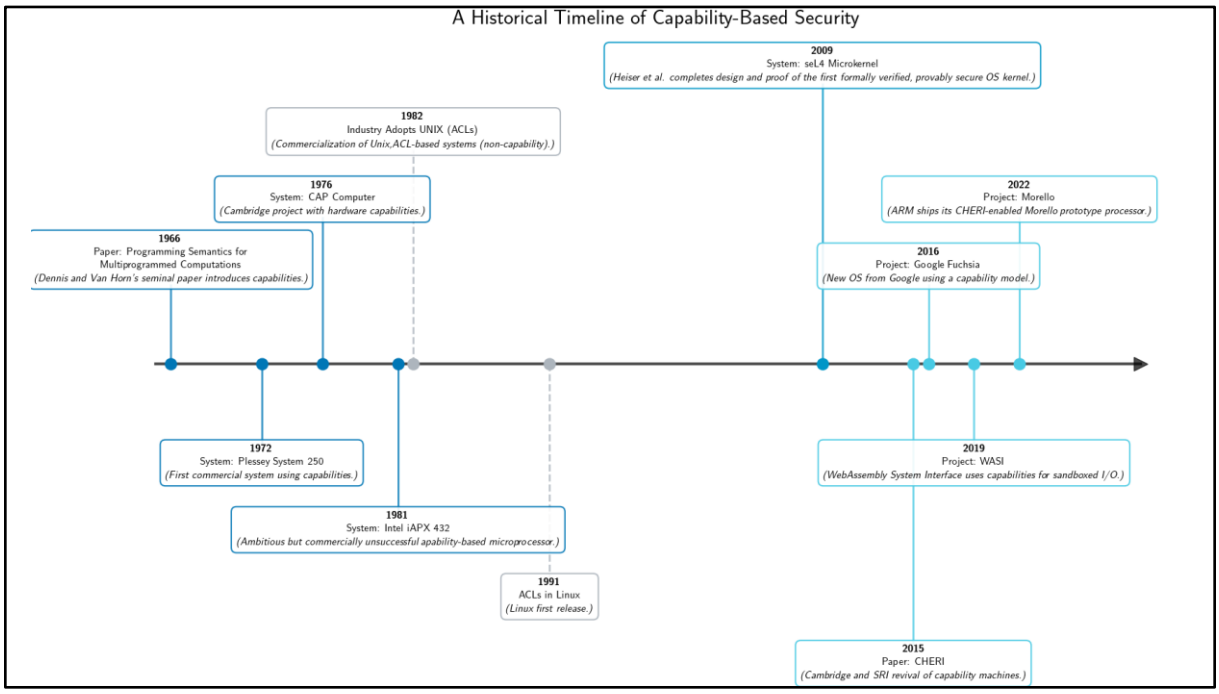
In the example, with capabilities Alice would show her capabilities to the compiler to show that she can write in the debugging file `output`. As she does not have capabilities for `bill`, she cannot use it as output file!


A Historical Timeline of Capability-Based Security
1966
Paper: Programming Semantics for
Multiprogrammed Computations
(Dennis and Van Horn's seminal paper introduces capabilities.)


A Historical Timeline of Capability-Based Security
1976
System: CAP Computer
(Cambridge project with hardware capabilities.)
1966
Paper: Programming Semantics for
Multiprogrammed Computations
(Dennis and Van Horn's seminal paper introduces capabilities.)
1972
System: Plessey System 250
(First commercial system using capabilities.)
1981
System: Intel iAPX 432
(Ambitious but commercially unsuccessful apability-based microprocessor.)


A Historical Timeline of Capability-Based Security
1982
Industry Adopts UNIX (ACLs)
(Commercialization of Unix,ACL-based systems (non-capability).)
1976
System: CAP Computer
(Cambridge project with hardware capabilities.)
1966
Paper: Programming Semantics for
Multiprogrammed Computations
(Dennis and Van Horn's seminal paper introduces capabilities.)
1972
System: Plessey System 250
(First commercial system using capabilities.)
1981
System: Intel iAPX 432
(Ambitious but commercially unsuccessful apability-based microprocessor.)
1991
ACLs in Linux
(Linux first release.)


A Historical Timeline of Capability-Based Security
2009
System: seL4 Microkernel
(Heiser et al. completes design and proof of the first formally verified, provably secure OS kernel.)
1982
Industry Adopts UNIX (ACLs)
(Commercialization of Unix,ACL-based systems (non-capability).)
1976
System: CAP Computer
(Cambridge project with hardware capabilities.)
1966
Paper: Programming Semantics for Multiprogrammed Computations
(Dennis and Van Horn's seminal paper introduces capabilities.)
1972
System: Plessey System 250
(First commercial system using capabilities.)
1981
System: Intel iAPX 432
(Ambitious but commercially unsuccessful apability-based microprocessor.)
1991
ACLs in Linux
(Linux first release.)


A Historical Timeline of Capability-Based Security
2009
System: seL4 Microkernel
(Heiser et al. completes design and proof of the first formally verified, provably secure OS kernel.)
1982
Industry Adopts UNIX (ACLs)
(Commercialization of Unix,ACL-based systems (non-capability).)
1976
System: CAP Computer
(Cambridge project with hardware capabilities.)
2022
Project: Morello
(ARM ships its CHERI-enabled Morello prototype processor.)
1966
Paper: Programming Semantics for Multiprogrammed Computations
(Dennis and Van Horn's seminal paper introduces capabilities.)
2016
Project: Google Fuchsia
(New OS from Google using a capability model.)
1972
System: Plessey System 250
(First commercial system using capabilities.)
2019
Project: WASI
(WebAssembly System Interface uses capabilities for sandboxed I/O.)
1981
System: Intel iAPX 432
(Ambitious but commercially unsuccessful apability-based microprocessor.)
1991
ACLs in Linux
(Linux first release.)
2015
Paper: CHERI
(Cambridge and SRI revival of capability machines.)

How to avoid confused deputies

Real problem. Ambient authority is used for convenience in many real systems, OS services, web servers,…

Solutions:

1) Re-implement access control in the privileged process
2) Let privileged process check authorization for Alice.
3) Capabilities can help!

In the previous example…
- Compiler has capabilities to the file `bill`.
- To compile Alice must give access to the debugging file `output`
  - Cannot give a capability for writing on `bill`!
  - Cannot confuse anyone!

47

The confused deputy, as we will see along the course, is a real problem any time there is ambient authority (and this happens in many real servers).

How to avoid it:

1) Have access control inside the privileged process: avoid the ambient authority. This is very expensive and cumbersome., thus prone to errors.
2) Give the privileged process the means to check Alice's permissions. There is ambient authority (the process) but this authority can check whether the user invoking the authority has the same permissions.
3) Use capabilities. When using capabilities instead of an ACL, there is no ambient authority anymore. The permissions are explicit in the capability.

In the example, with capabilities Alice would show her capabilities to the compiler to show that she can write in the debugging file `output`. As she does not have capabilities for `bill`, she cannot use it as output file!


A Historical Timeline of Capability-Based Security
Keynote: Where Are We With Scala's Capabilities?
The object capability model has been established since the 1960s. It is probably the most obvious and clean way to protect trusted from untrusted components in a complex system. Capabilities are a unifying concept for many aspects of programming, including permissions, effects, and resources. They can be the missing link that can make combinations of functional and imperative programming safe.
So why are object capabilities not used everywhere? I argue it's because they currently lack in both convenience and safety: Convenience: Passing all capabilities along long call chains to code that needs them can quickly get tedious. Safety: Access restrictions such as limited lifetimes or sharing are traditionally encoded using runtime mechanisms with the possibility of runtime failures.
At EPFL we have been working on overcoming these two impediments. Convenience: capabilities can be passed as implicit parameters in using clauses, and capability passing can be completely abstracted over using context functions. Safety: We have extended the type system to track capabilities in types. Specifically, we track which capabilities are closed over in a lambda or object. We are now two years into a project to make these ideas usable on a large scale. I will report on the state of capability checking today: the usage experience with these concepts, what measures we took to make the notations more ergonomic, and what our plans for the future are.
1966
Paper: Programmin
Multiprogrammed
(Dennis and Van Horn's seminal
(Cambridge and SRI revival of capability machines.)

Summary

Discretionary Access Control: owners establish permissions

Conceptualized as an Access Control Matrix

Implemented in two flavors:

- Access control list: permission associated to objects
- Capabilities: permission associated to owners

When relying on access control, it is always important to think about confused deputies

Computer Security and Privacy

Discretionary Access Control
Examples: Linux & Windows

Discretionary Access Control in Real life

We saw that many of the systems we use nowadays rely on discretionary access control:

- Social networks
- Cloud file sharing systems
- **Operative systems**

Many of the systems we interact with in our daily lives rely on discretionary access control to support their security policy.

In social networks, such as Facebook or Instagram, the user (owner) of the content (resources such as posts, photos, videos) decides which other users have access to this content.
In cloud file sharing systems, such as Google Drive or Dropbox, it is the same. The user (owner) of the files (resources) decides with whom to share this content.

The same holds for Operative Systems (OSs). In OSs users have accounts, and files are owned by those who create them. The owners are the ones that decide which other users/accounts can have access to the file. We will now see how the most used OSs, Linux and Windows, do this.

Unix: Principals & Groups

- User Identities (UIDs) and Group Identities (GIDs)
  - Originally 16-bit (now 32-bit) numbers.
  - Special UIDs: -2, 0, 1, …
- User Information
  - Each user has own directory `/home/username`
  - User accounts: `/etc/passwd`
    `username:password:UID:GID:info:home:shell`
- Users belong to one or more groups
  - Primary group (`/etc/passwd`), other groups (`/etc/group`)

In UNIX systems principals are users. Each user has an identity UID. There are some reserved UIDs, which we will see in the following slides.
Users belong to groups, with identity GID.

User accounts are defined in a file /etc/passwd. Each line defines a user as
`username:password:UID:GID:info:home:shell`
`info` is a comment field that can contain some information about the user;
`home`, the absolute path to the directory where the user will appear when they log in;
and `shell` the absolute path to the default shell of the user

If users belong to more groups, it will be marked in the file `/etc/group. There is one line per group, and for each group we have the ID and the list of users belonging to the group.` As in any group-based access control, each group the user belongs to provides new permissions to the user.

Security Architecture

- Everything is a file

- Each user “owns” a set of files

- Each file as a simple **A**ccess **C**ontrol **L**ist to express the access control policy to the file
  - The owner can **modify the permissions of the file** (but cannot give away ownership)
  - System files are owned by special users that can make system operations

- All user processes run by a user run with that user’s privileges

  Ambient authority!!

In UNIX, everything is a file.

Files are created, and owned, by users (they can be created by the root user).

This file has an access control list that says who has access to the file and what operations can they perform. System files are owned by special users which are the only ones with privileges to operate on system files.

All processes run by a user have the privileges of that user (this is a very practical case of the use of Ambient authority)

File Access Control Lists

- Files have **ACLs** attached to them
  - Each file is assigned an **owner UID** and **GID**
  - Each file has 9 permission bits
    - 3 actions: Read, write, execute
    - 3 subjects *owner, group, other*

- Different semantics between files and directories
  - *Directories*: Read → List files, Write → Add files, Exec → traverse the directory, "cd"

- 3 attributes: "`suid`", "`sgid`", and "`sticky`"

Permission bits (see next slide) provide permissions for the 3 groups in UNIX: the user, the user's group ,others.
The three permissions are **read**, **write** (modify or delete), **execute**

Each user owns their files and has access to them. UNIX has a very simple way of defining who else has access by defining three groups:

owner: the group is formed just by the file owner
group: the file's group
other: anyone that is not the owner or on one of the owner's group(s)

When the file represents a **directory** the permissions change semantics:

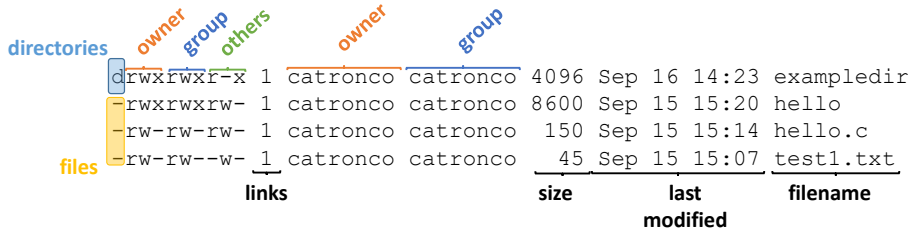
- Read -> the user has the right to list the files inside the directory, i.e., executing the command "ls"
- Write -> the user has the right to create a file in the directory (by creating a new file, moving a file, or renaming a file)
- Execute -> the user has the right to "move into" the directory, i.e., to execute the command "cd"

Note: in practice, in Unix systems, one cannot read and write the files on the

directory unless the execution permission (x) is also active

Besides the 9 permission bits, there can be three attributes explained in slides below.

Example of UNIX ACLs



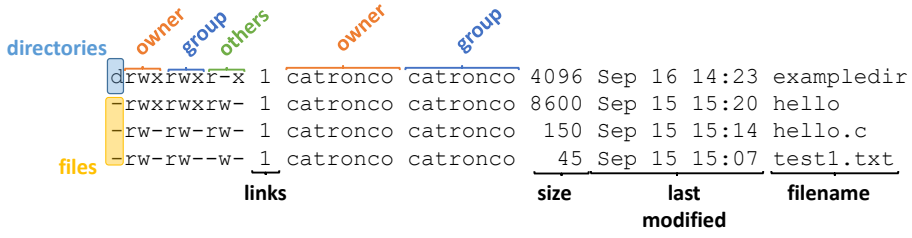
The d at the beginning stands for directory. When there is no letter it is a file.

The next 9 bits are the permission read (r), write (w), execute (x), for owner, file's group, and others.

links – number of symbolic links to the file

The owner can change permissions on files with the command **chmod**. This command receives as parameter the new permissions as either the letter representing the permission (r,w,x) or attribute (s,t); or as a binary that expresses the '1' bits.

Example of UNIX ACLs



Owner can change permissions on files

chmod [+r, -w, / 666, 662 / +t or **1**666, +s or **4**666] filename

The d at the beginning stands for directory. When there is no letter it is a file.

The next 9 bits are the permission read (r), write (w), execute (x), for owner, file's group, and others.

links – number of symbolic links to the file

The owner can change permissions on files with the command **chmod**. This command receives as parameter the new permissions as either the letter representing the permission (r,w,x) or attribute (s,t); or as a binary that expresses the '1' bits.

UNIX Access control in action

Compare:

`UID` / `GID` of process trying to perform action

with:

state of file (Owner, Group, mode bits)

Order matters in the comparison

1. If `UID` says you are owner: check bits for owner.
2. If not owner, but your group is owner, check `GID` with bits for group.
3. Otherwise check bits for "other"

`root` user is never denied access

To decide whether an action can be performed, UNIX compares the UID/GID of the process trying to perform the action (that of the user, remember ambient authority) with the information appearing in the access control list of the file:

```
-rwxrwxrw- 1 catronco catronco 8600 Sep 15 15:20 hello
```

Depending of what your UID/GID is, it will use the permissions of owner, group, or other

If the UID is the one of the root user, the permission will always be granted

Super users

Special "`root`" user account

- User ID 0
- Access system files and special operations
- Can access anything: (almost all) security checks disabled
- `root` is in the **TCB**!!

The diference between **sudo** and **su** is that

sudo – executes *one* action as super user

su – changes the current user to another user. If there is no argument, it changes to root, the super user ← doing this is very dangerous, as any action you would realize would not undergo security checks

To allow users to access systems files and services, there is the suid mechanism that we will see in a couple of slides

Super users

Never login as root!
- Some distributions assign no password
- Use "`sudo`" or "`su`" command
- Difference?
  (`$ sudo su catronco`)

Special "`root`" user account
- User ID 0
- Access system files and special operations
- Can access anything: (almost all) security checks disabled
- `root` is in the **TCB**!!

59

The diference between **sudo** and **su** is that

sudo – executes *one* action as super user

su – changes the current user to another user. If there is no argument, it changes to root, the super user ← doing this is very dangerous, as any action you would realize would not undergo security checks

To allow users to access systems files and services, there is the suid mechanism that we will see in a couple of slides.

Sudo su catronco: this command means: "As my current user, use my sudo rights to become the user catronco without needing to know catronco's password." You are leveraging your own administrative power to impersonate another user on the system.

Super users

Never login as root!
- Some distributions assign no password
- Use "`sudo`" or "`su`" command
- Difference?

(`$ sudo su catronco`)

Special "`root`" user account
- User ID 0
- Access system files and special operations
- Can access anything: (almost all) security checks disabled
- `root` is in the **TCB**!!

Normal users also need to access system services
but these services need to run with system privileges

`suid` / `sgid` mechanism

60

The diference between **sudo** and **su** is that

sudo – executes *one* action as super user

su – changes the current user to another user. If there is no argument, it changes to root, the super user ← doing this is very dangerous, as any action you would realize would not undergo security checks

To allow users to access systems files and services, there is the suid mechanism that we will see in a couple of slides

Sudo su catronco: this command means: "As my current user, use my sudo rights to become the user catronco without needing to know catronco's password." You are leveraging your own administrative power to impersonate another user on the system.

Exercise – ACL in Linux

A directory named **/project_beta** has the following permissions:

```
drwxr-x--- 2 manager_A core_devs 4096 Sep 24 11:35 /project_beta
```

- **manager_A**: The directory owner and a member of the core_devs group, is **sudo**.
- **developer_B**: A member of the core_devs group.
- **intern_C**: Is **not** a member of the core_devs group.

For each user, determine which of the following commands will succeed and which will fail. Explain why based on the Discretionary Access Control rules.

manager_A	`cd /project_beta`	`ls /project_beta`	`touch /project_beta/status.log`
developer_B	`cd /project_beta`	`ls /project_beta`	`touch /project_beta/notes.txt`
intern_C	`cd /project_beta`	`ls /project_beta`	`touch /project_beta/ideas.txt`

61

Directory Details:
Owner: user_A
Group: project_leads
Permissions: The owner (user_A) can read, write, and execute (rwx). Members of the project_leads group can read and execute (r-x). Other users have no permissions (---)

manager_A: All commands succeed. The system checks the manager's permissions (rwx). manager_A has execute (x) permission to cd into the directory, read (r) permission to ls its contents, and write (w) permission to touch (create) a new file.
developer_B (Group Member): Commands 1 & 2 succeed, Command 3 fails.
developer_B is not the owner, so the system checks group permissions (r-x), and developer is a group membe,. The groups has execute (
x) permission to enter and read (r) permission to list files. However, the group lacks write (w) permission, so the touch command fails.
intern_C ("Other"): All commands fail. intern_C is not the owner and is not in the core_devs group, so the "other" permissions (---) apply. Since no permissions are granted, all attempts to interact with the directory are denied.

Exercise – ACL in Linux

The system state is the same as before, but with a **new subdirectory inside**:

```
drwxr-xr-x 2 developer_B core_devs 4096 Sep 24 12:00 /project_beta/Foo
```

And inside this directory, a file:

```
-rw-r--r-- 1 developer_B core_devs 512 Sep 24 12:05 /project_beta/Foo/intern_project.txt
```

Q1: The parent directory `/project_beta` is owned by `manager_A`, but the subdirectory `Foo` is owned by `developer_B`. Find a scenario (sequence of commands) that would have led to this situation.

Q2: `intern_C` wants to read the project file. They try the following commands in sequence. Do they work?

```
cd /project_beta/Foo
cat intern_project.txt
```

Q3: `intern_C` tries again, this time using the file's full path from their home directory. Does this command work?

```
cat /project_beta/Foo/intern_project.txt
```

Q4: Who do they need to contact to fix the problem?

62

Q1: This is a normal, though deliberate, situation. A user cannot simply create a directory and give it to someone else. This ownership structure was likely created by a superuser (like
root , or manager_A using sudo). The administrator would have created the directory and then explicitly transferred ownership to developer_B using the chown command:

sudo chown developer_B /project_beta/Foo

This is a common way to delegate responsibility for a specific part of a project without giving away control of the parent directory.

Q2: The two-command sequence fails at the very first step.
cd /project_beta/Foo -> FAILS. To access any file or directory, a user must have **execute (x) permission** on *every parent directory* in the path.

Q3: Fails too, one need x permission for all the directory in the path to the file we try to access, to “traverse them”.

Q4: manager_A need to update the permission (or root)
What about symbolic link? What about hardlink?

Special rights: `suid/sgid`

Setuid and setgid bits serve to indicate that a file is not run with the privileges of the launcher, but **with the privileges of the owner** user/group

Specially useful to run programs that require root, respecting the least privilege principle, e.g., to change a password:

```
ls -l /bin/passwd
-rwsr-xr-x. 1 root root 27768 Aug 20 2020 /bin/passwd
```

Special rights: `suid/sgid`

How do you know if a `suid` program does what it is meant to do? and only what it is meant to do?

```
-rwxr-xr-x 1 root root 3492656 Dec  4  2017 python2.7
```

Setuid Root programs are dangerous! (in TCB)

Chen, H., Wagner, D., and Dean, D. "Setuid Demystified". In USENIX Security Symposium 2002
Kamp, P.H., and Watson, R.N. "Jails: Confining the omnipotent root". Proceedings of SANE 2000

When the suid bit is set, when the program is executed it runs with the permissions of the owner. When the program ends, the permissions are returned to normal.

Like with any other program, it is very hard to know if the program is doing only what it is meant to do. Thus, setting suid on `root` programs is very dangerous, as they run on the TCB! If something goes wrong there are no more protections

Special rights: `sticky bit`

"Restricted deletion bit" (chmod +t)

Directories:

prevents unprivileged users from removing or renaming a file in the directory

unless they own the file

Example: /tmp folder. Users can only edit their own files

Files:

historically prevented program from being moved from swap for fast load

current: linux ignores the bit

It applies to directories, and it indicates that

1) the directory can **only** be deleted by the directory owner or the super user

2) files in the directory can **only** be renamed by the directory owner or the super user

The sticky bit is for instance on the UNIX /tmp, a folder share by all users where their temporary files are stored (e.g., downloads from the internet). The sticky bit ensures that only owners or super users can delete their own files

Special rights: Nobody

Special user (User ID -2)

- owns no files
- belongs to no user

- Safer user to execute code you do not know, particularly obfuscated code
- Limits damages if they misbehave / get compromised

The **Nobody** user can be used to sandbox programs. It owns no files and belongs to no user, so even if the program is malicious and gains control of the user it gains no privileges.

What about Windows?

Principals = users, machines, groups,…

Objects = files, Registry keys, printers, …

Access control:

- Each object has a discretionary access control list (DACL)
- Each process (or thread) has an access token with
  - Login user account (process "runs as" this user)
  - All groups of which the user is a member(recursively!)
  - All privileges assigned to these groups

Compare DACL with the process' access token when creating a handle to the object

Windows also uses Discretionary Access Control Lists, one per object, that contain which users have rights to access the object and for what actions

Processes have access tokens

that contain the login of the user account (as the process will run with this user's privileges

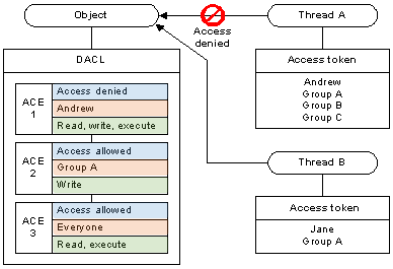
all groups of which the user is a member

the privileges of this groups

Tokens are compared with DACL of objects. If no permission, then no handle is provided.

What about Windows? DACL

List of Access Control Entries (ACEs)



Type: negative / positive

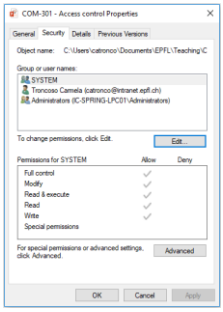
Principal

Permissions: more fine grained than UNIX

+ Flags and others…

Least Privilege by default

Run as administrator



https://docs.microsoft.com/en-us/windows/desktop/secauthz/dacls-and-aces

On the left a DACL. It contains different **Access Control Entities** (ACEs)

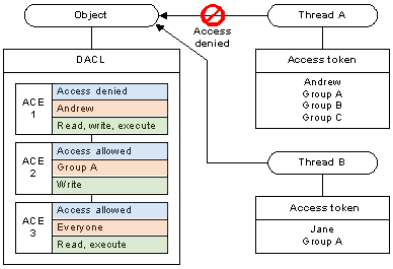
Each ACE expresses, for a login/group/everyone, the type of permission, and whether the permission is negative (not allowed actions for the principal) or positive (allowed actions for the principal)

ACEs with negative permissions are always first, so that they are check before positive permissions, following the fail safe defaults principle.

By definition, as in UNIX, processes run on least privilege. In order to change the user must run as administrator.

What about Windows? DACL

List of Access Control Entries (ACEs)



Type: negative / positive

Why negative first?

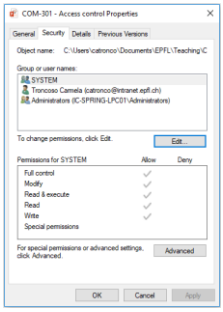
Principal

Permissions: more fine grained than UNIX

+ Flags and others…

Least Privilege by default

Run as administrator



https://docs.microsoft.com/en-us/windows/desktop/secauthz/dacls-and-aces

On the left a DACL. It contains different **Access Control Entities** (ACEs)

Each ACE expresses, for a login/group/everyone, the type of permission, and whether the permission is negative (not allowed actions for the principal) or positive (allowed actions for the principal)

ACEs with negative permissions are always first, so that they are check before positive permissions, following the fail safe defaults principle.

By definition, as in UNIX, processes run on least privilege. In order to change the user must run as administrator.