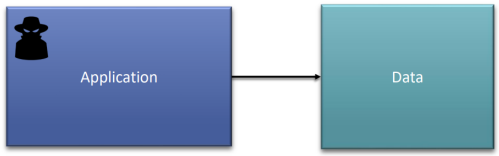
[Th 3] Priviledge separation

A privilege is the ability to access and modify a resource.

So the idea is that if an application can access to certain data and the attacker knows how to control the application then the attacker has complete access to the data.

Example:

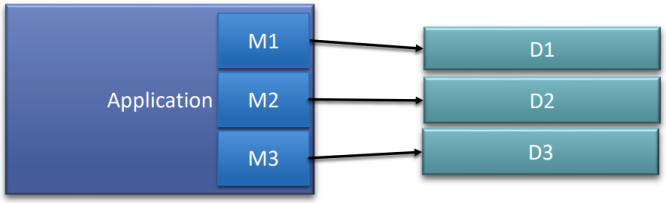


- in this case, the entire application has access to all the data managed and used by itself

To solve this problem we use the privilege separation strategy:

- decide who can access to certain data

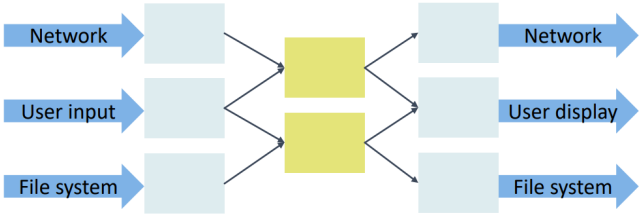
For example in this case we could divide the application into different modules and give each module only the required privilege in order to perform its work:



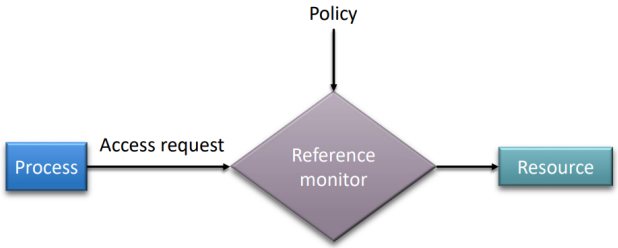
In this way, if an attacker can control a specific module he has access only to the privileges assigned to the module.

This principle can be applied to many other kinds of resources. The basic idea is to follow the principle of least privilege:

- **"A system module should only have the minimal privileges needed for its intended purposes"**



The typical privilege separation control structure is a process that requests access to the resource.
The reference monitor verifies that the process has the privilege to access the resource looking at the policy.



Privilege separation

The main idea is to divide the software and data to limit damage from bugs

Designers must choose the separation scheme:

- by user
- by service/type of data

Privilege separation needs to isolate, allow only controlled interaction, and retain good performance.

However, there are many techniques to obtain privilege separation, for example:

- virtual machine
- unix strategies

UNIX MECHANISM

They are based on the principals (entities that have the process rights).

Each process has a

- **user id (32 bit) (approximation)**
- **list of group IDs (32 bit)**

Root principal (authenticated root) is represented by the uid=0 and can bypass many checks.

In Unix, the processes are the subjects that do operations.
The objects to protect are files and directories, file descriptors (ci dicono che abbiamo i diritti su quel file), networks, processes, memory, and external devices.

Each process can use the privileges associated with its uid and its group id.

T**he Unix kernel decides if a process can work on an object by looking at the:**

- **Access Control List (ACL)**

The ACL is a function that maps objects to a couple (user, privilege).
(in pratica dice su questo oggetto l'utente ha un certo privilegio, hashmap)

In addition in Unix, we call **inode** a data structure that is associated with directories and files that contain a uid and a gid.
It contains the writing reading and executing permissions for owners and groups that are represented as an 8-bit vector of numbers that represents the 3 possible operations:

- **w for writing**
- **r for reading**
- **x for execution**

so when we use

```
chmod +x file
```

- we are adding the execution permission to the file inode

For example, if we have a file with an inode = 644:

- 6 means that the owner has (rw)
- 4 means that groups have only r
- 4 means that for all other users, the permission is r

When we open a file we obtain a **file descriptor** that says if we have access to this resource.

- These file descriptors can be shared between processes using sockets in order to pass the permissions.

In general, a process cannot use another process memory.

What processes can do on Unix?

A process can do debugging, send signals, etc but debugging and sending signals can be done on processes that have the same UID.

What network can do on Unix?

In general, it is possible to bind on a specific port, connect on specific addresses, and send and receive packets.
But the important thing is that only the root user (UID = 0) can connect to a port below 1024.

UNIX MECHANISM SUMMARY

In conclusion, we can say that the mechanism used in Unix **gives good protection to the users and is flexible enough to allow certain operations.**
One bad thing is that if we use **sudo** we can bypass all checks. **In addition, is not possible to have sub-permission** (root or not root).

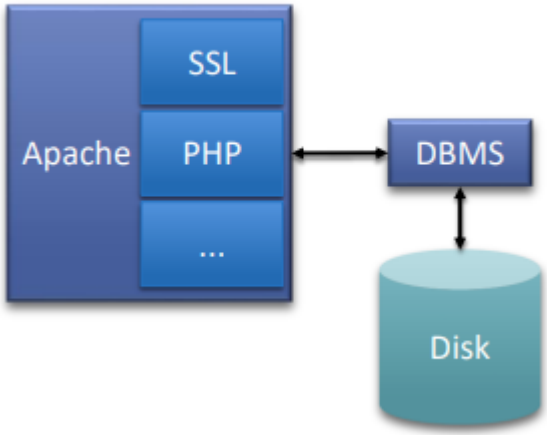
WEB SERVER ARCHITECTURE (APACHE)

Apache runs N identical processes leveraging on HTTP requests.
Each process is run as the same identical user "**www**" and the application code (example Php) runs on each process.

Any access on the OS (files, processes, etc) is performed using the www user UID.

Another problem is storage because we use SQL databases usually with one unique connection to the full DB.

- this means that if one component is compromised then the attacker has access to the entire DB.



The most common types of attacks on web applications are:

- unintended data disclosure
- remote code execution (RCE), for example, buffer overflows
- attacks to bugged code (SQL injection...)
- attack to web browsers (XSS ...)

WEB SERVER ARCHITECTURE (OKWS)

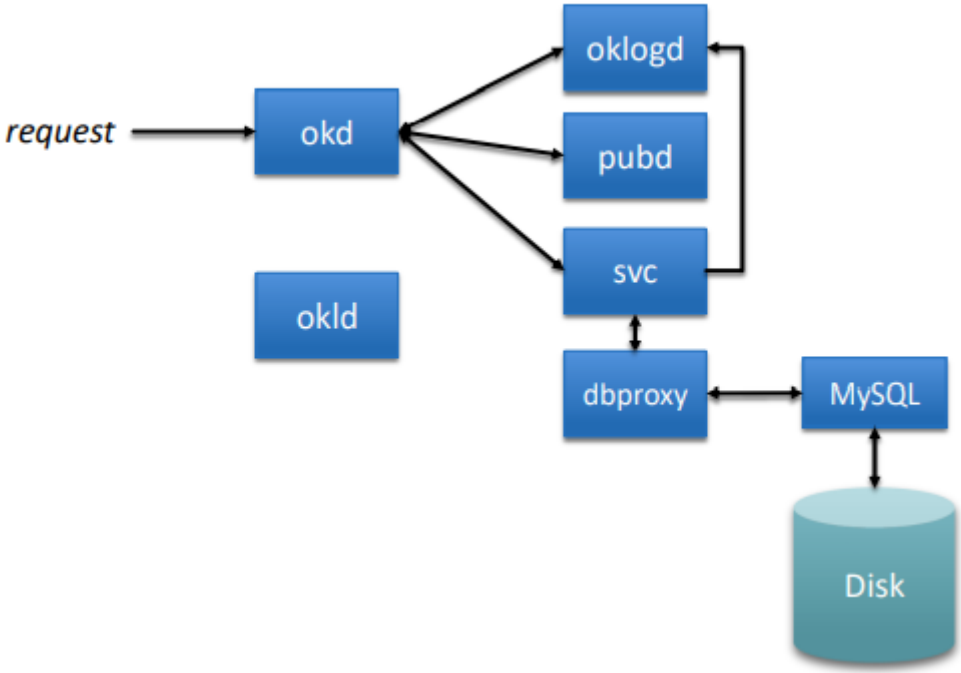
OKWS is a web server born to give secrecy to the data of a dating website.
Here many operations were done on the server side and it did not require a deep privilege separation because we needed a certain data sharing between the different users (to have a match with another user for example).

The application was structured in different modules:

- **the dispatcher that was responsible for managing requests (okd)**
- **the logger used to manage login (oklogd)**
- **a template generator (pubd)**

- **a number of services**
- **the system launcher**

The important thing here is that the system was not directly connected to the DB but there was a **dbproxy** used to manage the connection between the system and the DB.



Each service is executed with a different uid and gid but the okld that was executed as root.

Another important thing is that each process is confined to the chroot in a specific folder.

Each component communicates with the others using the UNIX domain sockets, exchanging the file descriptors.

OKWS DBPROXY

It is an intermediate layer that ensures that each process cannot access other processes' data.
Its protocol is defined by the programmer but in general, the most used template was SQL.

It is based on the use of:

- **20-byte tokens that are passed to the services that are checked using a list of valid tokens**
- **so each query is prefixed and is represented by the token**

But if we can get the token we can compromise the query assigned to the token.

OKWS Isolation

The isolation in this web server is based on the fact that each service ruins a separate UID and GID (only okld is run as root)
Each component communicates with the others using the UNIX domain sockets, exchanging the file descriptors.
We don't use per-user isolation because it may require memorizing the user's uid increasing the okld complexity and reducing the performance.

OKWS Achievements

OKWS is able to mitigate most of the problems but is not able to mitigate XSS attacks.
In addition, it is based on the assumption that the programmer does things in the right way and designs the application in the right way defining the dbproxy protocols.

It is not used in web applications because it requires programmers to define the APIs and it was not easy to define a priori the fixed queries.
It was a good alternative to Apache and the only service run as root was okld and is a good solution for client-side vulnerabilities.

OKWS LIMITATIONS

In general, OKWS could have some different limitations:

- an attacker could find SQL injection attacks in some dbproxy
- he can find logic bugs in service code for example in authentication service code
- he can find client-side vulnerabilities

CAPABILITIES

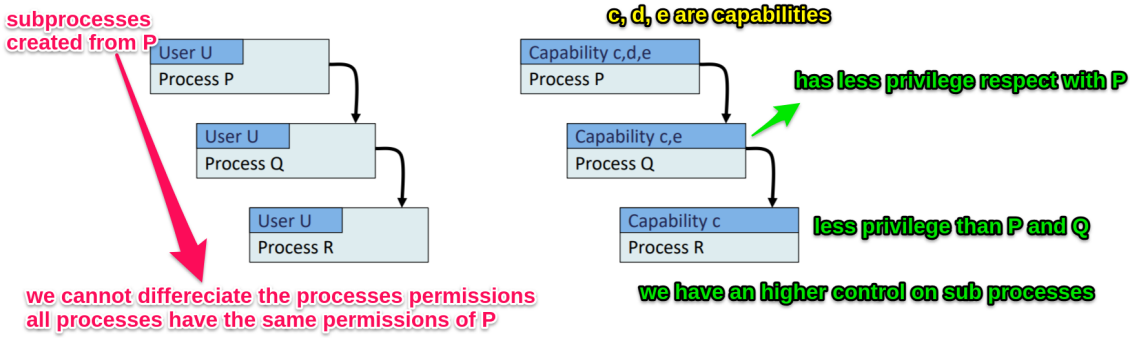
In this structure, each user has a capability (a ticket) that allows access to a specific resource.
It can be a random bit sequence or managed by the OS.
Also, the UNIX file descriptors can be seen as capabilities for a certain file.

A capability can be passed from one user to another and there is a reference monitor that controls them.

ACLS vs CAPABILITIES

We know that ACLS generates and associates each object with a list that contains couples (user id , privileges), so they are based on user authentication.
On the other hand, capabilities are only tickets and the controller doesn't need to know the identity of processes or users.

This image explains the difference between capabilities and ACLS:



A process can transfer at runtime his capabilities. With ACL we need to execute other subprocesses with the owner privileges.
On the other hand, to remove a privilege using the ACL we just remove the user from the list, differently with capabilities we can revocate a privilege only in certain cases.

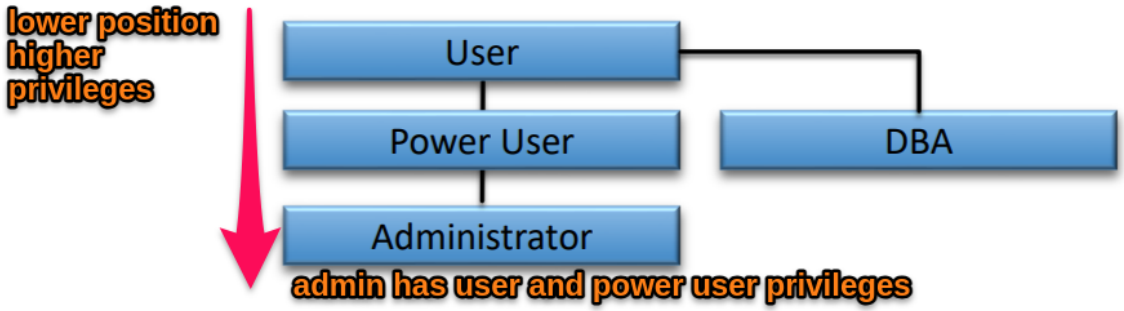
LINUX uses both ACL and capabilities, they are not alternatives, we can just use one of them or the other to respect with our needs.

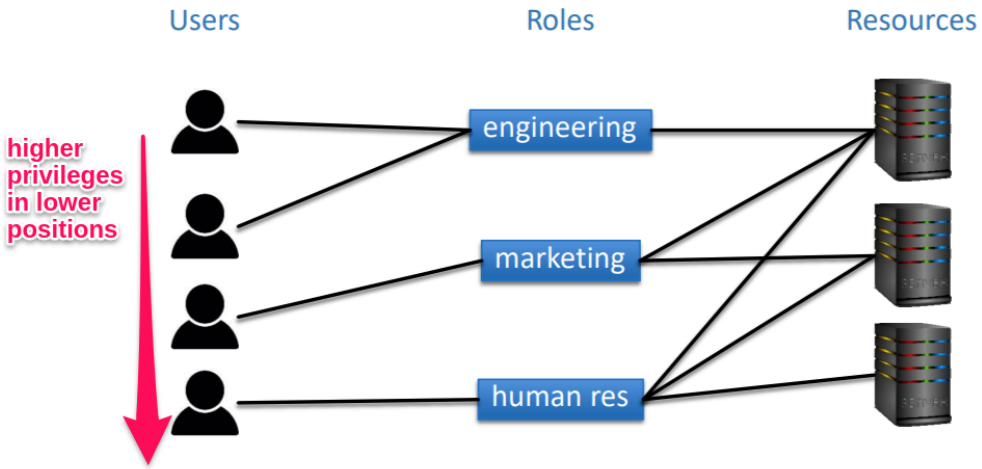
ROLE-BASED ACCESS CONTROL

Role-based access control (RBAC) is used basically in DBMS. The idea is to introduce user role hierarchies.

A role is a set of users (a group) to which we assign some privileges.

Each sub-role obtains the privileges of its ancestor.
So defining a new role we are defining only the additional privilege it has with respect to its parent group.


lower position
higher
privileges
User
Power User
DBA
Administrator
admin has user and power user privileges


Users
Roles
Resources
higher
privileges
in lower
positions
engineering
marketing
human res

SANDBOXING

Sandboxing is a mechanism used to separate running applications.
In general**, it is the practice of isolating a piece of software so that it can access only certain resources, programs, and files within a computer system, so as to reduce the risk of errors or malware affecting the rest of the system.**
The objective is to run applications in a way that is not possible to afflict the entire system in case of problems.

For example, we could use sandboxing when we download untrusted code from the internet (Google Drive checks files in sandboxes).

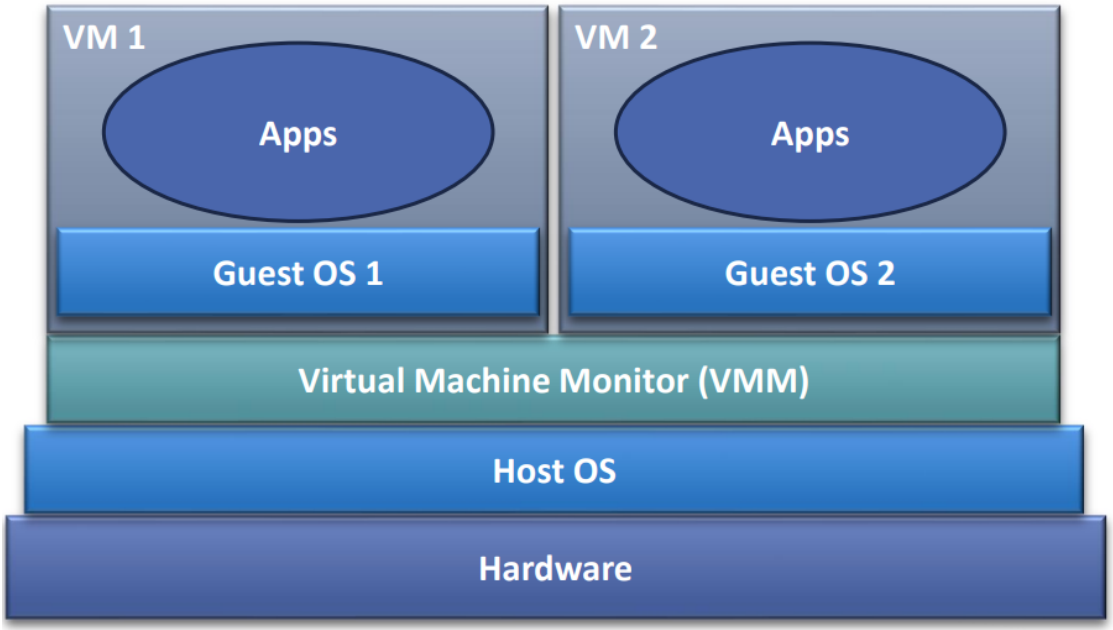
Some typologies of sandboxing are based on the security primitives of the OS, for example in UNIX we can refer to user id or group id.

Other mechanisms could be:

- language-level isolation (like javascript)
- virtual machines

VIRTUAL MACHINES

A virtual machine is a very strict isolation mechanism.
Using it we can run untrustworthy code inside a virtualized environment.



The VMM is a layer that is used to create the virtual machine. It is also called a hypervisor or virtualizer

The virtual machine concept is based on a very strong security assumption:

- malware can infect guest OS (VM OS) but cannot escape from the VM
- malware cannot infect the host OS and cannot escape into other VMs on the same hardware

Basically, we require that also VMM (which is much simpler than a full OS) must be protected.

The good things about a VM are that

- the code in the sandbox has no interactions outside of the VM
- is compatible with other isolation mechanisms

Bad things are that:

- we have problems with sharing processes, files, etc
- VMs can introduce overhead

LINUX KVM AND QEMU

The KVM (kernel-based virtual machine) is the module in the LINUX kernel that makes the kernel work as a hypervisor, it is used to manage virtual CPUs and virtual memory.

QEMU (Quick Emulator) is a software that is used to emulate devices in a virtual way:

- **so we can emulate for example a RAM or a CPU or the BIOS**

LINUX CONTAINERS

A Linux container provides the illusion of a virtual machine without using a VM and for this reason, is more efficient.
A container is a Linux process but strongly isolated, it has:

- **limited access to kernel namespaces**
- **limited access to system calls**
- **no access to the file system**

They behave like a VM:

- they start with a VM image
- have their own IP address
- have their own file system

The basic idea is to use containers for executing a privilege separation.

From this bases, the plan is to turn this single process into a virtual distributed application.

1. **We create a container for each service and copy the right files into the container.**
2. **we assign to each container a IP address**
3. **we connect each container with TCP**
4. **we setup firewalls to limit the communication between containers**

EXAMPLE: AMAZON FIRECRACKER
Amazon lambda service runs customer-supplied Linux applications (FIRECRACKER).
The security challenge here is to isolate the code of each customer from the others

FIRECRACKER
Its design is based on the use of

1. **a VM monitor with very lightweight machines, called microVMs.**
2. **a KVM for virtual CPU and memory.**
3. **Reimplementing QEMU**
4. block devices, instead of using the file system we use here
    1. 4 kb where we can read and write the entire block
5. NO BIOS
    1. just load the kernel into a VM, init and run it

The implementation is based on:

- rust language
    - 50k lines of code
- running it on a jailed process
    - chroot to limit the files accessible from other processes and network
    - separate userid

How lambda uses firecracker?

Basically, we have many workers (firecracker micro VMs) that have a fixed number of micro VMs slots that are runnable.

Sandboxing low-level code in browsers

The main objective is to allow web apps to run native code on the client host and not on the server.
So this can be useful to run languages other than javascript and to guarantee the support for legacy applications.

We have 3 different approaches to obtaining it:

APPROACH 1: ASK USER TO TRUST DEVELOPER CODE

The idea is to ask the user if trust or not the code we want to execute.
However, the final user may not have the knowledge to execute the right choice.

APPROACH 2: SANDBOXING AND HARDWARE PROTECTION

Similar to OKWS is to protect the hardware or the sandbox executing the code into a secure user space like the VMs.
But here we need to control what system calls the untrusted code can invoke.

Some problems are:

- different OS can impose different and incompatible requirements
- OS kernel vulnerabilities are frequent
- not every OS has sufficient sandboxing mechanisms
  - example many of them require root permissions **(but we must not run a browser as root!)**
- possible hardware vulnerabilities

APPROACH 3: Software issues isolation

Here the strategy is to verify line by line the x86 code to understand if it is safe.
This strategy is used in the Native client of Google

Native client

The strategy in the native client is to not execute non-authorized code or in general code that goes out of the imposed bounds.

So we perform a revision of the entire code, if the revision is okay then we can execute it.
In general x86 code instructions length is not fixed so we could have some ambiguity.

The best approach is to use reliable disassembling that checks if the code executes only instructions that are well-known by the verifier and that checks that each jump goes is a valid memory destination.

To check if a module accesses or executes out-of-bounds code we can use the segmentation:

- **the memory is divided into segments (base + size)**
- **for each segment, we store a pointer in a table**

We can also force to create code of a certain segment and use it to store all pointers

- **we just reject each code that modifies these pointers**