SUPSI

Operating System Security (1)

Operating Systems

Amos Brocco, Lecturer & Researcher

14 settembre 2018

Objectives

- Understand basic security mechanisms used in current OS
- Study how DAC and MAC work
- Study the concept of application sandboxing and isolation
- Study how we can control system resources
- Study how secure boot works

Browsing
- Get a rapid overview.

Reading
- Read it and try to understand the concepts.

Studying
- Read in depth, understand the concepts as well as the principles behind the concepts.

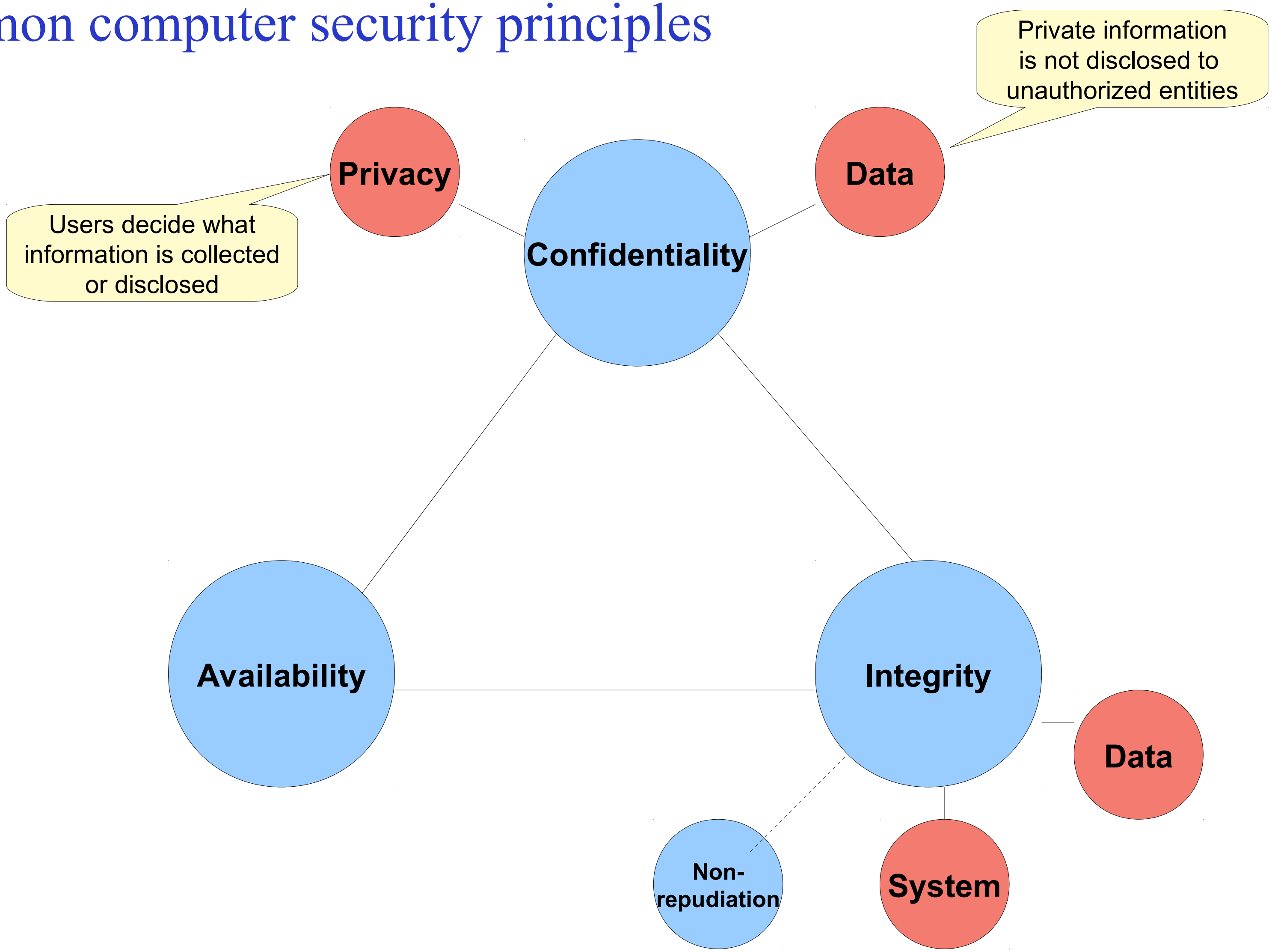
You are also encouraged to try out (compile and run) code examples!

Security of a computer system

- The security of a system is determined a **security policy**
  - A **security policy** is a statement of what is, and what is not, allowed.
  - A **security mechanism** is a method, tool, or procedure for enforcing a security policy(*how* to enforce a policy)
- A **security model** is a model that represents a particular policy or set of policies
  - models typically follow one or more **principles**

(reference Matt Bishop, “Computer Security: Art and Science”, 2015)

Common computer security principles



Ideal security model

- **Inescapable**
  - inability to break security policies by circumventing access control mechanisms
- **Invisible**
  - seamless user and administrative interaction
- **Feasible**
  - cost-effective and practical to implement

* J. Daniel E. Geer “The shrinking perimeter: Making the case for data-level risk management” Veradsys White Paper, 1.2004

The system tries to protect itself

- To counter security treats an operating system first needs to employ some mechanisms to protect itself:
  - separate address spaces
  - separate kernel/user execution contexts
  - user authentication
  - access control (authorization)
  - threat detection and defense techniques

… in this class we will focus on *separate execution contexts*, *authentication* and *authorization*

Separate execution contexts

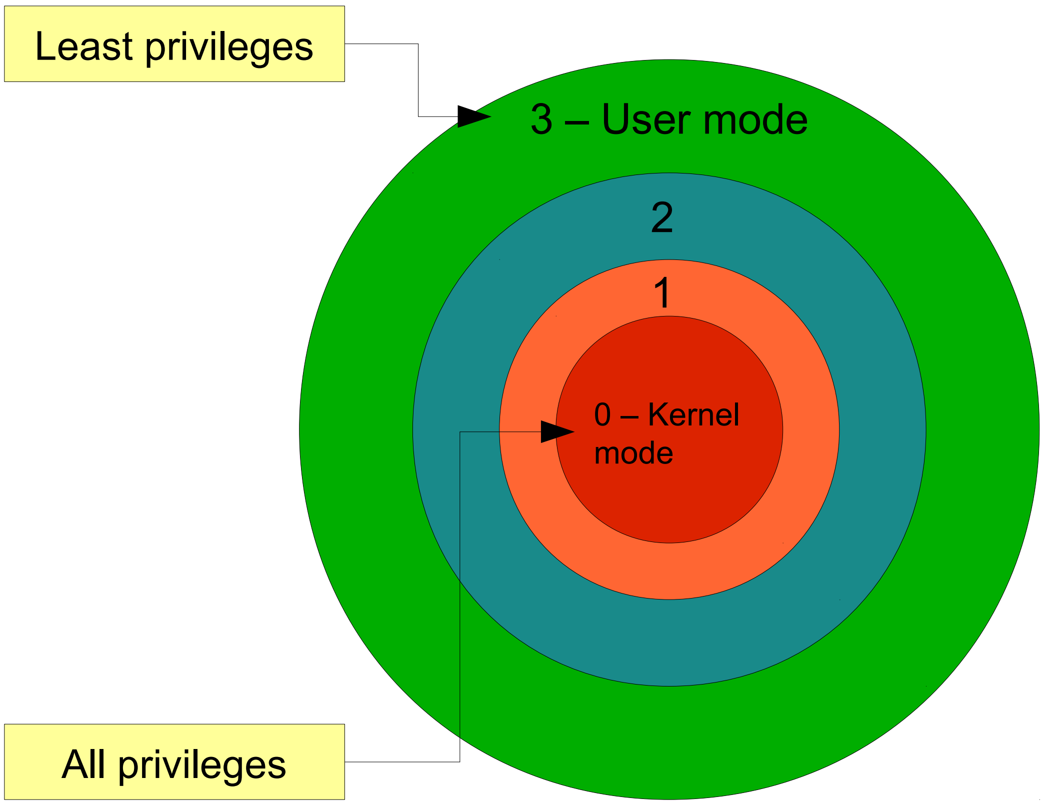
- An operating system must make use of some robust **protection** mechanism
  - to prevent user processes from interfering with or bypassing operating systems mechanisms and abstractions
  - to prevent user processes from using potentially dangerous machine instructions
  - to prevent user processes from calling system procedures without control

Kernel mode, user mode

- Some instructions (for example those to disable interrupts or that modify address translation tables) should not be available to user processes.
- The CPU enables the distinction between two execution modes:
  - **Privileged** (***kernel mode***)
    - Complete control of the hardware
    - “With great power, comes great responsibility”: an error might crash the whole system
  - **Unprivileged** (***user mode***)
    - Hardware* and abstraction access through system calls
    - Errors should only affect the process itself, not others (isolation)

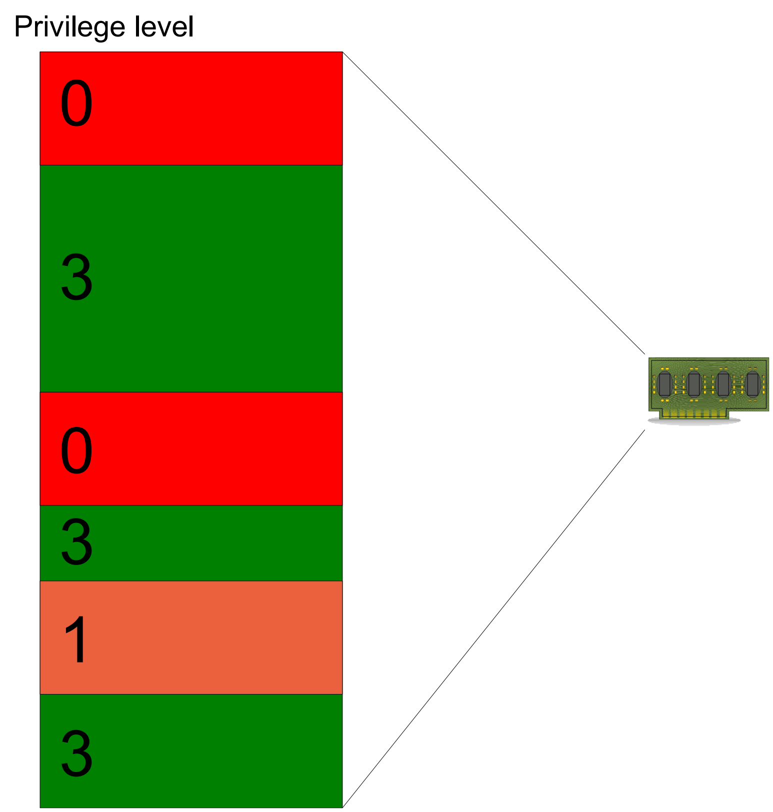
* apart from memory and CPU

Ring based security



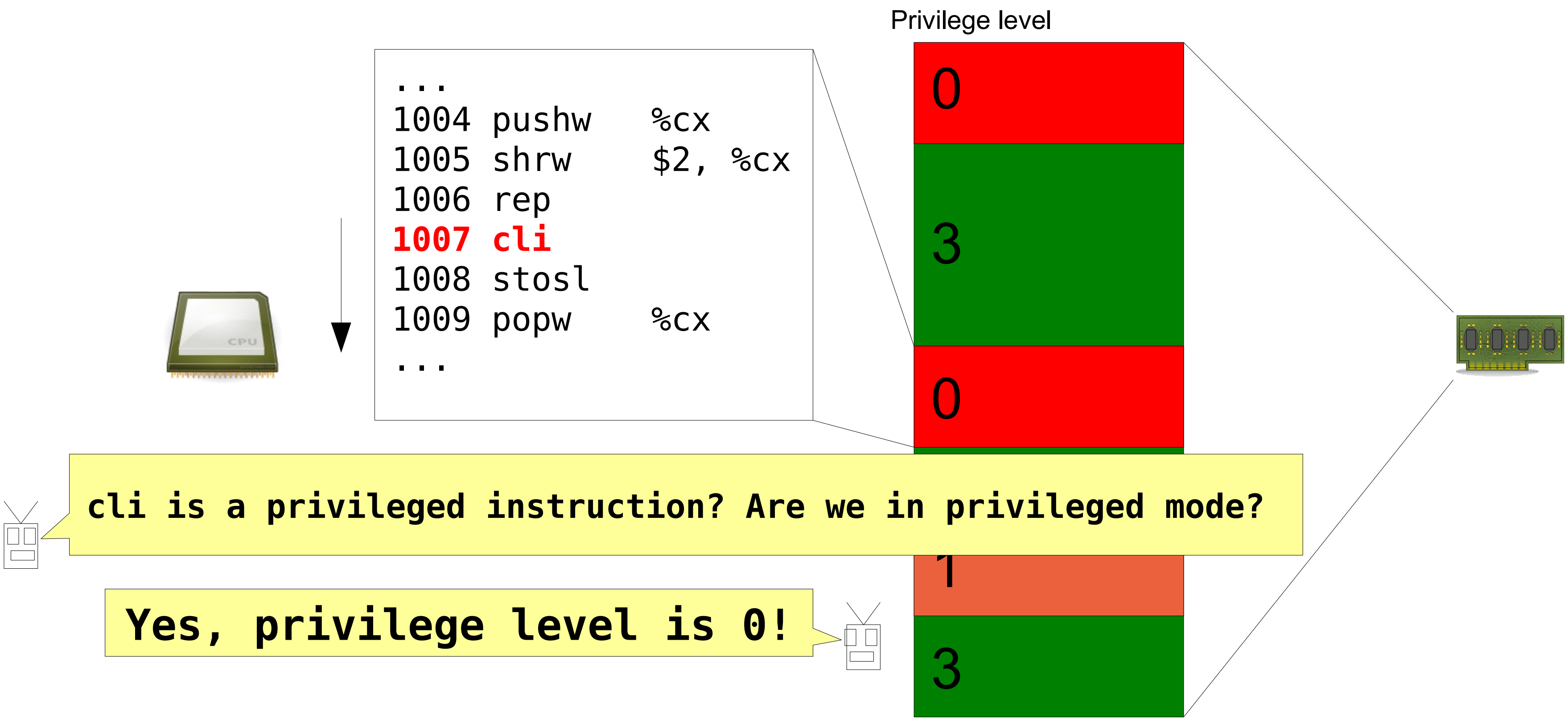
How does the CPU know it it's running privileged code?

- Memory areas * can be assigned with a privilege level:
  - The privilege flag of these areas determines the privilege level for the executing code
  - If a privileged instruction is executed from an unprivileged level an protection fault is generated
- On x86 the current privilege level can be obtained by reading a CPU register (code selector)



* *segments*, *pages* → see chapter about memory management

Example


Privilege level
...
1004 pushw %cx
1005 shrw $2, %cx
1006 rep
1007 cli
1008 stosl
1009 popw %cx
...
0
3
0
1
3
CPU
cli is a privileged instruction? Are we in privileged mode?
Yes, privilege level is 0!

Privileged vs unprivileged: an example

- Unprivileged version

```
#include <stdio.h>

void main(void)
{
    printf("Something bad will happen... see you in 'dmesg' :)\n");
    __asm__ __volatile__("cli" :);
    __asm__ __volatile__("sti" :);
    // Crash: traps: gpf[3282] general protection ip:4004f1 sp:7fff8eadaeb0 error:0 in gpf[400000+1000]
}
```

- Privileged version (Linux kernel module)

```
#include <linux/module.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");

static int __init init_privileged(void) {
  printk("Hello world from a dangerous module\n");
  __asm__ __volatile__("cli" :);
  __asm__ __volatile__("sti" :);
  return 0;
}

static void __exit exit_privileged(void) {
  printk("Goodbye cruel world!\n");
}
module_init(init_privileged);
module_exit(exit_privileged);
```

`cli`: disable interrupts
`sti`: enable interrupts

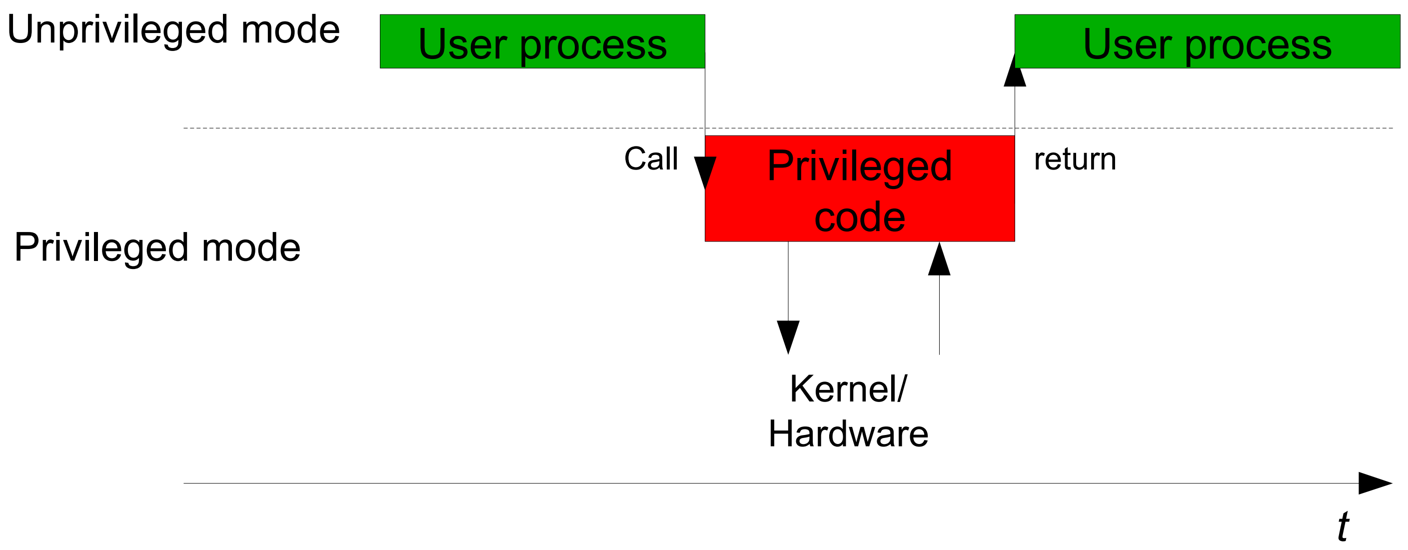
x86 privileged instructions

We are not done yet

- **Facts**:
  - The kernel runs in privileged mode and sets up user process memory as unprivileged.
  - System calls are implemented by the kernel, and thus stored in a higher privileged area of the memory in order to run in privileged mode.

- **Question**:
  - How can user processes *jump to privileged mode* to execute these system calls?

System calls: overview



System calls: naïve implementation (jump in the code)

User mode

```
X="foo"
Y="bar"
writeOnScreen("Hello world");
...
```

Jump to start of writeOnScreen

```
writeOnScreen(msg) {
      t = getCurrentTerminal();
      write(t, msg);
      return 0;
}

formatHardDrive(drv) {
   ....
}
```

Return

Kernel mode

System calls: naïve implementation (jump in the code)

User mode

```
X=”foo”
Y=”bar”
formatHardDrive(“/dev/sda”);
...
```

```
writeOnScreen(msg) {
      t = getCurrentTerminal();
      write(t, msg);
      return 0;
}

formatHardDrive(drv) {
   ....
}
```

Kernel mode

Security issues

For security reason we cannot freely jump between memory regions with different privilege levels *:

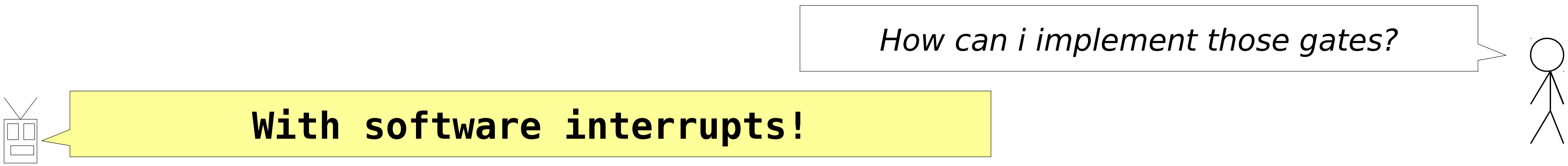
- If we could jump from a less privileged area to a privileged area (any address) we could bypass protection mechanisms within the kernel...
- If we could jump from a privileged area to an unprivileged area we could also change the return address...

... so how can we safely implement system calls?

- We need a way to allow jumping only at specific addresses (**gates**), not anywhere...

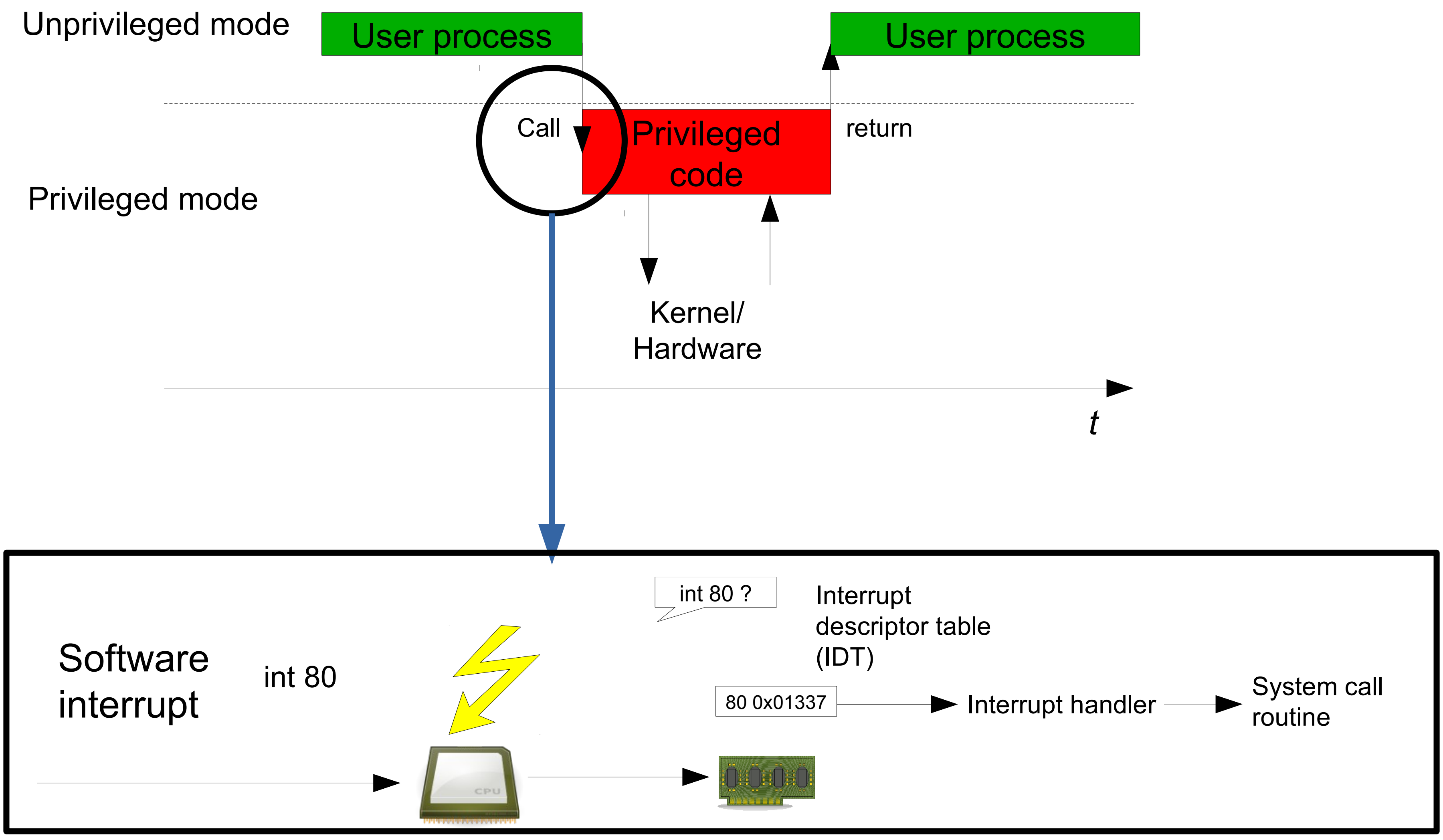
* Note: references to data at lower privilege levels are permitted; references to data at higher privilege levels are NOT permitted

Hardware can lend a hand



- You should know that **interrupts** stop the execution of a process and give control to the operating system (which runs at a higher privilege level!), to a pre-defined handling routine!
  - Unprivileged software can generate interrupts too, but the handling routine is determined by the kernel (i.e. unprivileged code cannot jump anywhere, but only at predefined addresses)

System calls with software interrupts



Privileged vs unprivileged mode

- **Facts:**
  - User applications need to perform system calls (we cannot execute solely in unprivileged mode)
  - Jumping from unprivileged mode to privileged mode (and back) for a system call does cost more than calling a function at the same privilege level
  - Calling functions using software interrupts makes debugging trickier
- **Question:**
  - So why we don't implement most of the functionalities at the kernel level?

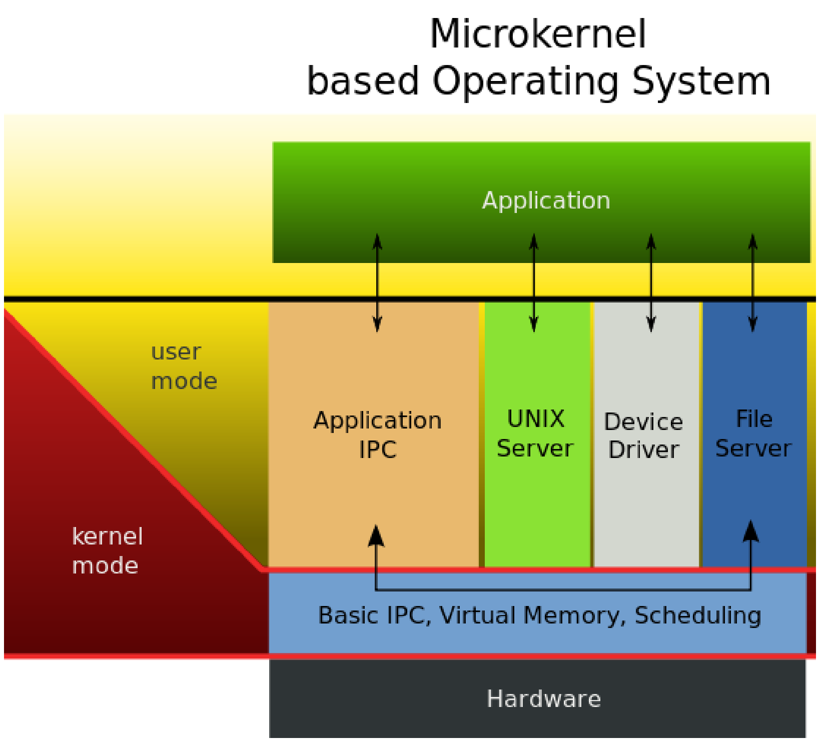
Principle of least privilege

- So why we don't implement most of the functionalities at the kernel level?
  - Security, robustness
  - **“Principle of least privilege”**
    - Give software only the privilege it needs
    - Give access only to the information and resources that are necessary for its legitimate purpose
  - We should not execute everything in privileged mode

Setting the bar: kernel architectures

- **Micro-kernel**
  - Only a minimal set of functionalities run in privileged mode
  - Typically the rest of the operating system is based on independent servers which communicate using message passing
- **Monolithic**
  - All functionalities run in privileged mode
- **Hybrid**
  - Modular structure similar to a micro-kernel, but executed mainly in privileged mode

Micro-kernel



Example: Minix 3

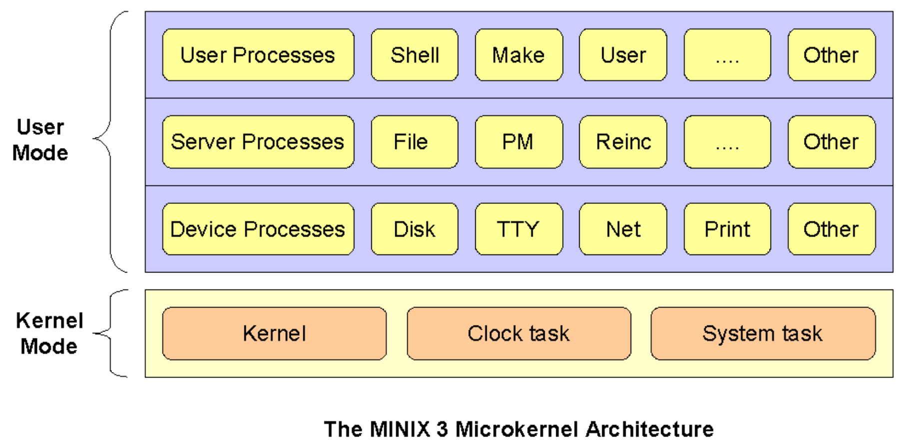
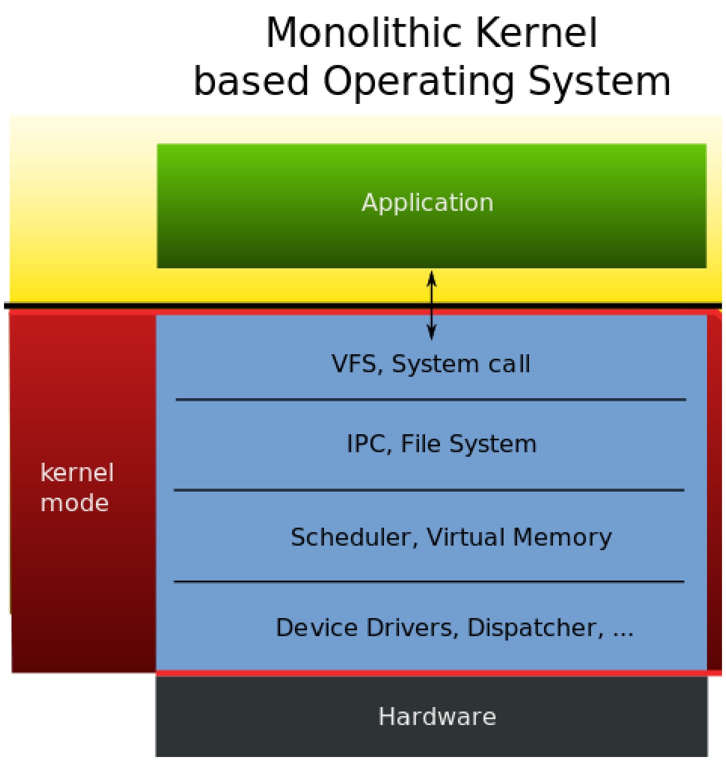


Image by Faisal.akeel at the English language Wikipedia, GFDL http://www.gnu.org/copyleft/fdl.html

Monolithic kernel



Example: Linux

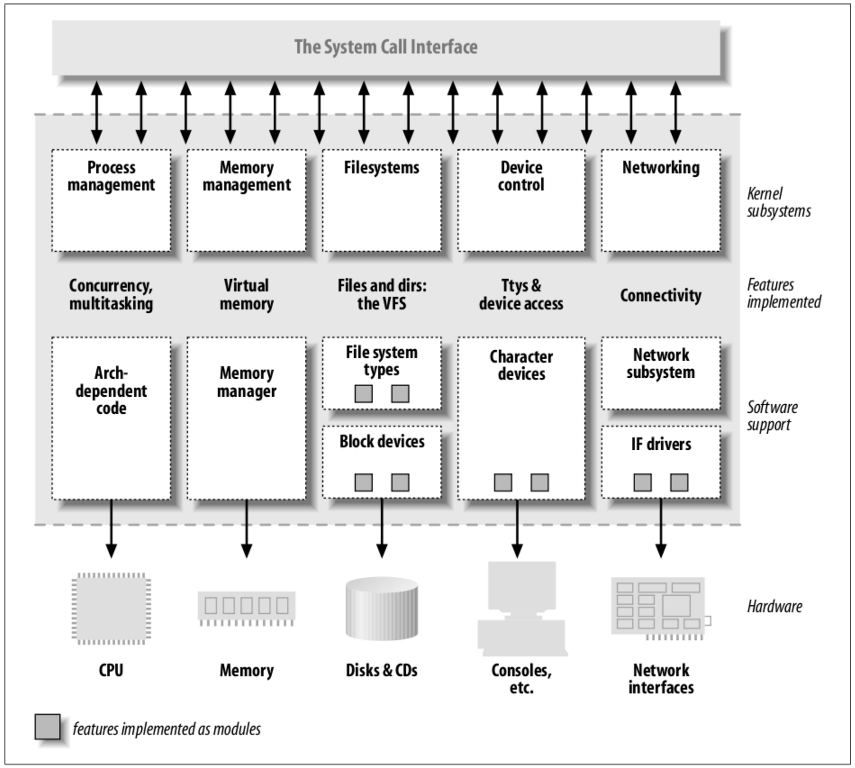
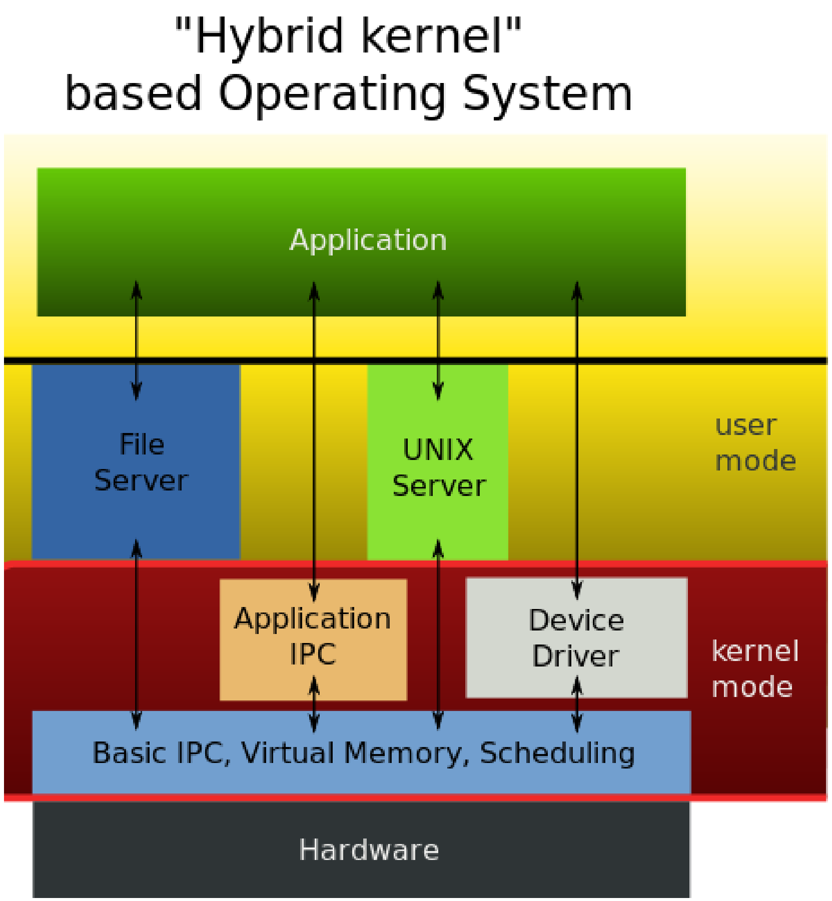


Image from Linux Device Drivers, 3rd Ed, by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. O'Reilly

Hybrid kernel



Example: Windows NT

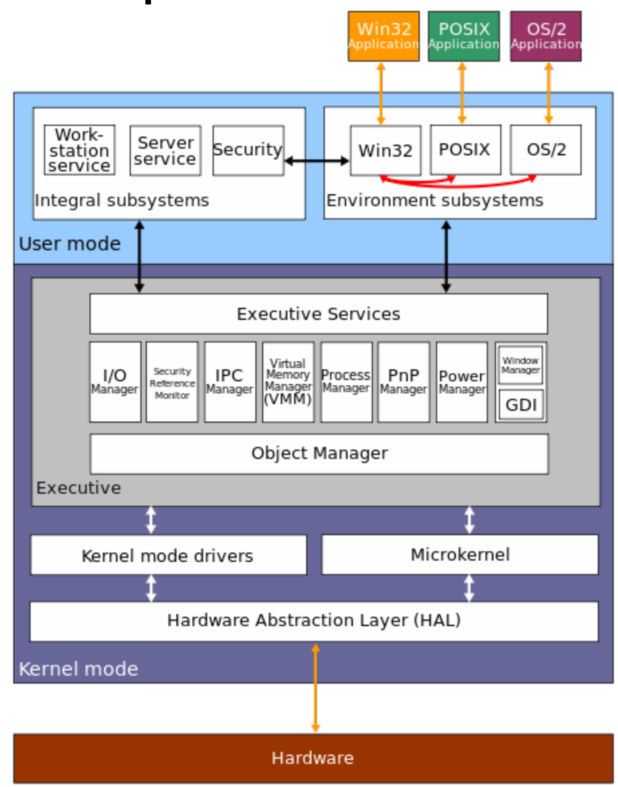


Image by Grm wnr at en.wikipedia GFDL (www.gnu.org/copyleft/fdl.html) from Wikimedia Commons

Authentication vs Authorization

- **Authentication** is the process of to verifying that a user, a device (computer, server) or a service is what it claims to be (identity check)

- **Authorization** is the process of determining what an authenticated entity can do on a system or on a network (access control)

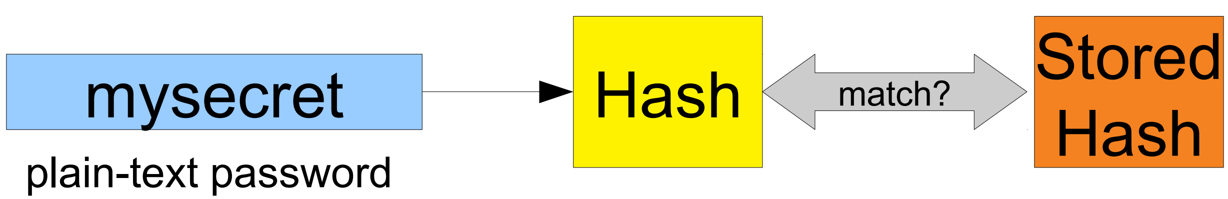
User authentication

- User authentication is use to prevent users from accessing the system or a resource.
- Typical user authentication mechanisms include:
  - **password-based authentication**
    - the user has to provide an **identifier** and a **password** which is matched again a list or database: the system can deny access to users which are not registered in the system, moreover depending on the user identity different privileges can be assigned to a user
  - **token-based authentication**
    - authentication is based on a (physical) object
  - **biometric authentication**
    - authentication is based on the user's physical characterstics

Password-based authentication

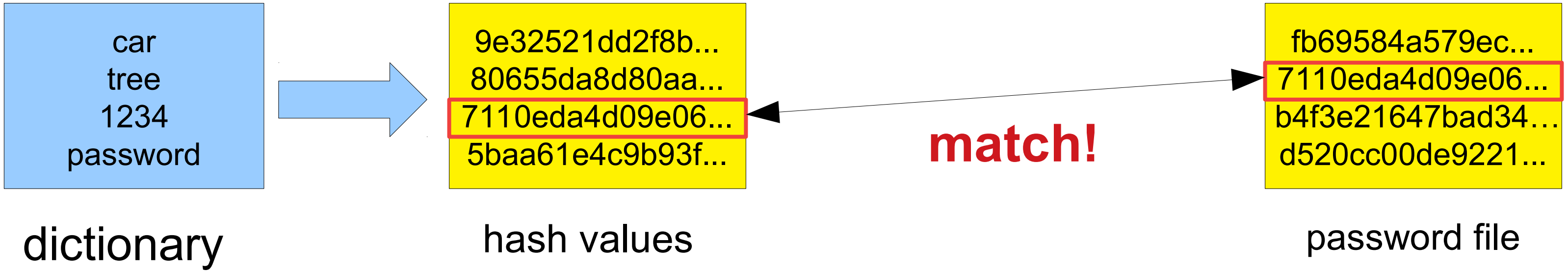
- Password-based authentication requires a match between the user's inserted password and some stored value...
  - *Should password-based authentication systems keep the real password?*
    - **No!** If the file / database is compromised an attacker would have access to all the passwords

    - An authentication system can keep and compare hash values of a password:



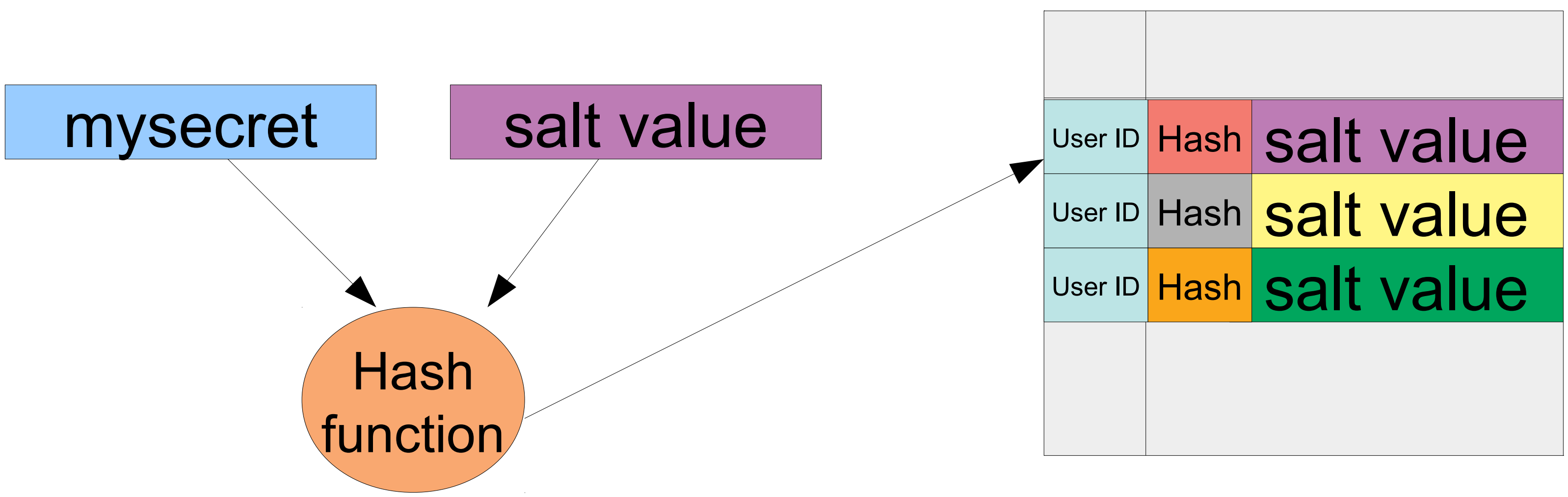
Hashed passwords are not enough

- Hashed password are not enough to ensure that the authentication system cannot be compromised
  - if two persons use the same password the corresponding hash will be the same
  - it is trivial to know if a person uses the same password on different computers
  - if a person uses a short password a dictionary attack becomes very easy: just get a large enough dictionary, generate hash values for each entry, and then compare that with the authentication database



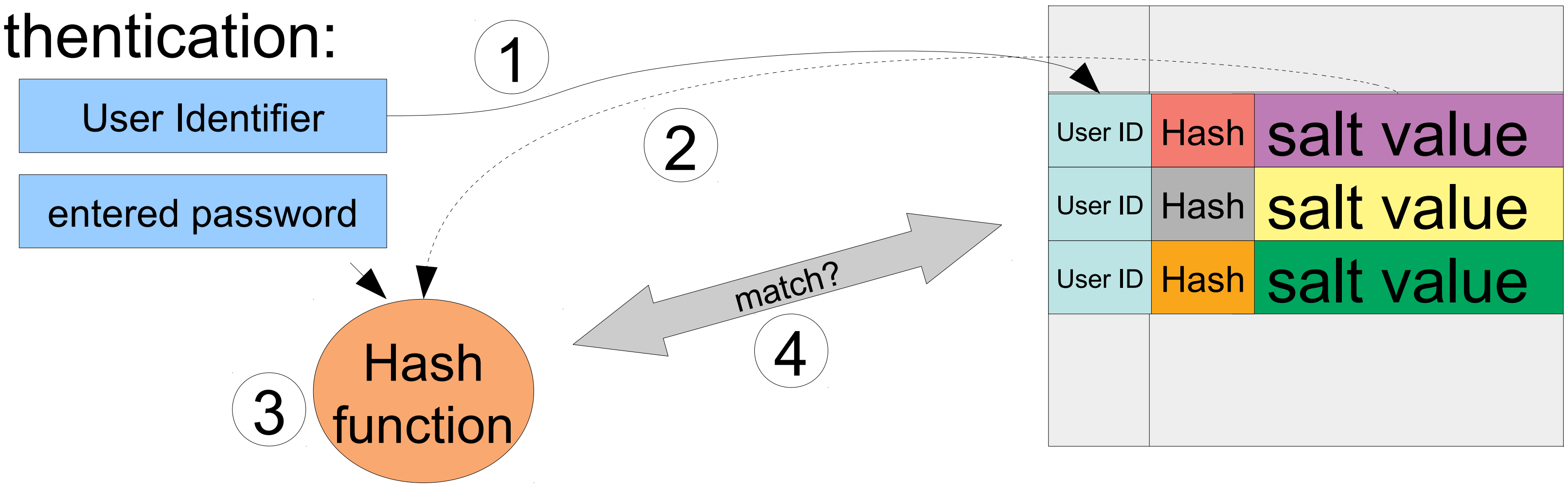
Salt value

- Instead of storing the hash of the password alone we can store the password combined with a fixed-length (random) **salt value**
  - the hashed password is stored along with the plaintext salt value
    - *How can this be more secure?*

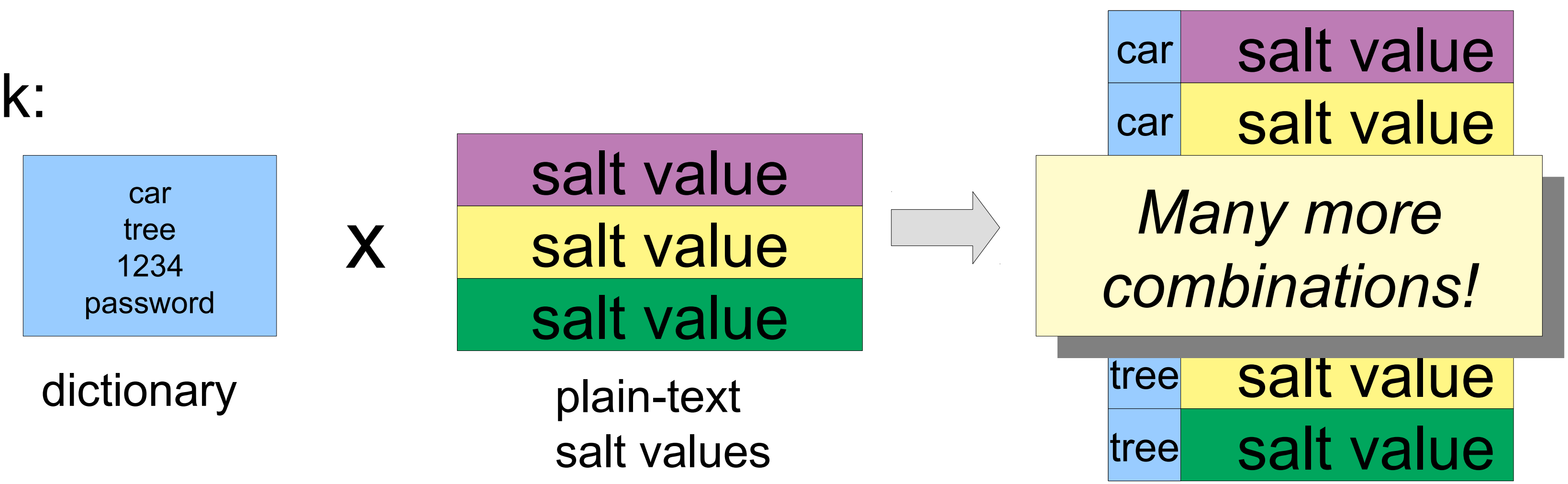


Salt value

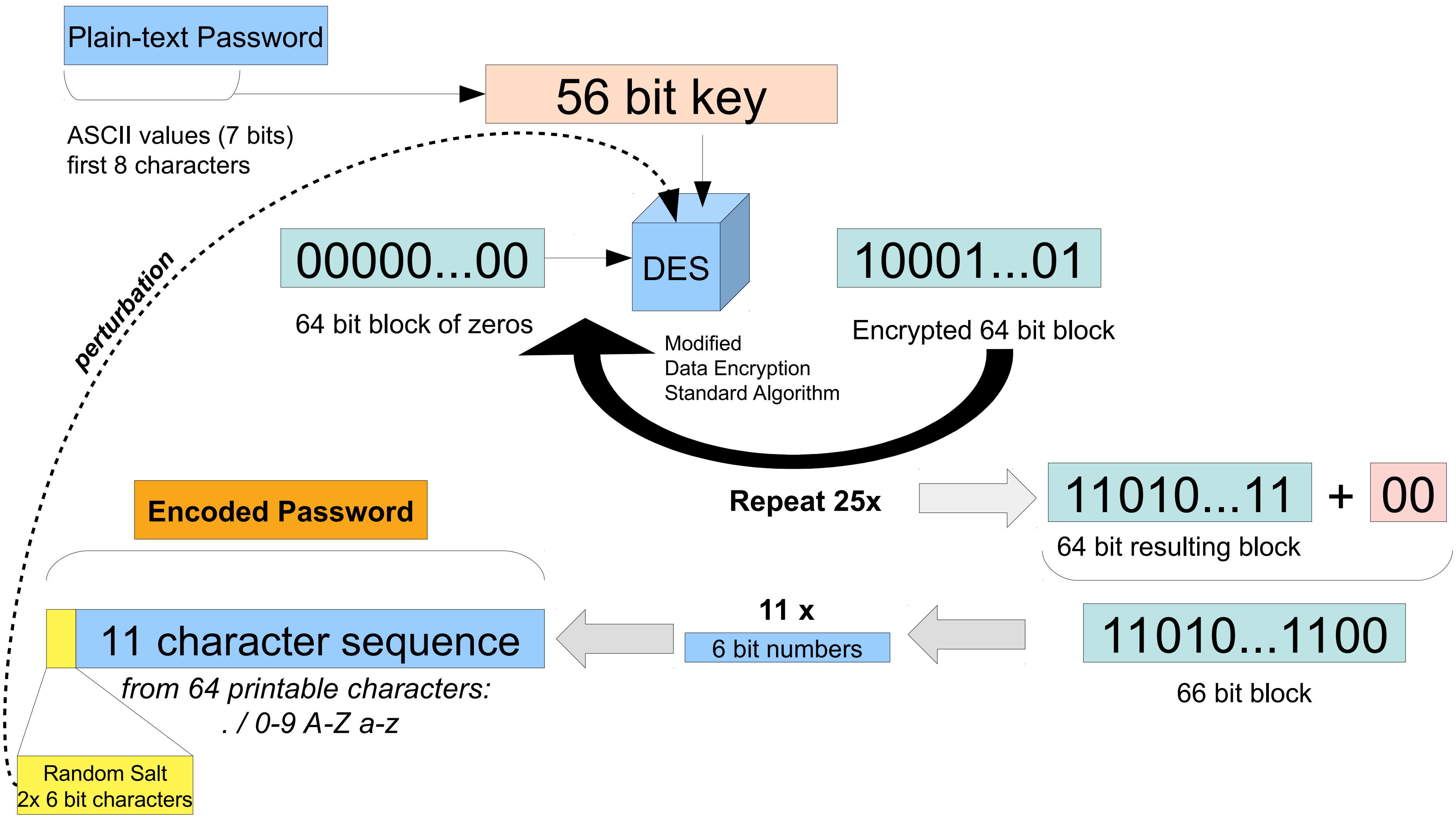
Authentication:



Attack:



Password-based authentication in Unix (and older Linux systems)



Newer Linux systems (glibc2)

- Newer Linux systems and other systems based on the GNU Libc library use stronger password encoding mechanisms:
  - MD5
  - Blowfish
  - SHA-256
  - SHA-512

  (refer to **`man 3p crypt`** and **`man crypt_r`**)

Example

```
#define _XOPEN_SOURCE
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

int main( ) {
	printf("DES Encrypted password: %s\n", crypt("mypassword", "xy"));
	printf("MD5 Encrypted password: %s\n", crypt("mypassword", "$1$mysalt$"));
	printf("SHA256 Encrypted password: %s\n", crypt("mypassword", "$5$mysalt$"));
	printf("SHA512 Encrypted password: %s\n", crypt("mypassword", "$6$mysalt$"));
	return 0;
}
```

```
DES Encrypted password: xyoxiBrqcbujE
MD5 Encrypted password:
$1$mysalt$JyKGOJR1343sJ7N91hXVI/
SHA256 Encrypted password:
$5$mysalt$k2OW8Hl3izzYeeoOKd6p5HX1/DuPJ8/et4DuwlE
BC/4
SHA512 Encrypted password:
$6$mysalt$Ts3752w7WPBw6ENoJhmynzPn47RFb2ze39cgq94
y81EGSOvtYpxE6Tatj3ZREJqN5Qn.lBw7wL81kAod63xb5/
```

The hash algorithm used by crypt depends on the prefix of the salt value

```
gcc -o cryptexample cryptexample.c  -lcrypt
```

/etc/passwd file on Unix / Linux systems

- Hashed passwords were initially stored in a file called **`/etc/passwd`**
- Today the file is used to store just some details about each user, not the password. Each line in this file has the following structure:

`username:`*`password`*`:UID:GID:fullname:home directory:logon shell`

User identifier (numerical)

Group identifier (numerical)

Path to the shell used after command-line login. Can be set to /bin/false to disable logins

- Example:

  `attila:x:1000:100:Amos Brocco:/home/attila:/bin/bash`

  – since the file is used for authentication it needs to be accessible by anyone → this is something that should be avoided

The shadow file (/etc/shadow)

- The **`/etc/shadow`** file is used to store encrypted passwords
  - the file is readable and writable only by the system administrator (***root***)
  - the structure is:

`username:passwordhash:lastchange:minimumdays:maximumdays:warndays:inactivedays:expiredays`

- username: username, as in **`/etc/passwd`**
- passwordhash: hash (with algorithm prefix). Prefix ! for locked accounts
- lastchange: days since epoch* that the password was last changed
- minimumdays: minimum interval (days) between password changes
- maximumdays: days before forced password change
- warndays: for how many days before expiration the user should be warned
- inactivedays: how many days after the password expires before the account is disabled
- expiredays: days since epoch that indicate when the login expires

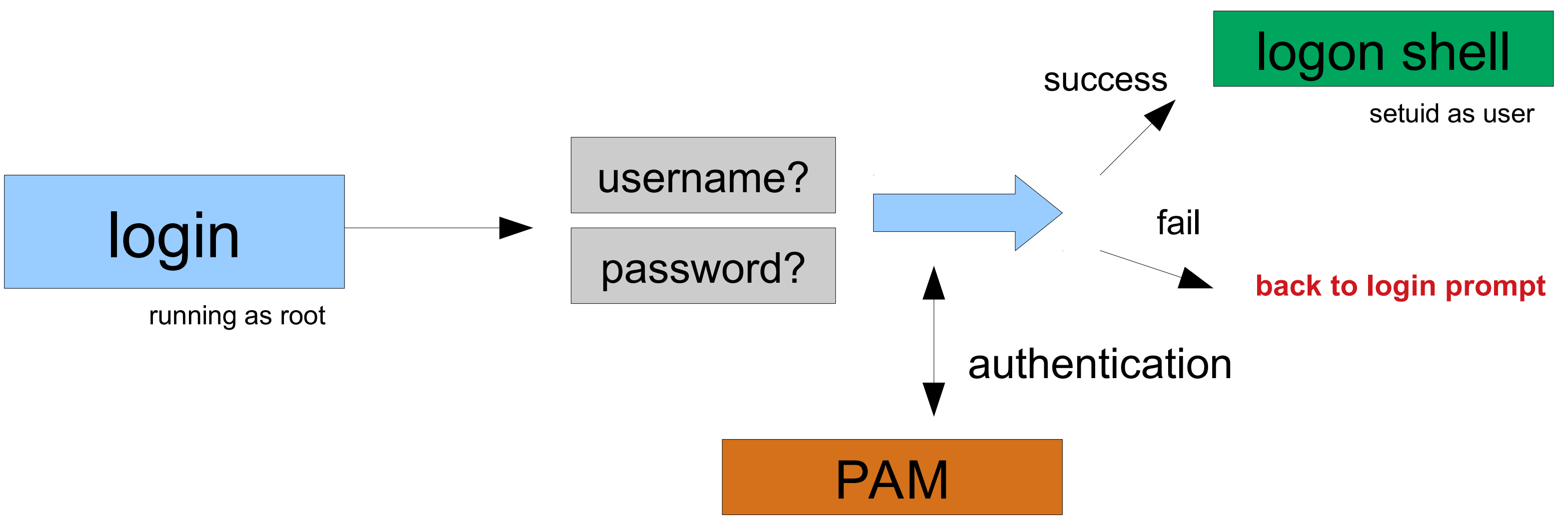
* epoch = 1.1.1970

Authentication: Linux PAM

- Linux supports a modular authentication mechanism called **PAM** (**Pluggable Authentication Modules**)
  - it provides dynamic authentication support
  - authentication modules are dynamic libraries which provide different functionalities (*realms*):
    - **auth**: responsible for authenticating a user
    - **account**: used for checking accounts
    - **password**: used for updating the passwords of a given service
    - **session**: used to setup and cleaup the user's session

Typical login procedure on a Unix/Linux system

- The login procedure (login program) in Linux / Unix runs with root privileges (the executable is *setuid* to root):
  1) the program asks for the username and password
  2) the credentials of the user are verified against the configured PAM modules
  3) if the user is authenticated the login program will start the login shell and use the **setuid** system call to change the UID associated with the shell process to the user's ID



PAM configuration

- The configuration file is **`/etc/pam.conf`** or located in **`/etc/pam.d`**

```
password        requisite    pam_cracklib.so
```

Realm

Keyword

Module

required
If the module fails the whole operation fails, but only after all other modules are invoked (to prevent an attacker from determining which module failed)

requisite
If the module fails the operation is immediately terminated without invoking any other module

sufficient
If the module succeeds it is enough to satisfy any other sufficient module (which will not be invoked)

optional
The operation fails only if the module is the only one used for the authentication

Linux PAM: an example Auth module (excerpt)

```
#include <security/pam_modules.h>

PAM_EXTERN int pam_sm_authenticate(pam_handle_t *pamh, int flags,
                                   int argc, const char **argv)
{
        struct pam_conv *conversation;
        struct pam_message localmessage;
        const struct pam_message *message;
        struct pam_response *response;
        const char *username;
        char *password;
        int error;
        /* Get the conversation item */
        if (pam_get_item(pamh, PAM_CONV, (const void**) &conversation) != PAM_SUCCESS) {
                return PAM_AUTH_ERR;
        }
        /* Get the username */
        if (pam_get_user(pamh, &username, 0) != PAM_SUCCESS) {
                return PAM_AUTH_ERR;
        }
        /* Define the ask password message */
        localmessage.msg_style = PAM_PROMPT_ECHO_OFF;
        localmessage.msg = "(C) netID Password:";
        message = &localmessage;
        /* Ask for the password */
        response = NULL;
        error = conversation->conv(1, &message, &response, conversation->appdata_ptr);
```

Linux PAM: an example Auth module (excerpt)

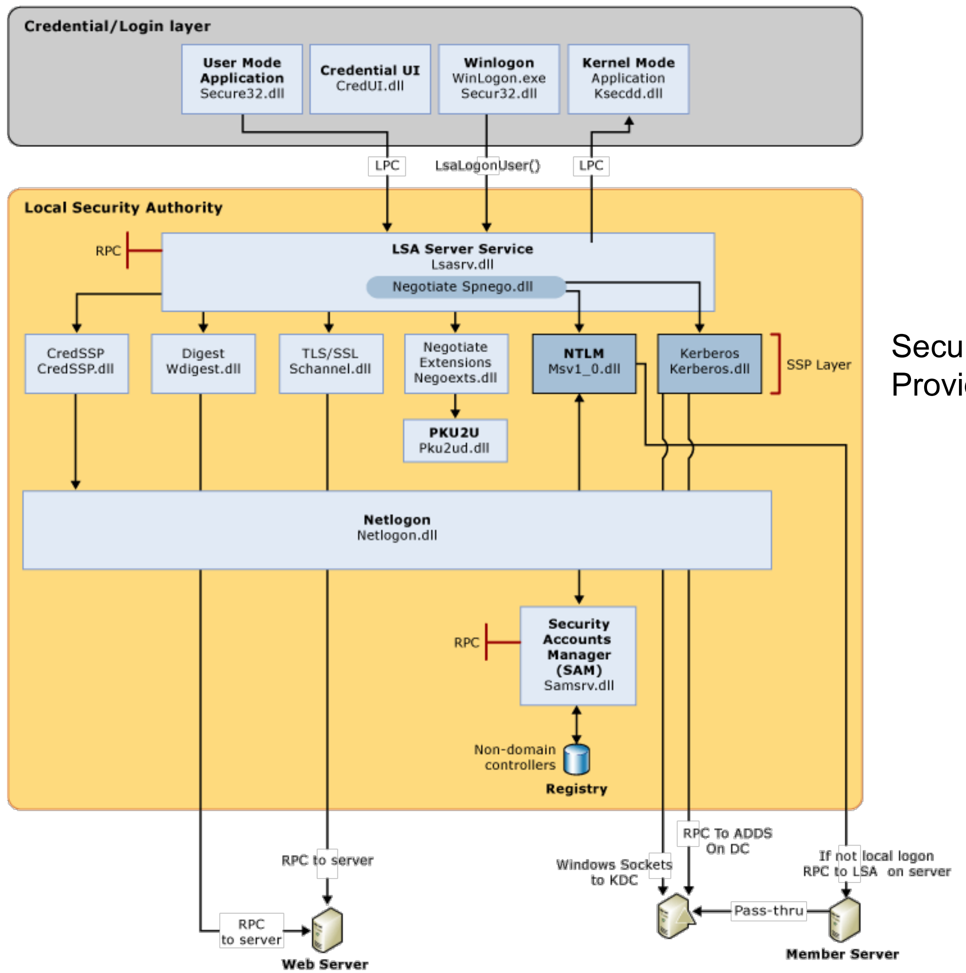
```
        if (response != NULL) {
                if (error == PAM_SUCCESS) {
                        password = response[0].resp;
                } else {
                        free(response->resp);
                }
        } else {
                return PAM_AUTH_ERR;
        }
        /* Check the login */
        error = authenticate((char*)username, (char*)password);
        free(password);
        /* Return result */
        return error == 1 ? PAM_SUCCESS : PAM_AUTH_ERR;
}

PAM_EXTERN int pam_sm_setcred(pam_handle_t *pamh, int flags,
                              int argc, const char **argv)
{
  return PAM_SUCCESS;
}
```

Windows LSA and SAM

- The **Local Security Authority (LSA)** manages interactive logons into the system
- The **Windows Security Account Manager** (SAM) manages user's password hashes
  - can be used to authenticate local and remote users
  - password hashes are stored in a Windows registry *hive* (a group of keys that has set of supporting files containing backups of its data, in this case `%SystemRoot%/system32/config/SAM`)
- Passwords are never transmitted in plain-text, but hashed using LM hash (*weak*), NTLM hash, AES key or Digest
- When logging in the LSA relies on SAM to check user credentials and obtain access token for the user

Windows authentication



Security Service Providers

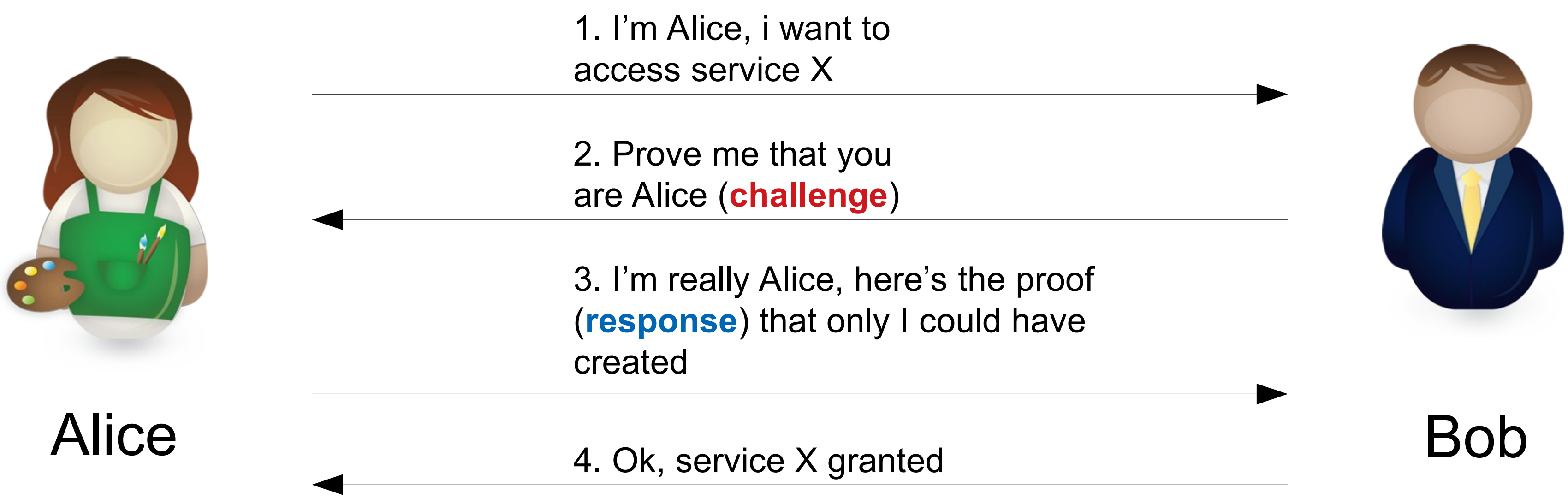
Source: Microsoft Technet, Credentials Management in Windows Authentication
https://technet.microsoft.com/en-us/library/dn169014(v=ws.10).aspx

Windows network authentication and Hashes

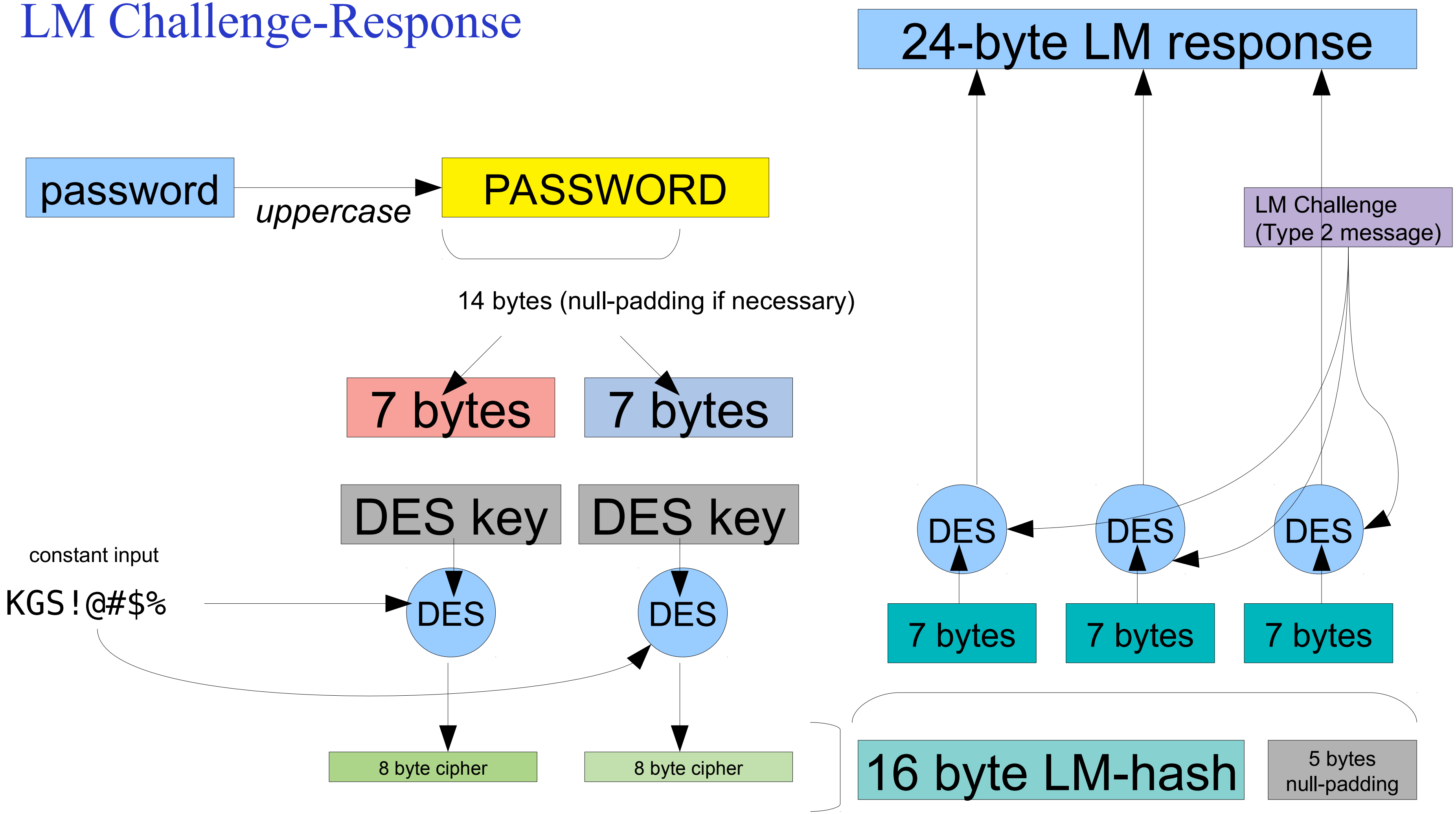
- **Windows Challenge/Response** (**NTLM**) is an authentication protocol for Windows-computer networks and for stand-alone systems
  - the user name, domain name and password hash are used to authenticate a user **without sending the password over the network**
  - During its lifetime Windows implemented several different hashing algoritms and network authentication methods:
    - LAN Manager (LM) → uses the **LM hash**
    - Windows NT (NTLM)
    - NTLM version 2

      (pre-Active Directory)
      Uses the **NT hash**, a.k.a Unicode hash
    - LDAP
    - Kerberos (v5, default since Windows XP, Windows Server 2000)

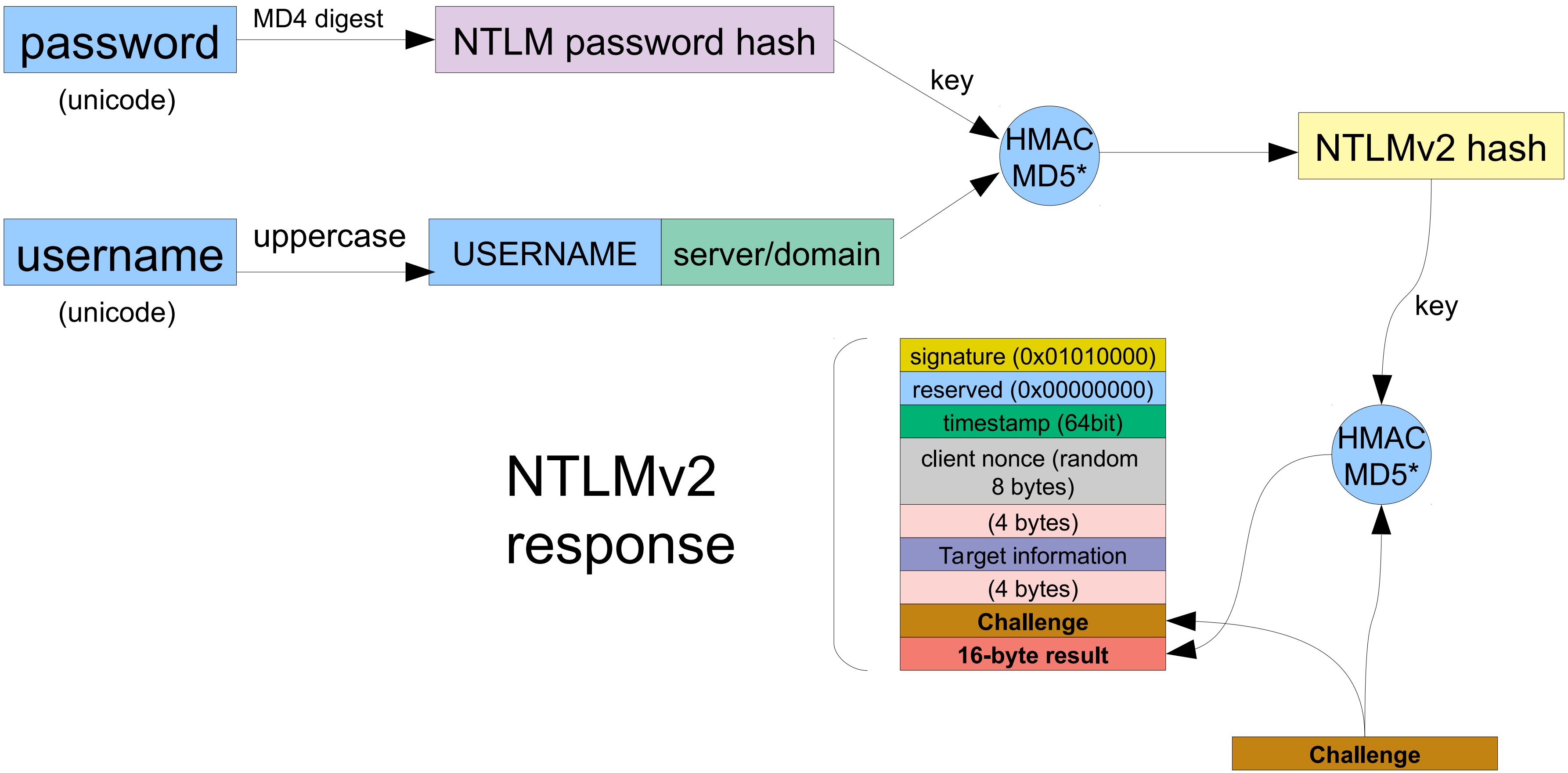
Challenge-Response Protocol



LM Challenge-Response



NTLM (v2) Challenge-Response

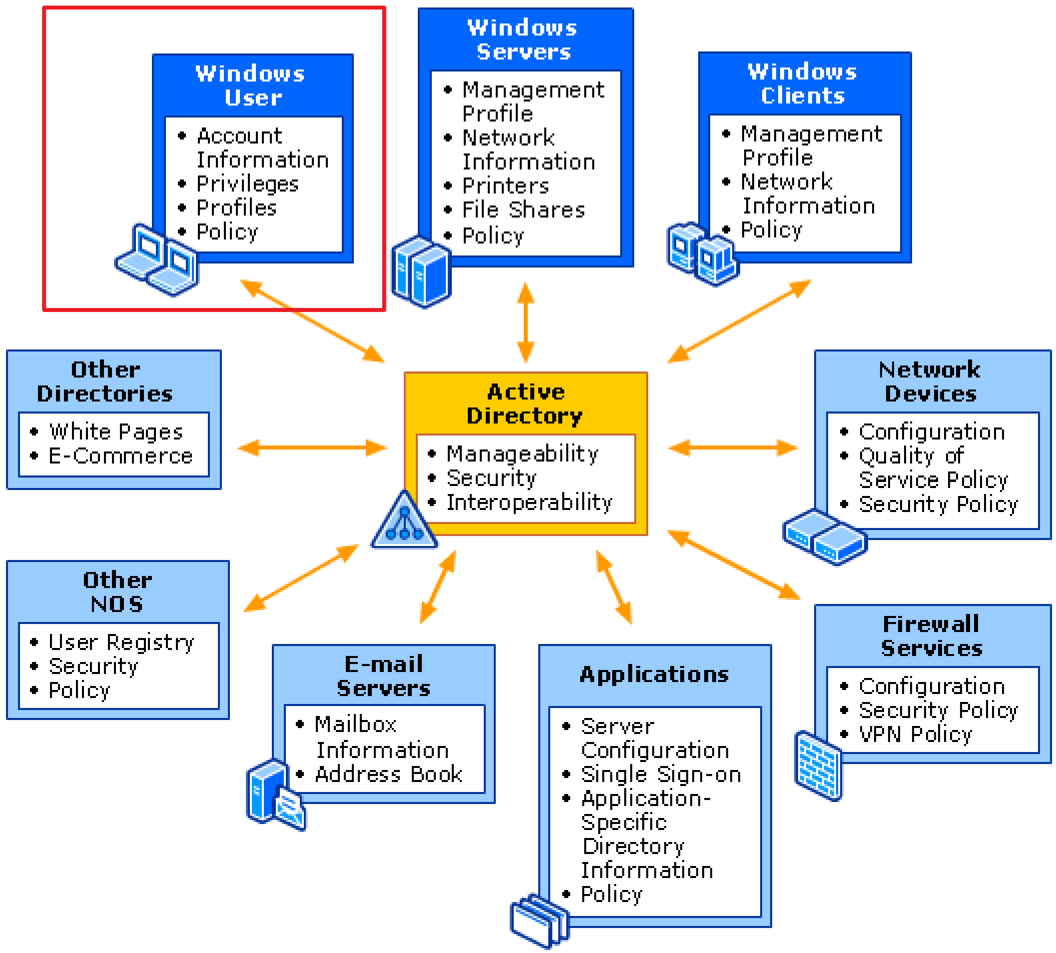


* HMAC: Keyed-Hashing for Message Authentication (http://www.ietf.org/rfc/rfc2104.txt)

Network authentication: LDAP and Active Directory

- A **directory service** is a system that stores, organises, and provides access to information (mapping *keys* to *values*)
- **LDAP** (Lightweight Directory Access Protocol) is a standard network protocol designed for querying directory services and accessing/modifying data
  - **Active Directory** is a proprietary implementation from Microsoft of a directory service to provide authentication, policy, and other services which can be accessed using the LDAP protocol
    - *open source alternative: OpenLDAP*

Network authentication: Active Directory services

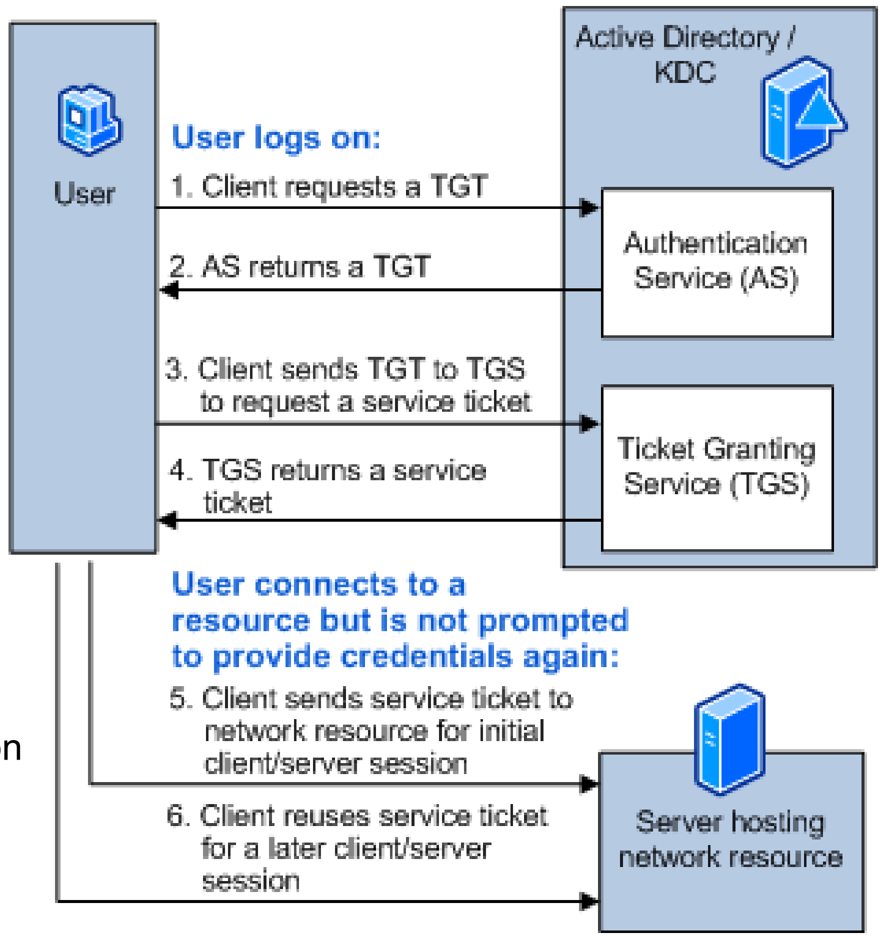


source: https://technet.microsoft.com/en-us/library/cc780036(v=ws.10).aspx

Network authentication: Kerberos

- **LDAP/Active Directory** authentication is centralized
  - passwords (hashes) are stored on a central server
  - … but users need to login with <u>every</u> service
- **Kerberos** is single sign-on (SSO)
  - users login once and get a token which can be used for multiple services on the nework

Network authentication: Kerberos authentication



TGT: Ticket-granting ticket
KDC: Kerberos Key Distribution Center

(Source: https://technet.microsoft.com/en-us/library/bb463152.aspx)

Token Based Authentication

- Physical or digital objects owned by a user can be used as **tokens** for authentication
- Examples:
  - magnetic cards
  - smart cards
  - bluetooth tokens
  - NFC tokens
- Tokens can be used to generate one-time passwords for two factor authentication
  - dynamic password generators (ex. RSA SecurID)
  - challenge-response system

Token Based Authentication

UBS Access Card

(source: https://www.ubs.com/content/dam/ubs/microsites/digital/images/ebanking-mobile/kartenleser.png)

RSA SecurID

(source: http://www.tokenguard.com/RSA-SecurID-SID700.asp)

PhotoTAN

(source: https://play.google.com/store/apps/details?id=ch.raiffeisen.phototan&hl=it)

Biometric Authentication

- Biometric authentication is based on unique **physical characteristics** of a human user:
  - fingerprints
  - hand geometry
  - facial characteristics
  - eye (iris / retinal) characteristics
  - signature
  - voice

Authorization / Access Control

- Beside an authentication mechanism there’s a need for **access control**
  - “what type of access is permitted?”
- Most common types of access control:
  - Discretionary Access Control (**DAC**)
  - Mandatory Access Control (**MAC**)
  - Role-based Access Control (**RBAC**)

Discretionary Access Control (DAC)

- Access is granted based on the identity of the requestor and on access rules
  - **discretionary** (“exercised at one's own discretion”): access rights to a resource given to a person can enable other person to access the same resource
    - DAC allows subjects the discretion to decide access rights on objects they own
  - enables fine-grained control and “least-privilege” access to resources

Discretionary Access Control (DAC): issues

- Letting users decide access permissions for the objects they own can be problematic if a systematic verification of system’s security principles is required
  - users with read privileges can create copies of a resource with different access rights
  - “who really has/had access to what?”

Example of DAC: Unix file permissions

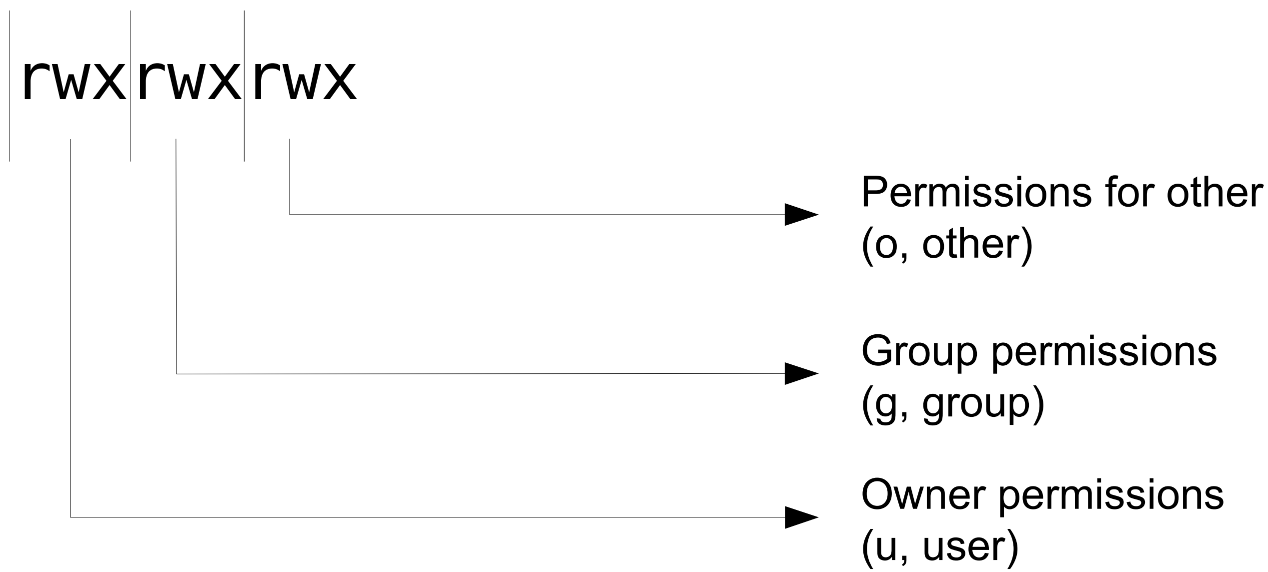
- Unix file permissions allow for **discretionary access control** (DAC):
  - the owner of an object (file, resource) can restrict access to objects (file, resources) based on the identity of other subjects and/or groups to which they belong
  - (or the other way) I can allow access to objects I own based on the identity of other users

Example of DAC: Unix user and groups access rights

- Each user is associated with a **username** (login name)
  - The system uses numerical identifiers (UID) to each username
- Users are members of one or more **groups**
  - Each group has a name (for example: Students) and a numerical ID (GID)
- It is possible to assign resource access privileges to single users or groups
  - for example, to allow read and write access to the file Accounting.txt only to users of the Accounting group
- The administrator (login name: root) has all the privileges on the filesystem

Example of DAC: Unix user and groups access rights

- Each file has an owner, a group and some access rights:
  - **r** : read, **w** : write, **x** : execute
  - only the owner (and the root user) can change access rights



- File permissions are stored within the i-node structure

Example of DAC: ACL (Access Control Lists)

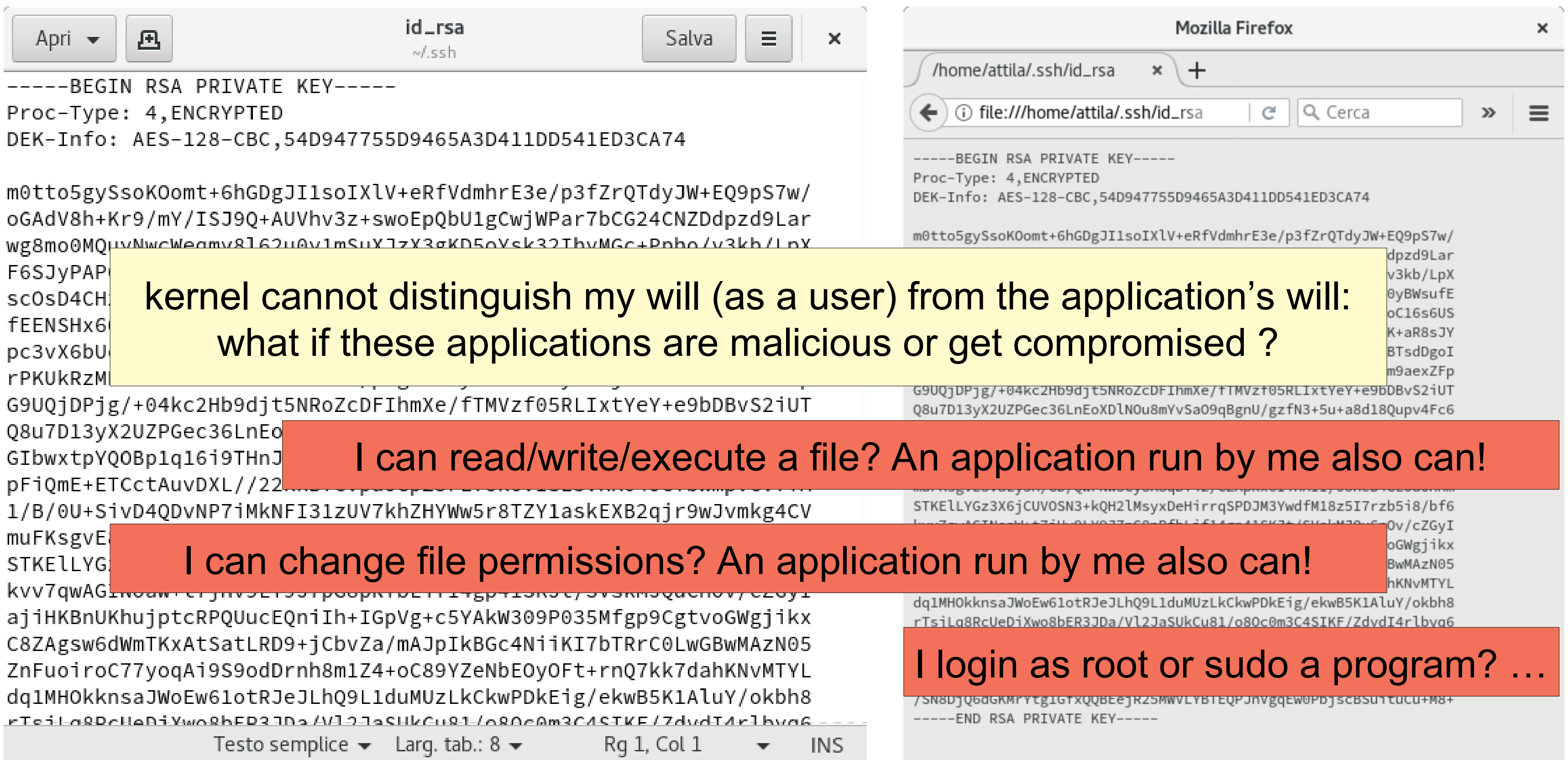
- **Access Control Lists (ACL)** enable more fine-grained control over access rights
  - per user access rights
  - per group access rights
- ACL can be inherited, hence we distiguish between the following types of access control rules:
  - **Explicit:** defined by the user
  - **Inherited:** inherited from the parent directory
    - Inheritance in Linux and MacOS is static: permissions are copied when the file is created. Inheritance in Windows is dynamic.
    - **Explicit permissions overwrite inherited permissions**
  - **Effective:** combination of explicit an inherited permissions

… but this is not always enough

```
attila@localhost:~> ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/attila/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/attila/.ssh/id_rsa.
Your public key has been saved in /home/attila/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:qFIPBSF3++rj9+8yfQ47LJbjd8tlWiM62tw3QIAjcdA attila@localhost.localdomain
The key's randomart image is:
+---[RSA 2048]----+
|  . +..o+..      |
|   o o o.E .     |
|      o . . .    |
|     . o     .   |
|    o . S   .    |
|   . + .     .   |
|  . . o    +....+|
|   . .. . X.B++B.|
|     .oo =+%=**..|
+----[SHA256]-----+
attila@localhost:~> ls -l ~/.ssh/
totale 12
-rw------- 1 attila users 1766 31 ott 11.37 id_rsa
-rw-r--r-- 1 attila users  410 31 ott 11.37 id_rsa.pub
-rw-r--r-- 1 attila users 1178 11 ott 17.00 known_hosts
```

who has access to my private keys?

… but this is not always enough



kernel cannot distinguish my will (as a user) from the application's will: what if these applications are malicious or get compromised ?

I can read/write/execute a file? An application run by me also can!

I can change file permissions? An application run by me also can!

I login as root or sudo a program? …