

Computer Security and Privacy Access Control

Carmela Troncoso
SPRING Lab
carmela.troncoso@epfl.ch

Some slides/ideas adapted from: Philippe Oechslin, George Danezis, Ninghui Li

Computer Security and Privacy

Access control Introduction

Carmela Troncoso
SPRING Lab
carmela.troncoso@epfl.ch

Some slides/ideas adapted from: Philippe Oechslin, George Danezis, Ninghui Li

What is “access control”?



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).

What is “digital” access control?

ACCESS CONTROL: Security mechanism that ensures that
“all accesses and actions on objects by principals are WITHIN the security policy”

Example questions access control systems need to answer:

- Can Alice read file “/users/Bob/readme.txt”?
- Can Bob open a TCP socket to “http://www.abc.com/”?
- Can Charlie write to row 15 of the table GRADES?



“authorized”
“has permission”

Only events within the security policy



“unauthorized”
“access denied”

4

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

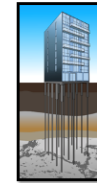
Databases Management Systems (DBMS)

Operating System

control access to files, directories, ports,...

Hardware

Memory, register, privileges

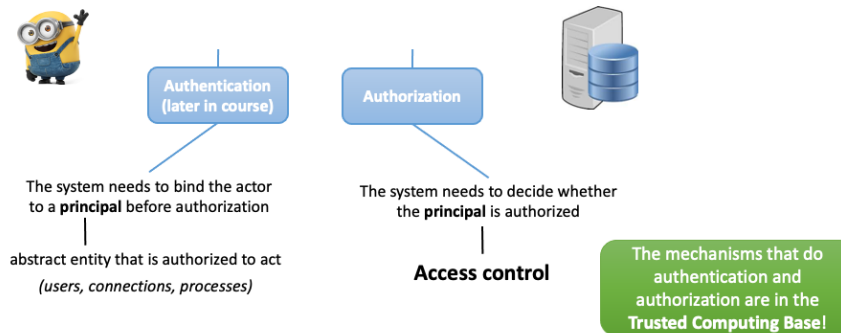


5

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.

Where does access control (usually) fit?



6

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?

Implementing access control

What you **SHOULD DO**: Systematic calls to “reference monitor”

- All over the program add checks that call the monitor
 - Checks authorisation required, and provide evidence as to the principals and objects
 - “Central” subsystem establishes whether the checks pass or not

Apache Shiro
<https://shiro.apache.org>

```
if ( subject.isPermitted("user:delete:jsmith") ){  
    //delete the 'jsmith' user  
} else {  
    //don't delete 'jsmith'  
}
```

Least common
mechanism??

8

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.

Who decides the access policy? Two approaches

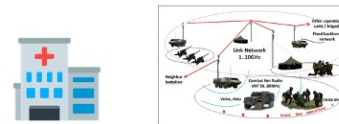
DISCRETIONARY ACCESS CONTROL (DAC)

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



MANDATORY ACCESS CONTROL (MAC)

- **Central security policy** assigns permissions
- Organizations with need for central controls
 - Military – focus on confidentiality
 - Hospital environment – focus on confidentiality and integrity
 - Banking – focus on integrity



9

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.

Computer Security (COM-301)

Discretionary Access control

Carmela Troncoso
SPRING Lab
carmela.troncoso@epfl.ch

Some slides/ideas adapted from: Philippe Oechslin, George Danezis, Ninghui Li

Implementing Discretionary Access Control

How??

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



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**

11

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.

12

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 **M_{s,o}** 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

13

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

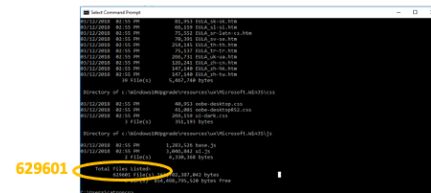
Answering the questions:

- Alice *can* read file1
- Bob *cannot* write file1
- Alice *cannot* write on file3

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

Not suitable for direct implementation!

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



$O(f \cdot u)$

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

Two more issues:

usually very sparse – extremely inefficient

error prone – hard to have a global view

14

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 $f \cdot u$, where f is the number of files in the system, and u is the number of users. As soon as one of the two number grows a little, the matrix size grows very fast.

There are two 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) Filling such a sparse, large, matrix is **error prone**: it is hard to keep a global view of the full system and make sure that no permission is entered in the wrong entry. Implementing checks to ensure that the ACM is always correctly filled may be as hard as establishing the matrix itself.

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)

Access Control Lists (ACLs)

Associate permissions to **objects**

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

Notice blanks are not stored!!

Permissions are associated to **objects**

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



can store close/with the resource
easy to determine who can access a resource
easy to revoke rights by resource



difficult to check at runtime
difficult to audit all rights of a user
difficult to remove all permissions from a user
(better remove authentication!)
difficult delegation

18

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)

Role Based Access Control (RBAC)

Systems have too many subjects! that come and go!

Large dynamic ACLs 😞

Subjects are similar to each other: assign same rights

e.g., a doctor has the same privileges as another doctor

- 1) assign permissions to *roles*
- 2) assign *roles* to subjects
- 3) subjects select an active *role* – they have the permissions of the active role

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

19

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 select 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: "Carmela'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!

20

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 almost impossible 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

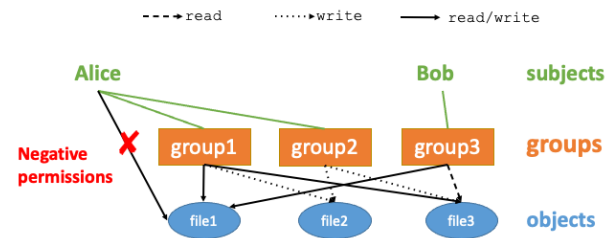
R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-Based Access Control Models. IEEE Computer, 29(2):38–47, 1996

21

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**.

Group based access control

Negative permissions to implement fine-grained policies



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

24

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

Permissions associated to **subjects**

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

Notice blanks are not stored!!



can store with the subject (portable!)
 easy to audit all subject permissions
 delegating is "simple"



revoking permission on one object is hard
 transferability, once the capability is given
 how can we prevent sharing?
 authenticity, how to check?

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

28

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 also have drawbacks:

- Revoking just one permission is hard. 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).

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)



no need to repeat the subject all the time (usability)



least privilege becomes harder to enforce
confused deputy problem!

31

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

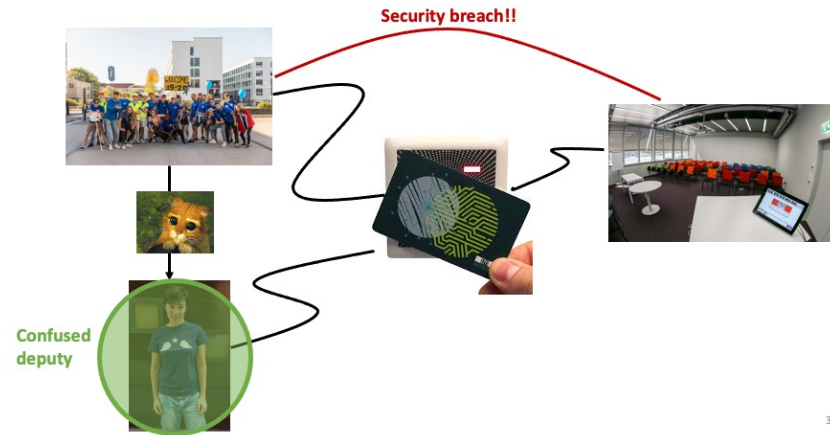
**Problem with ambient authority:
A privileged program can be tricked to misuse its authority
(Confused deputy problem)**

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



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

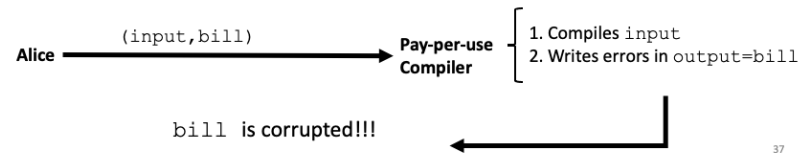
	input	output	bill
Alice	write	read	read
Compiler	read	read write	read write

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?



37

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!

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!

39

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!

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

Carmela Troncoso
SPRING Lab
carmela.troncoso@epfl.ch

Some slides/ideas adapted from: Philippe Oechslin, George Danezis, Ninghui Li

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 **Access Control List** to express the access control policy to the file
 - 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!!

44

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 file, Exec → “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

directories		owner	group	others	owner	group						
d	rwxrwxr-x	1	catronco	catronco	4096	Sep 16 14:23	exampledir					
-	rwxrwxrwx	1	catronco	catronco	8600	Sep 15 15:20	hello					
-	rw-rw-rw-	1	catronco	catronco	150	Sep 15 15:14	hello.c					
-	rw-rw--w-	1	catronco	catronco	45	Sep 15 15:07	test1.txt					
		links			size	last modified	filename					

Owner can change permissions on files

chmod $\left[\begin{array}{l} +r, -w, \\ 666, 662 \\ +t \text{ or } 1666, +s \text{ or } 4666 \end{array} \right] \text{ filename}$

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!!

49

The difference 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

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

Never login as root!

- Some distributions assign no password
- Use “sudo” or “su” command
- Difference?

```
( $ sudo su catronco )
```

51

The difference 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

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

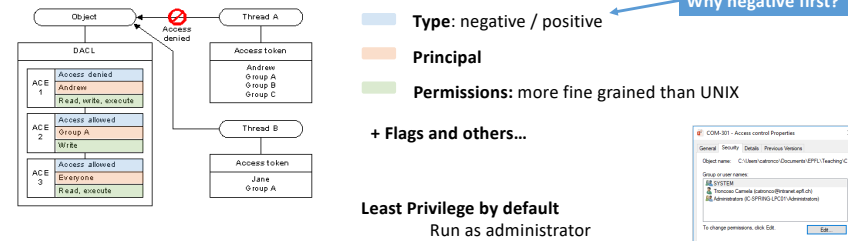
 - all groups of which the user is a member

 - the privileges of this groups

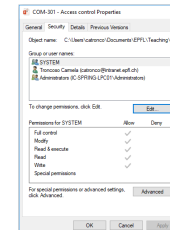
Tokens are compared with DACL of objects. If no permission, then no handle is provided.

What about Windows? DACL

List of Access Control Entries (ACEs)



<https://docs.microsoft.com/en-us/windows/desktop/secauthz/dacIs-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 checked 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.