SAPIENZA
UNIVERSITÀ DI ROMA

"Sapienza" University of Rome
Faculty of Information Engineering, Informatics, and Statistics
Departmet of Computer Science

Cybersecurity

Lecture notes integrataed with the book "Computer Security: Principles and Practice", W. Stallings, L. Brown

Author
Simone Bianco

6 gennaio 2024

Indice

Information and Contacts

Personal notes and summaries collected as part of the *Cybersecurity* course offered by the degree in Computer Science of the University of Rome "La Sapienza".

Further information and notes can be found at the following link:
https://github.com/Exyss/university-notes. Anyone can feel free to report inaccuracies, improvements or requests through the Issues system provided by GitHub itself or by contacting the author privately:

- Email: **bianco.simone@outlook.it**
- LinkedIn: **Simone Bianco**

The notes are constantly being updated, so please check if the changes have already been made in the most recent version.

Suggested prerequisites:

Preventive learning of material related to the *Computer networks* and *Operating Systems* courses is recommended

Licence:

1

Introduction to Cybersecurity

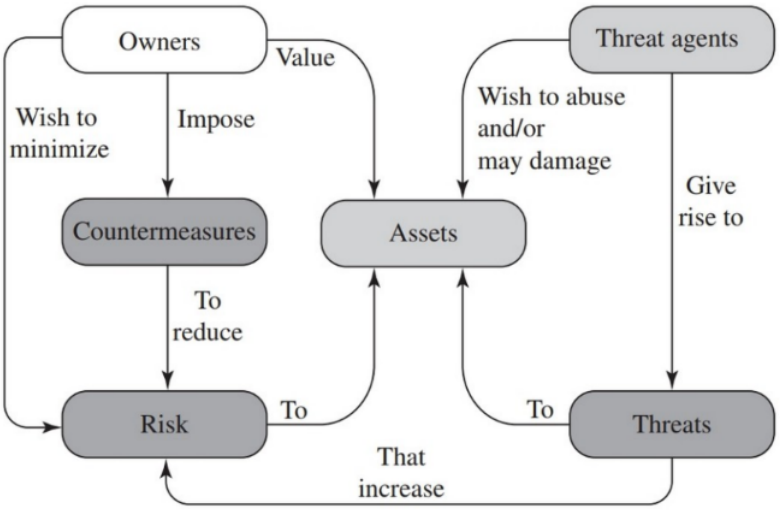
1.1 Fundamental concepts

The National Institute of Standards and Technology (NIST) defines **computer security** as the prevention of damage, protection and restoration of computers, electronic communications systems and services and any other type of digital structure.

In this course, we define **computer security** as measures and controls that ensure **confidentiality**, **integrity** and **availability** of information system assets including hardware, software and information being processed, stored, and communicate.

In order to talk about cybersecurity, first we have to give the following **essential definitions**:

- **Threat**: any circumstance or event with the potential to adversely impact organizational operations
- **Threat agent** (or *Adversary*): anyone who conducts or has the intent to conduct detrimental activities
- **Countermeasures**: a device or a technique that has the objective of impairing detrimental activities
- **Risk**: a measure of the extent to which an entity is exposed to a thread, such as the impact that would arise if an unaccounted event occurs and his likelihood of occurrences
- **Vulnerability**: weakness in an information system, internal controls, implementation, etc... that could be exploited or triggered by a thread source



Osservazione 1

The security of a system, application or protocol is always relative to the set of desired properties and the capabilities of the potential threat agent

Example:

- Standard file access permission in Linux or Windows systems are not effective against an adversary who can boot the system from a CD

Definition 1: Types of attacks

In order to distinguish between kinds of threats, we define the following **types of attack**:

- **Active attack**: an attempt to alter system resources or affect the operation.

  In particular, we establish four categories of active attack: **replay**, **masquerade**, **modification of messages** and **denial of service**

- **Passive attack**: an attempt to learn or make use of information from the system that does not effect the system resources

  In particular, we establish four categories of passive attack: **release of message contents** and **traffic analysis**

- **Inside attack**: initiated by an entity inside of the system's *security perimeter*, namely an **insider** who is authorized to access the system resources, using them in an unapproved way

- **Outside attack**: initiated by an entity outside of the system's *security perimeter* who is

1.2 Confidentiality, Integrity and Availability (CIA)

Definition 2: Confidentiality

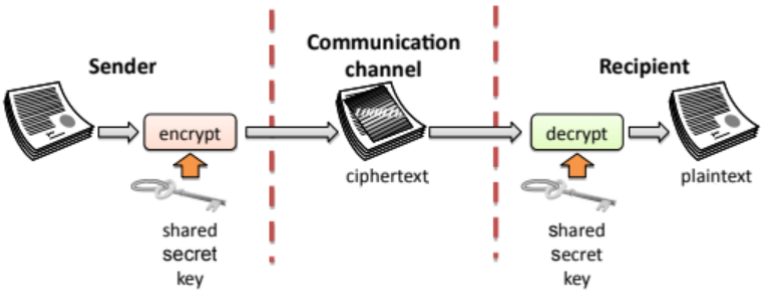
We define **confidentiality** as the avoidance of the unauthorized disclosure of information

Example:

- Confidentiality involves the protection of data, providing access for those who are allowed to see it while disallowing others from learning anything about its content

In order to **ensure** confidentiality is preserved, three main tools are used:

- **Encryption**: the transformation of information using a secret called *encryption key* in order to make the transformed information readable only by those who know another (or the same) secret, namely the *decryption key*



- **Access control**: rules and policies that limit access to confidential information to established people and/or systems
- **Authentication**: the determination of the identity or role that someone has, usually done through a number of different factors, such as something the person has, knows or is
- **Authorization**: the determination if a person or system is allowed to access resources based on an policy
- **Physical security**: the establishment of physical barriers to limit access to protected computational resources

Definition 3: Integrity

We define **integrity** has the property that something must not be altered in an unauthorized way

Examples:

- Integrity involves the use of backups, checksums, data correcting codes, etc...

Definition 4: Availability

We define **availability** as the property that something is accessible and modifiable in a timely fashion by those who are authorized to do so

Examples:

- Availability involves the use of physical protections and computational redundancies

The concepts of confidentiality, integrity and availability establish what is know as the **CIA security triad**. In order to be secure, a system should try to minimize the number of fallacies that conflict with the triad.

However, other concepts are used to describe the security of a system:

- **Authenticity**: the ability to determine that statements, policies and permission issued by a person are genuine.
- **Accountability**: the requirement for actions of an entity to be traced uniquely back to that same entity through the use of activity records
- **Anonymity**: the property that certain records or transactions are not to be attributable to any individual

1.3 Threat consequences and types

We can categorize events based on their ability to pose a threat on one or more concepts of the CIA triad or based on the type of attack implied by those events.

The first categorization can be reduces to the following types of events:

- **Unauthorized disclosure**: a circumstance or event whereby an entity gains access to data for which the entity is not authorized. This type of event is a threat to **confidentiality**
- **Deception**: a circumstance or event that may result in an authorized entity receiving false data and believing it to be true. This type of event is a thread to either **system integrity** or **data integrity**
- **Disruption**: a circumstance or event that interrupts or prevents the correct operation of system services and functions. This type of event is a threat to **availability** or **system integrity**
- **Usurpation**: a circumstance or event that results in control of system services or functions by an unauthorized entity. This type of event is a threat to **system integrity**

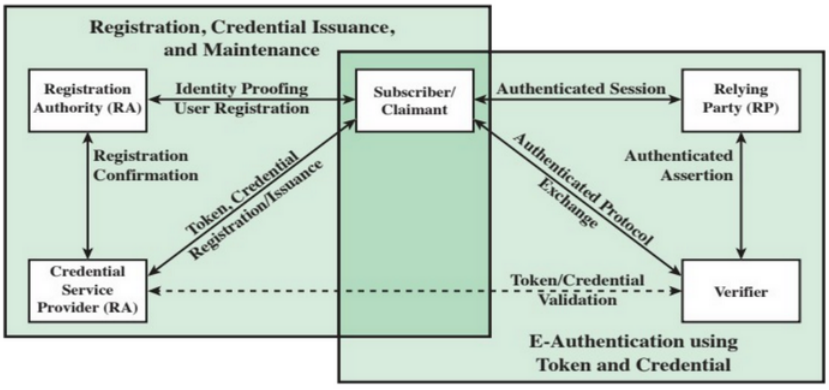
Instead, the second categorization can be reduces to the following types of attacks:

- **Interception**: the eavesdropping of information intended for someone else during its transmission over a communication channel
- **Falsification**: unauthorized modification of information, such as the *man-in-the-middle attack*, where a network stream is intercepted, modified and retransmitted to the original receiver
- **Denial of service (DoS)**: the obstruction or degradation of data service and/or information access
- **Masquerading**: the fabrication of information that is supposed to be from someone who is not actually the author
- **Repudiation**: the denial of commitment or data reception, such as the attempt to back out of a contract or protocol that requires the different partied to provide receipts acknowledging that data has been received
- **Inference** (or *correlation/traceback*): the integration of multiple data sources and information flows to determine the source of a particular data stream or piece of information

1.4 Authentication

As we already discussed, authentication can be described as the process of establishing confidence in the user identities that are presented electronically to an information system through the use of:

- Something the individual **knows**, such as a password or a PIN
- Something the individual **possesses**, such as a token or a key card
- Something the individual **is**, such as biometrics (fingerprints, iris, face, ...)
- Something the individual **does**, such as dynamic biometrics (handwriting, voice pattern, ...)



The use of more than one of these authentication means is called **multifactor authentication**, ensuring greater security as the number of methods used increases.

One of the most common means of authentication is the use of **passwords**. Usually, the user provides a name and a password, which then get compared by the system with the ones stored in their memory. The **user ID** determines that the user is authorized to access the system and the his privileges.

Definition 5: Hash function

An **hash function** is a one-way-function (meaning that it irreversible) capable of converting a string of plain text into an incomprehensible string of text of fixed length called **hash**

Since they are impossible to reverse, the best way to store passwords is through the use of **hash functions**. A good hash function must be capable of being efficient to compute while also being able to minimize the possibility of two string **colliding** into the same hash.

Definition 6: Salt

We define as **salt** a random string fed as an additional input to an hashing function by getting attached to the original input before being hashed.

The use of salts ensures that the input becomes sufficiently large, making the output more secure

Example:

- Modern UNIX systems store passwords by looping 1000 iterations of MD5 hash function with a salt of up to 48 bits, producing a 128 bit hash value

By **storing the hashed password**, one can check if the given password is correct simply by hashing it and then check if it matches the stored hash.

Many programs are used to **crack password** by exploiting the fact that people usually choose easily guessable password and/or short passwords, making it easy to **brute force** through the use of a cracking software:

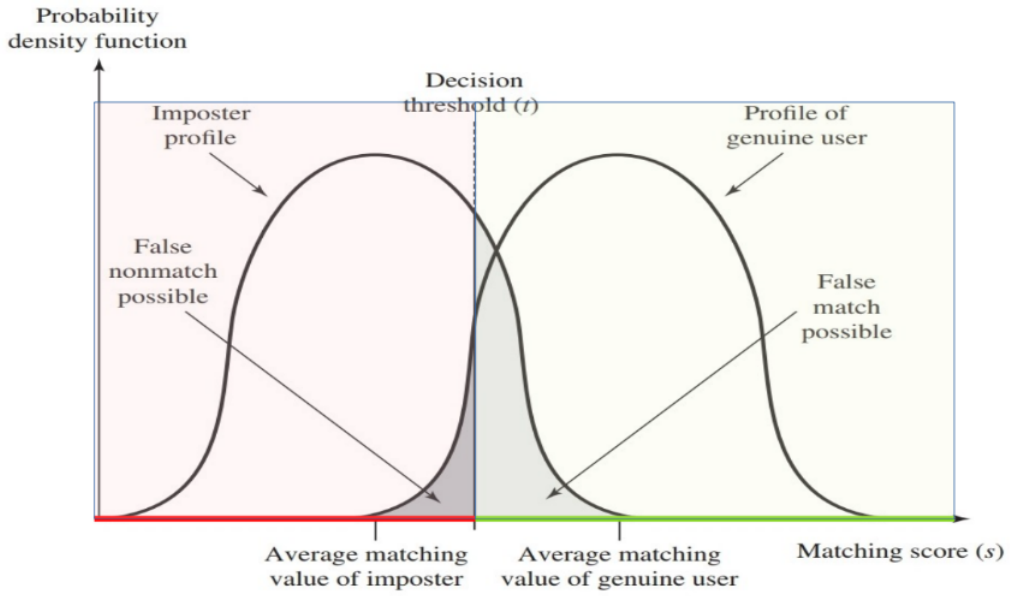
- **Dictionary attacks** are based on a large list of possible passwords, testing them one by one. Each password must be hashed using each different salt value and then compared to the stored hash values.
- **Rainbow table attacks** are based on pre-computed enormous tables of hash values fro all salts. This attack can be countered by using a sufficiently large salt value and a sufficiently large hash length

Definition 7: Token

We define as **token** a small string of text able to identify a user or an entity

Common examples of tokens include barcodes, magnetic stripe cards, smart tokens and smart cards realized through the use of RFID technology.

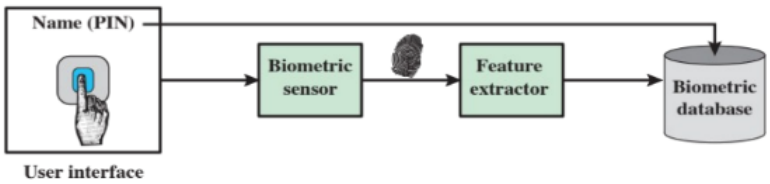
Less common examples of tokens include biometrics: the data gets read and then converted to a *reference vector*, which then gets compared to the stored one through the use of matching techniques based on *similarity* (since a perfect copy is never possible to replicate).



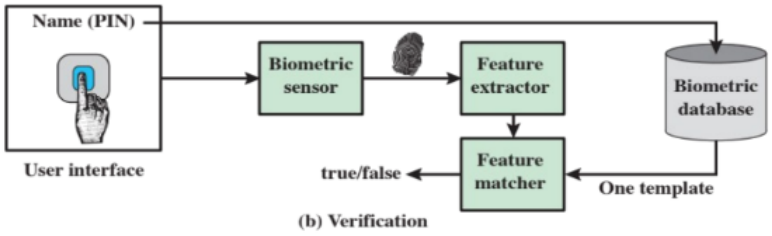
Biometrics such as voice and face recognition are usually low cost with low accuracy, while biometrics such as iris and fingerprint scanning are medium to high cost while also being pretty accurate.

Biometric authentication systems usually involve one or more of the following three types of operations:

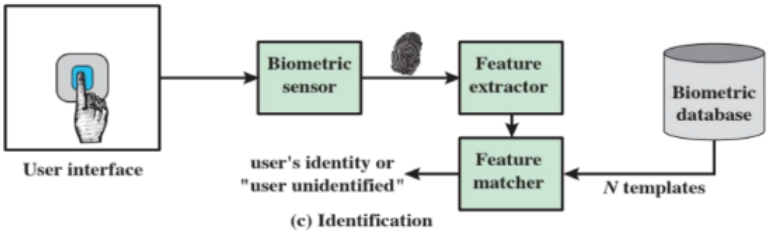
- **Enrollment**: the user registers his biometric data through the use of a PIN and a scanner



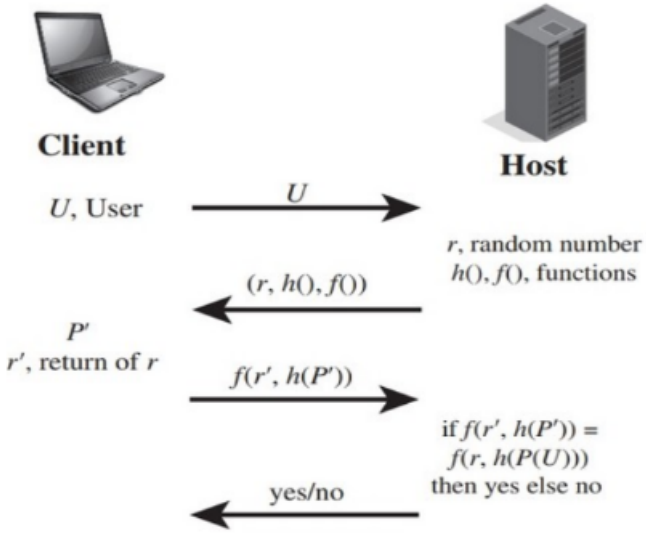
- **Verification**: the user gets recognized by giving the registered PIN and his biometric data by matching. Requires a previous registration and one sample of the user's biometric data



- **Identification**: the user gets identified by giving only his biometric data. Requires a previous registration and a chosen amount of samples of the user's biometric data



Modern systems are also able to do **remote user authentication** over a network, the Internet or more complex communication links. While being convenient, this types of authentication include **additional security threats** such as eavesdropping, password capturing and repli attacks. To avoid this threats, they generally rely on some form of a challenge-response protocol.



1.5 Access Control

Definition 8: Access Control

We define **access control** as the process by which use of system resources is regulated according to a security policy and is permitted only by authorized entities

One of the main access control models is the **Discretional Access Control (DAC)**, which controls access based on the identity of the requestor and on access rules stating what requestors are allowed and not allowed to do. This is achieved through the use of a scheme in which an entity may be granted access rights that permit the entity, by its own volition, to enable another entity to access some resource.

Common ways to implement the DAC model include:

- **Access Control Matrix**: one dimension identifies subjects asking data access to the resources (the users), while the other dimension identifies the objects that may be accesses. Each entry of the matrix indicates the access rights of the associated subject to the associated object. An empty entry defaults to no access right granted

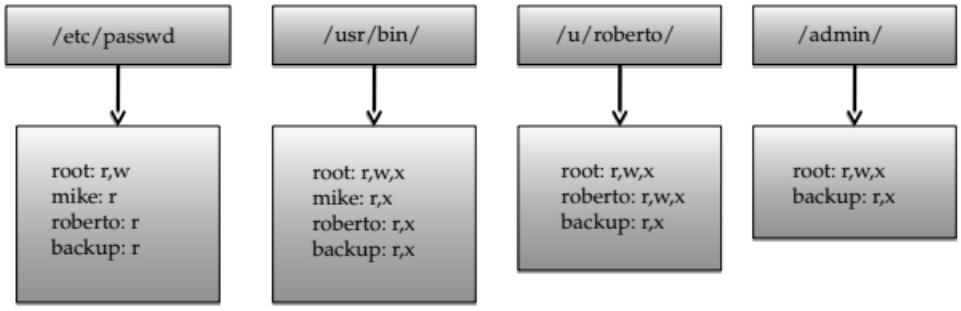
	/etc/passwd	/usr/bin/	/u/roberto/	/admin/
root	read, write	read, write, exec	read, write, exec	read, write, exec
mike	read	read, exec		
roberto	read	read, exec	read, write, exec	
backup	read	read, exec	read, exec	read, exec
...	...	...	...	...

- **Extended Access Control Matrix**: considers the ability of a subject to create another subject and to have "owner" access rights to that subject. Can be used to define a *hierarchy of subjects*

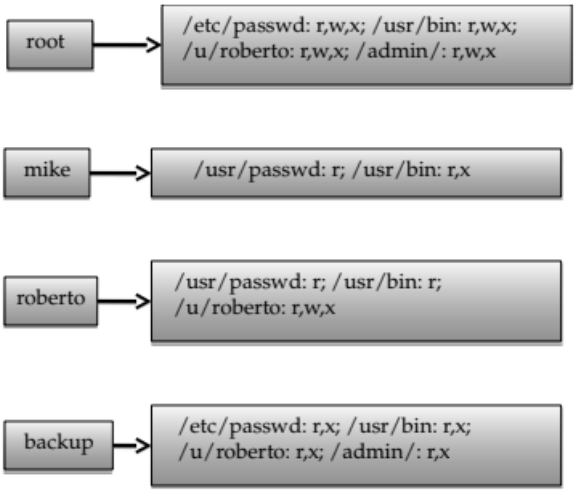
		OBJECTS								
		subjects			files		processes		disk drives	
		S_1	S_2	S_3	F_1	F_2	P_1	P_2	D_1	D_2
SUBJECTS	S_1	control	owner	owner control	read *	read owner	wakeup	wakeup	seek	owner
	S_2		control		write *	execute			owner	seek *
	S_3			control		write	stop			

* - copy flag set

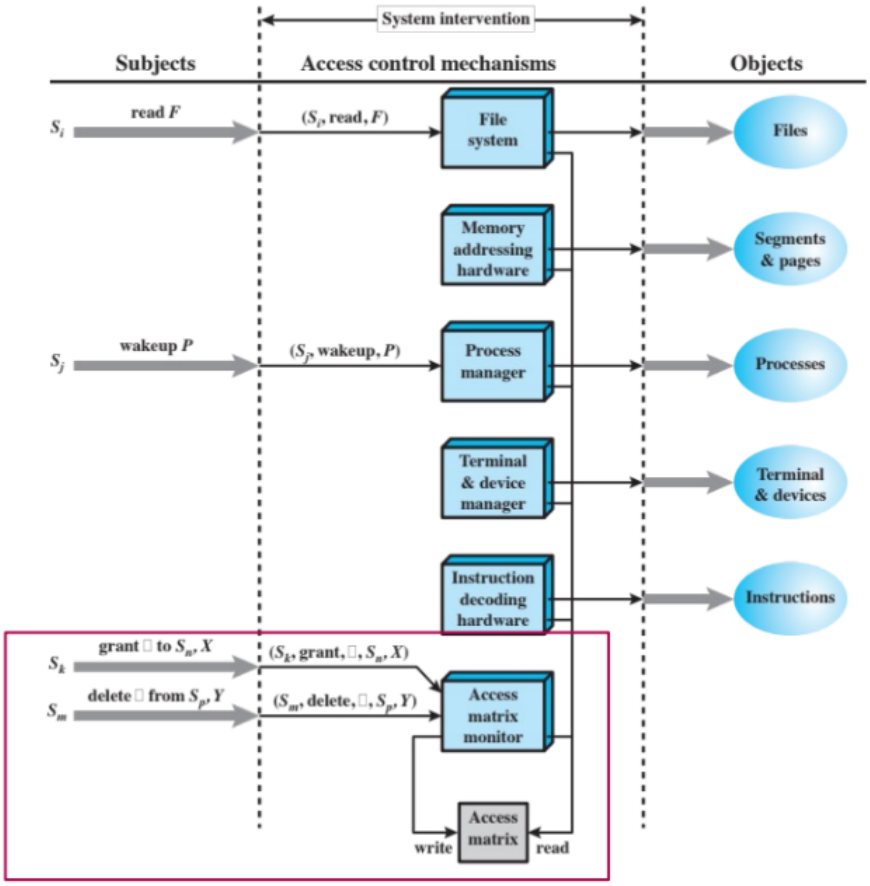
- **Access Control List**: each object has an associated list which enumerates all the subjects that have access rights for that object, specifying the granted rights



- **Capability List**: each subject has an associated list which enumerates all the objects and the access rights granted for each one of them to that subject (same as ACL but objects and subjects are swapped)



- **UNIX File Access Control**: a minimal ACL version, where each objects is identified by the owner (User ID), the primary group (Group ID) and 12 protection bits (Read, Write and Execute bits for object owner, group members and all other users)
- **Access Control Function**: every access by a subject to an object is mediated by the controller for that object. The controller's decision is based on the current contents of the matrix. Certain subjects have to authority to make specific changes to the access matrix



Another way to manage access control is through the **Mandatory Access Control (MAC)** model, where each subject and each object gets assigned a security class, forming a strict hierarchy and being referred to as **security levels**. A subject is said to have a **security clearance** of a given level, while an object is said to have a **security classification** of a given level.

Through **Multilevel Security (MLS)**, the MAC model defines four access modes:

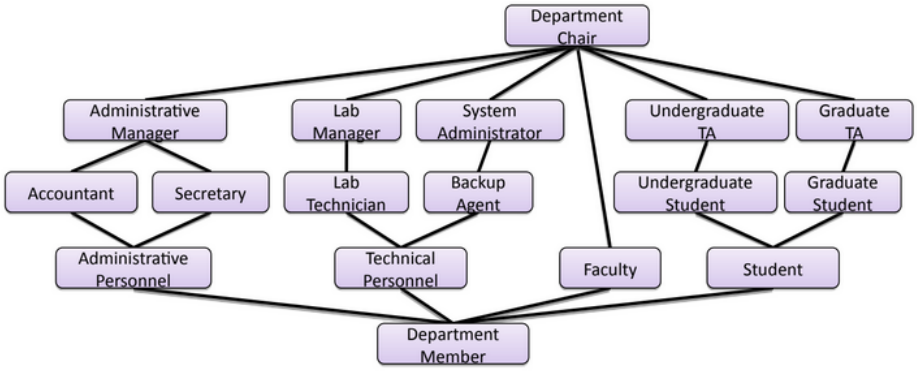
- **read**: the subject is granted read access to the object
- **append**: the subject is granted write access to the object
- **write**: the subject is granted read and write access to the object
- **execute**: the subject is granted the ability to execute the object

Confidentiality is achieved if a subject at high level may not convey information to a subject at lower level, unless that flow accurately reflects the will of an authorized user as revealed by an authorized declassification:

- **No read up**: a subject can only read an object of less or equal security level
- **No write down**: a subject can only write into an object of greater or equal security level

1.5.1 Role-based Access Control

A more advanced model of access control is **Role-based Access Control (RBAC)**, where access rights are defined on roles instead of directly on subjects, allowing to descrive organizational access control *policies* based on job functions.



A user's permissions are determined by its roles rather by identity or clearance, increasing flexibility and scalability in policy administration. Each role is assigned to users through the use of an **User Assignment table**, while each access right gets assigned to roles through the use of a **Permission Assignment table**.

Example:

- Consider the following user and permission assignments:

User	Role
Alice	Radiologist
Alice	GP
Bob	GP
Charlie	Radiologist
David	Nurse

Role	Permission
Nurse	(read, prescription)
GP	(read, prescription)
GP	(write, prescription)
GP	(read, history)
Radiologist	(read, history)
Radiologist	(insert, image scane)

- The corresponding access matrix is defined as:

	Prescription	History	Image scan
Alice	read, write	read	insert
Bob	read, write	read	insert
Charlie		read	insert
David	read		

Through the years, the standard RBAC model, namely the **RBAC0** model, has evolved into four sub-models. The first sub-model is **RBAC1**, where roles are structured in an hierarchy, making lower level roles inherit access rights of their related superior level, reflecting an organization's role structure. Formally, we say that $x \leq y$ if and only if x is a specialization of y. If $x \leq y$, then the role x inherits permissions of role y. The $\leq$ relationship forms a *partial order* on the defined roles.

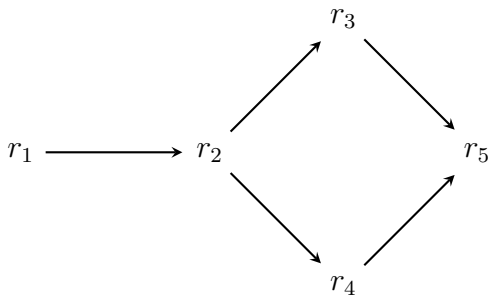
Example:

- Consider the following user and permission assignments:

User	Role
u_1	r_2
u_2	r_3
u_3	r_4
u_4	r_5

Role	Permission
r_1	p_1
r_2	p_2
r_3	p_3
r_4	p_4
r_5	p_5

- Consider now the following hierarchy of roles:



- The corresponding access matrix is defines as:

	$\mathbf{p_1}$	$\mathbf{p_2}$	$\mathbf{p_3}$	$\mathbf{p_4}$	$\mathbf{p_5}$
$\mathbf{u_1}$	×	×			
$\mathbf{u_2}$	×	×	×		
$\mathbf{u_3}$	×	×		×	
$\mathbf{u_4}$	×	×	×	×	×

The second sub-model is **RBAC2**, where role hierarchy is replaced with the definition of **constraints**, providing means of adapting RBAC to the specifics of administrative and security policies of an organization. Constraints are defined through **relationships** among roles or a **condition** related to roles:

- **Mutually exclusive roles**: a user or permission can only be assigned to one role of the defined mutually exclusive set
- **Cardinality**: setting a maximum number of assignable roles
- **Prerequisite roles**: dictates that a user can only be assigned to a particular role only if it is already assigned to some other specified role

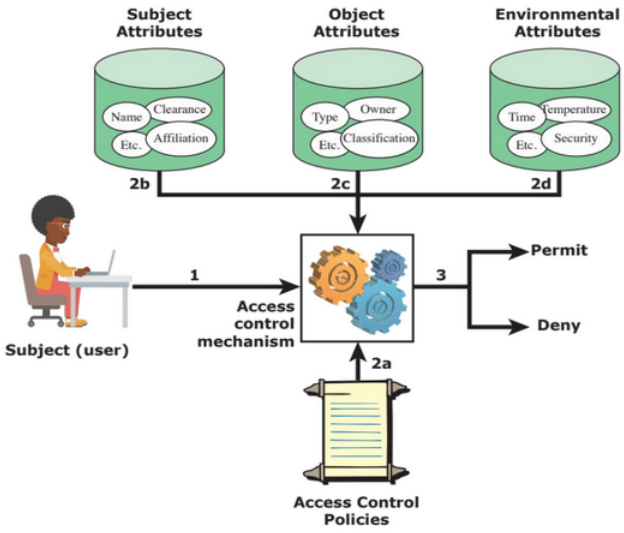
The last sub-model is **RBAC3**, a combination of RBAC1 and RBAC2 (role hierarchy and constraints).

1.5.2 Attribute-based Access Control

Another type of advanced access control model is **Attribute-based Access Control (ABAC)**, which uses attributes to define authorizations that express conditions on properties of both the resource and the subject. The main obstacle to the adoption of this model in real systems has been a concern about the performance impact of evaluating predicates on both resource and user properties for each access.

There are three types of usable attributes:

- **Subject attributes**: a subject is an active entity that causes information to flow among objects or changes the system state. Attributes define the identity and characteristics of the subject
- **Object attributes**: an object is a passive information system-related entity containing or receiving information. Objects have attributes that can be leverages to make access control decisions
- **Environmental attributes**: describe the operational technical and even situational environment or context in which the information access occurs. These attributes have so far been largely ignored in most access control policies



Definition 9: Policy

A **policy** is a set of rules and relationships that govern allowable behavior within an organization, based on the privileges of subjects and how resources or objects are to be protected under which environment conditions

Example:

- Consider the following policy:
  - Movies rated R can only be accessed by users of age 17+

  - Movies rated PG13 can only be accessed by users of age 13+
  - Movies rated G can only be accessed by everyone
- The following function for the role $R1$ determines if a user u can access the movie m with the given environment values e:

$$\begin{aligned} \text{R1: can\_access}(u, m, e) \leftarrow\ & (\texttt{Age}(u) \geq 17 \wedge \texttt{Rating}(m) \in \{\text{R}, \text{PG13}, \text{G}\}) \vee \\ & (\texttt{Age}(u) \geq 13 \wedge \texttt{Rating}(m) \in \{\text{PG13}, \text{G}\}) \vee \\ & (\texttt{Age}(u) < 13 \wedge \texttt{Rating}(m) \in \{\text{G}\}) \end{aligned}$$

In the RBAC model, as the number of attributes increases to accomodate finer-grained policies, the number of roles and permissions grows exponentially. The ABAC model, instead, deals with additional attributes in an efficient way.

Example:

- Suppose that:
  - Movies are classified as either New Release or Old Release, based on release date compared to the current date
  - Users are classified as Premium User and Regular User, based the fee they pay
  - The policy states that only premium users can view new movies
- In the RBAC model, we have to double the number of roles and the number of separate permissions in order to distinguish each user by age and fee
- In the ABAC model, we can simply define the following functions for the roles $R1, R2$ and $R3$:

$$\begin{aligned} \text{R1:can\_access}(u, m, e) \leftarrow\ & (\texttt{Age}(u) \geq 17 \wedge \texttt{Rating}(m) \in \{\text{R}, \text{PG13}, \text{G}\}) \vee \\ & (\texttt{Age}(u) \geq 13 \wedge \texttt{Rating}(m) \in \{\text{PG13}, \text{G}\}) \vee \\ & (\texttt{Age}(u) < 13 \wedge \texttt{Rating}(m) \in \{\text{G}\}) \end{aligned}$$

$$\begin{aligned} \text{R2:can\_access}(u, m, e) \leftarrow\ & (\texttt{MembershipType}(u) = \text{Premium}) \vee \\ & (\texttt{MembershipType}(u) = \text{Regular} \wedge \\ & \texttt{MovieType}(m) = \text{OldRelease}) \end{aligned}$$

$$\text{R3:can\_access}(u, m, e) \leftarrow \text{R1:can\_access}(u, m, e) \wedge \text{R2:can\_access}(u, m, e)$$

2

Common vulnerabilities

2.1 Malware

Definition 10: Malware

We define **malware (malicious software)** as any program that is inserted into a system with the intent of compromising the confidentiality, integrity or availability of the victim's data, applications or operating system or otherwise annoying or disrupting the victim

Malware gets usually classified by two major characteristics:

- **Propagation mechanism**: how the malware spreads in order to reach the desired targets, including:
  - Infection of existing content by viruses that is subsequently spread to other systems
  - Exploit of software vulnerabilities by worms or drive-by-downloads to allow malware to replicate
  - Social engineering attacks that convince users to bypass security mechanisms to install Trojans or to respond to phishing attacks
- **Payload actions**: the actual infective actions performed by the malware once the target gets reached, including:
  - Corruption of system or data files
  - Theft of service, such as making the system a "zombie" agent of attacks as part of a botnet
  - Theft of information from the system, such as keylogging
  - Stealthing, such as hiding its presence on the system

Initially, the development and deployment of malware required considerable technical skill by software authors. Through the years, virus-creation **toolkits** where developed, followed by even more general attack kits. These types of toolkits are often known as **crimeware**. They include a variety of propagation mechanisms ad payload modules that even novices can deploy. Attack variants that can be generated by attackers using these toolkits creates a significant problem for system defenses.

Another significant malware development turning point is the change from attackers being individuals often motivated to demonstrate their technical competence to their peers to more organized and dangerous attack sources, such as politically motivated attackers, organized crime, etc...

This has significantly changed the resources available and motivation behind the rise of malware and has led to development of a large underground economy involving the sale of attack kits, access to compromised hosts and to stolen information

Definition 11: Advanced Persistent Threat

We define as **Advanced Persistent Threat (APT)** the well-resourced, persistent application of a wide variety of intrusion technologies and malware to selected targets

Typically, APT get attributed to state-sponsored organizations and criminal enterprises. They differ from other types of attack by their **careful target selection** and **stealthy intrusion** efforts over **extended periods**.

The aim of these types of attack varies from theft of intellectual property or security and infrastructure related data to the physical disruption of infrastructure. Once initial access has been gained, further range of attack tools are used to maintain and extend their access. APT attacks characteristics can be reduced to the following three points:

- **Advanced**: they use a wide variety of intrusion technologies and malware including the development of custom malware if required. The individual components may not necessarily be technically advanced but are carefully selected to suit the chosen target
- **Persistent**: they include determined application of the attacks over an extended period against the chosen target in order to maximize the chance of success. A variety of attacks may be progressively applied until the target is compromised
- **Threats**: they pose threats to the selected targets as a result of the organized, capable and well-funded attackers intent to compromise the specifically chosen targets. The active involvement of people in the process greatly raises the threat level from that due to automated attacks tools and also the likelihood of successful attacks

2.1.1 Types of malware

Definition 12: Virus

A **virus** is a piece of software that infects other programs by modifying them to include a copy of the virus itself, making it capable of replicating itself and spreading through network environments.

They are made of three major components:

- **Infection mechanism** (or *Infection vector*): means by which the virus propagates
- **Trigger** (or *Logic Bomb*): events and conditions that determines when the payload is activated or delivered
- **Payload**: the actual malevolent actions of the virus

When a virus gets attached to an executable program, it can do anything that the program is permitted to do. Usually, they execute secretly when the host program is run:

- **Dormant phase**: the virus is idle and will eventually be activated by some event. Not all viruses have this stage
- **Triggering phase**: the virus gets activated to perform the functions for which it was intended
- **Propagation phase**: the virus places a copy of itself into other programs or into certain system areas on the disk. Each infected program will contain a clone of the virus with will itself enter a propagation phase. The propagated virus may not be identical to the original spreader
- **Execution phase**: the virus executes the payload, which may be harmless or damaging

A less known but more common type of viruses are **macro viruses**, viruses attached to documents that use **macro programming** (which usually are simple scripts) capabilities of the document's application to execute and propagate, infecting scripting code used to support active content in a variety of user document types. They are platform independent, easy to write and can rapidly spread.

Example:

- Microsoft Office Word documents can contain some macros to define advanced operations on the document which can be exploited to insert a macro virus

Viruses area **classified** by two characteristics:

- **Classification by target**:
  - **Boot sector infector**: the virus infects a master boot record or simple boot record, spreading when the system gets booted from the disk containing the virus

  - **File infector**: the virus infects files that the operating system or shell considers to be executable
  - **Macro virus**: the virus infects files with macro or scripting code that is interpreted by an application
  - **Multipartite virus**: the virus infects in multiple ways
- **Classification by concealment strategy**:
  - **Encrypted virus**: a portion of the virus creates a random encryption key and encrypts the remainder of the virus
  - **Stealth virus**: a form of virus explicitly designed to hide itself from detection by anti-virus software
  - **Polymorphic virus**: a virus that mutates with every infection
  - **Metamorphic virus**: a virus that mutates and rewrites itself completely at each iteration and may change behavior as well as appearance

Definition 13: Worm

A **worm** is a program that actively seeks out more machines to infect and each infected machine serves as an automated launching pad for attacks on other machines

Osservazione 2: Viruses vs Worms

Worms are similar to a viruses, but they do not modify the host program. They simply replicate themselves more and more to cause slow down the computer system. Also, worms can be controlled by remote, while viruses are independent once deployed

Worms exploit software vulnerabilities ina client or server programs, using network connections to spread from system to system, usually carrying some form of payload.

Worm replication usually happens through one of these capabilities:

- **E-mail or instant messenger facility**: the worm sends an attachment containing a copy of itself to other systems
- **File sharing**: the worm creates a copy of itself or infects a file as a virus on removable media
- **Remote execution capability**: the worm executes a copy of itself on another system
- **Remove file access capability**: the worm uses a remove file access or transfer service to copy itself from one system to another
- **Remote login capability**: the worm logs onto a remote system as a user and then uses commands to copy itself from one system to the other

Worms discover their targets through the use of various methods:

- **Scanning**: the worm searches for other systems to infect on the network
- **Random**: each compromised host probes random addresses in the IP address space using different seeds
- **Hit-list**: the attacker compiles a long list of potential vulnerable machines, which then gets split into portions, each given to an already compromised host
- **Topological**: the attacker uses information contained on an infected victim machine to find more hosts to scan
- **Local subnet**: if a host gets infected behind a firewall, that host looks for targets in its own local network

Other common but less know types of malware include:

- **Drive-by-Downloads**:
  - They exploit browser and plugin vulnerabilities
  - The user views a webpage controlled by the attacker which contains code that exploits the bug to download and install malware on the system without the user's knowledge or consent
  - In most cases, the malware doesn't actively propagate but spreads only when the user visits the malicious web page
- **Watering-Hole attacks**:
  - A variant of drive-by-downloads used in highly targeted attacks
  - The attacker researches their intended victim to identify websites they are likely to visit, then scans these sites to identify those with vulnerabilities that allow their compromise, waiting for one of their intended victims to visit one of the compromised sites
  - Attack code may even be written so that it will only infect systems belonging to the target organization and take no action for other visitors of the site
- **Malvertising**:
  - The attacker places a malware on websites without actually compromising them, paying for advertisements that are highly likely to be placed on their intended target websites and incorporate malware in them
  - The code may be dynamically generated to either reduce the chance of detection or to only infect specific systems
- **Clickjacking** (or *User interface redress attack*):
  - The attacker can force the user to do a variety of thinks from adjusting the user's computer setting to unwittingly sending the user to websites that might have malicious code

  - Can be achieved by hiding a button under or over a legitimate button, making it difficult to be detected
  - Similarly, this technique can be user to hijack keystrokes: an user can be led to believe that they are typing into a legitimate textbox but are instead typing into an invisible frame controlled by the attacker

- **Ransomware**:
  - The system gets infected usually through a worm that encrypts a large number of files, demanding a payment to decrypt them
  - Tactics such as threatening to publish sensitive personal data or permanently destroy the encrypted data are sometimes used to increase the pressure on the victim to pay up

2.1.2 Types of malware payload

The simplest type of malware payload is plain **system corruption**, causing real-world damage such as damage to the physical equipment. The logic bomb code of this type of payload is set to "explode" when certain conditions are met. As an example, the *Chernobyl virus* was common virus set to rewrite the whole BIOS code after exploding, making the host completely unbootable.

Another type of payload is **attack agent bots**: the malware takes over another Internet attached computer and uses that computer to launch or manage attacks. After propagation, every host infected by these type of malware becomes part of a **botnet**, a collection of bots capable of acting in a coordinated manner.

Each botnet is controlled through a **remote control facility** (also called **Control & Command - C&C**). The presence of a C&C host is what distinguishes a simple worm from a bot: the first one propagates and activates itself, while the latter is initially controlled from some central facility.

Typical means of implementing the remote control facility is through an **IRC server**, where bots join a specific channel on this server, treating incoming messages as commands, or through the use of **peer-to-peer protocols** to avoid a single point of failure.

The most common type of payload widely known is **information theft**, usually divided into three subcategories:

- **Keyloggers**: the malware captures keystrokes to allow the attacker to monitor sensitive information
- **Spyware**: the malware captures data about the compromised machine, monitoring the activity, redirecting certain webpage requests to fake sites and dynamically modifying data exchanged between the browser and certain websites

- **Phishing**: the attacker exploits social engineering to leverage the user's trust by masquerading as communication from a trusted source. The name of this type of attack comes as a variant of the word *fishing* to mimic its general intention.

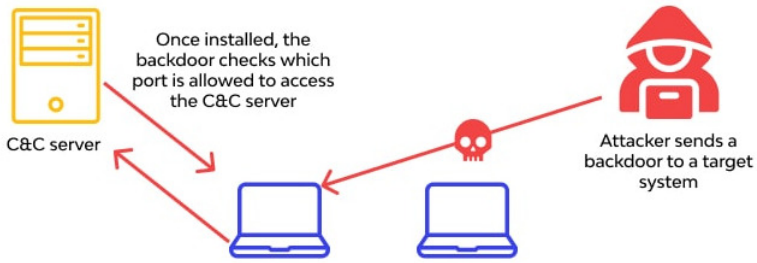
Example:

- The attacker includes a URL in a spam e-mail that links to a fake website that mimics the login page of a banking, social network or similar site
- The website suggests that urgent action is required by the user to authenticate their account, asking the user to input his credentials, stealing them

A more sophisticated type of phishing attacks is **spear-phishing**, where the recipients are carefully researched by the attacker. The e-mail is crafter to specifically suit its recipient, often quoting a range of private information known only people close to the victim to convince them of its authenticity.

The last type of malware payload consists in **stealthing** attacks. These types of attack are based on silent malware that is almost unperceivable by the user:

- **Backdoor stealthing**: the malware contains a secret entry point, allowing the attacker to gain access and bypass the security access procedures. They are difficult to implement in modern operating systems due to restrictive checks



- **Rootkit**: a set of hidden programs installed on a system to maintain covert access to that system. Hides by subverting the mechanisms that monitor and report on processes, files and registries of the computer, usually giving administrator privileges to the attacker

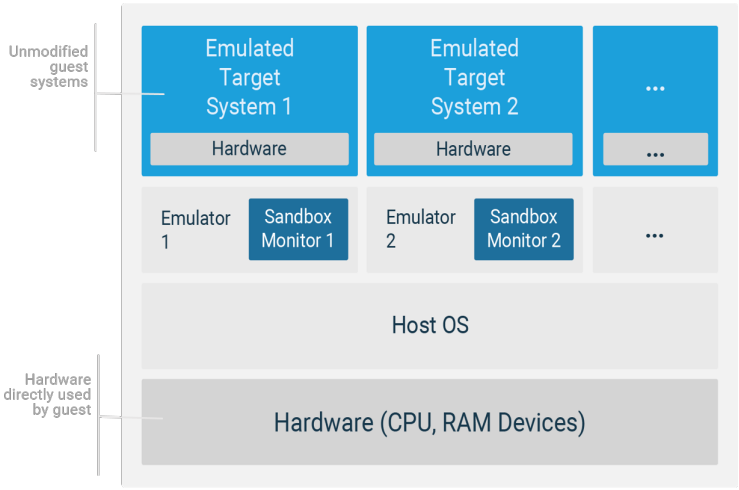
2.1.3 Malware countermeasure approaches

The ideal solution to any kind of malware is **prevention** by spreading awareness, suggesting the use of policies, vulnerability and threat mitigation. If prevention fails, **technical mechanisms** can be used to support detection, identification and removal of the malicious software. One of the most common and widely used mechanism is **anti-virus software**. Through the years, these type of software has been extensively improved:

- **Generation I - Scanners**: the software compares the signature of every file installed on the machine with every signature stored in his database. Limited to the detection of known malware
- **Generation II - Heuristic scanners**: the software uses heuristic rules to search for probable malware instances
- **Generation III - Activity traps**: memory-resident programs that identify malware by its actions rather than its structure in an infected program
- **Generation IV - Full-featured protection**: packages consisting of a variety of anti-virus techniques used in conjunction, including scanning and activity traps

New types of anti-virus software also use what is known as **sandbox analysis**, where the suspected malicious code gets emulated in a "sandbox" environment, usually a virtual machine, allowing the code to execute in a **controlled environment** where its behavior can be closely monitored without threatening the security of the real system.

However, since the emulation has to eventually stop and declare the software safe or unsafe, modern malware has **adapted** to just wait a sufficient amount of time before actually executing the payload, making the determination of the duration of each simulation the most difficult design issue with sandbox analysis.



Other recently developed techniques include **host-based behavior-blocking software**, which integrates with the operating system of the machine and monitors program behavior in real time for malicious actions, blocking potentially them before they have a chance to affect the system, and **perimeter scanning approaches**, where the anti-virus software is typically included in e-mail and web proxy services running on an organization's firewall.

2.2 Denial of Service (DoS)

Definition 14: Denial of Service (DoS)

We define as **Denial of Service (DoS)** any action that prevents or impairs the authorized use of networks, systems or applications by exhausting resources such as CPUs, memory, bandwidth and disk space

DoS attacks can be used to influence the availability of some service by degrading network bandwidth, system resources or application resources, typically by overloading and/or crashing the network handling software by sending a number of valid requests, each consuming significant resources.

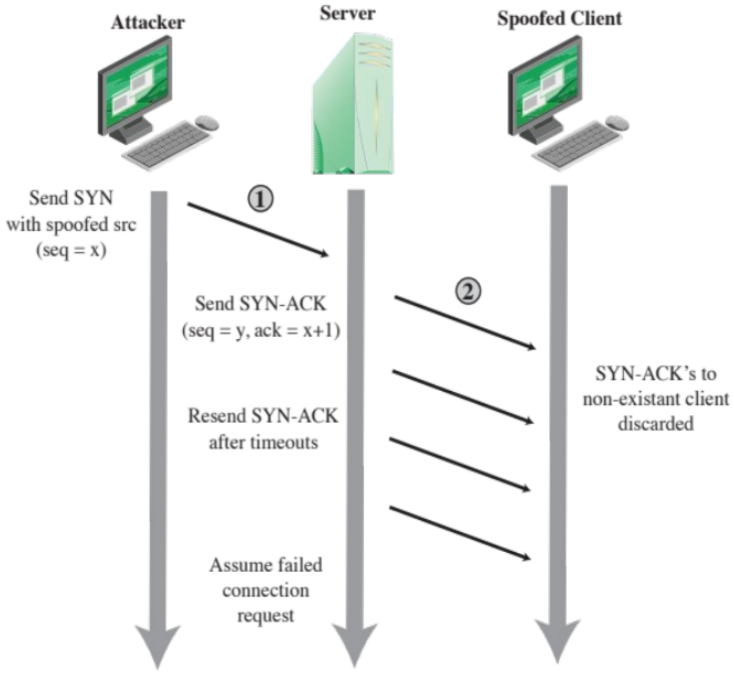
Classic DoS attacks are based on **flooding ping command** which aims to overwhelm the capacity of the network connection to the target organization. Traffic can be handled by higher capacity links on the path, but packets are discarded as capacity decreases, heavily affecting the network performance. The source of the attack is clearly identified unless a **spoofed address** is used (namely a fake address), which is usually forged via the raw socket interface on the operating system, making attacking systems harder to identify

More generally, **flooding attacks** are classified based on the network protocol used. Each type of flooding attack aims to overload the network capacity on some link to a server. Virtually, any type of network packet can be used to instantiate a flooding attack. The most commonly used are:

- **ICMP flood**: realized by sending excessive ICMP protocol echo requests called *pings*
- **UDP flood**: realized by sending excessive UDP packets directed to some port number on the target system
- **TCP SYN flood**: realized by sending excessive TCP packets directed

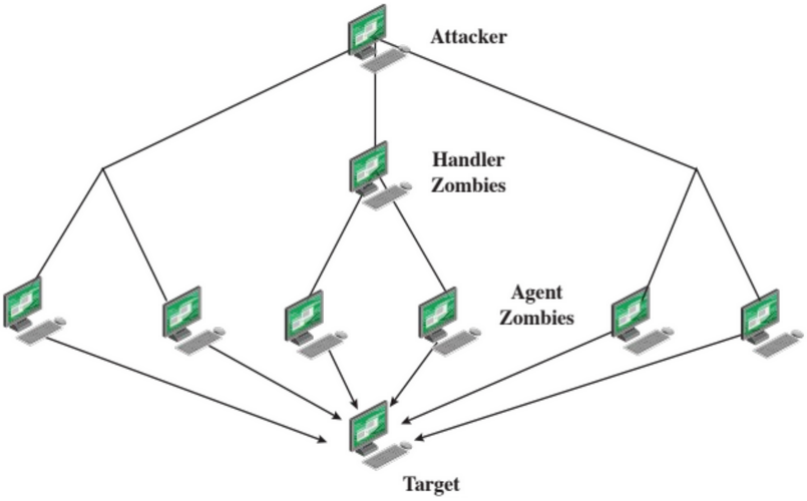
A particular type of DoS attacks uses **SYN Spoofing**, affecting the ability of a server to respond to future connection requests by overflowing the tables used to manage them, making access to the service impossible for legitimate users. This attack is realized through the exploitation of the **TCP three-way connection handshake**:

- The attacker sends a SYN packet to the server by using a spoofed address
- The server tries to send a SYN-ACK packet to the spoofed address
- Since the spoofed address is non-existent or not actively waiting for that packet, the request times out
- The server retries to send the SYN-ACK package a couple of times, eventually assuming the connection failed
- By sending a large number of spoofed SYN packets to server, the latter will eventually become unavailable to manage other requests



Definition 15: Distributed DoS (DDoS)

We define as **Distributed DoS (DDoS)** a particular type of DoS attack realized through a large collection of systems under control of the attacker, such as a **botnet** previously created through spreading a worm malware. The machines used in the attack are called **agent zombies** and their are usually coordinated by **handler zombies**, both part of the botnet.

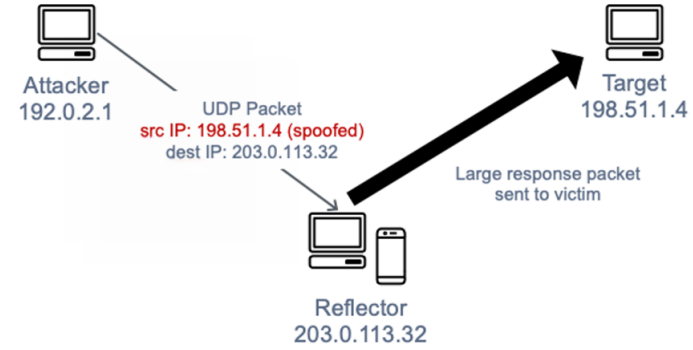


The use of botnets becomes particularly good in case of **HTTP flood**: each bot bombards the targeted webservers with HTTP requests, starting from a given HTTP link and following all links on the provided website in a recursive way (**spidering**). Another type of HTTP DoS attack is **Slowloris**, where the machines attempt to monopolize the servi-

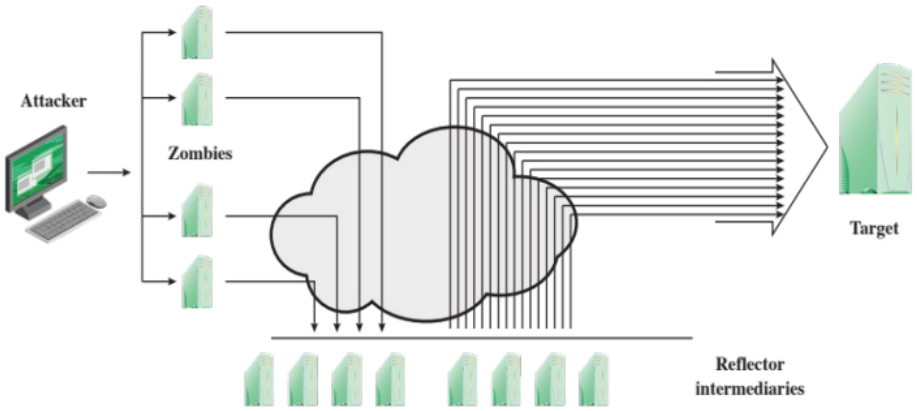
ce by sending HTTP requests in an excessively slow way, making the requests effectively never complete without letting the connection timeout.

More complex DoS attacks are realized through a technique called **reflection**:

- The attacker sends a packet with a spoofed source address to a known service on an intermediary host, where the spoofed address corresponds to the victim's address
- When the intermediary responds, the response is sent to the victim
- The goal is to reflect the attack off the intermediary (the **reflector**) instead of the original attacker



A more advanced form of reflection attack is called **amplification attack**, where the attacker exploit the behavior of the DNS protocol to convert a small request to a much larger response (amplification), flooding the target with responses.



Example:

- The following simple DNS request sends a 64 bytes package, but the DNS response is 3223 bytes long (50x amplification)

```
dig ANY isc.org @x.x.x.x
```

Another type of DDoS attack is realized through the exploit of **memcached**, a high-performance caching mechanism for dynamic websites that allows to speed up the delivery of web contents. The idea is to make a request that stores a large amount of data

and then send a spoofed request to make such data delivered to the victim via UDP. Memcached DDoS attacks can bring an amplification factor of 50000x, making them really dangerous.

Even though they are based on a very simple concept, **DoS attacks can't be entirely prevented**: a high traffic volume may be legitimate or not, especially for famous sites, making it hard to detect if an attack is incoming or not. The most common **prevention techniques** include:

- Blocking spoofed source addresses
- Filters that can ensure the path back to the claimed source address is the one being used by the current packet
- Usage of modified TCP connection handling codes
- Blocking IP directed broadcasts
- Usage of mirrored and replicated servers when high-performance and reliability is required

2.3 Buffer overflows

Definition 16: Buffer overflow

We define as **buffer overflow** any condition under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information

Buffer overflows are a very common vulnerability and potentially one of the most dangerous ones. They exploit programming errors due to a process **attempting to store data beyond the limits** of a fixed-size buffer, overwriting the adjacent memory locations, corrupting data or enabling execution of code chosen by the attacker. A buffer could be located on the stack, in the heap or in the data sections of the process.

Example:

- Consider the following C code:

```
#include <stdio.h>

void main(){
    char str1[8] = "Hello!";
    char str2[8];

    gets(str2);
    printf("String 1: %s - String 2: %s", str1, str2);
}
```

- After compiling and executing the code with input `TEST`, we get the following output

```
[exyss@exyss ~]$ ./test.out
TEST
String 1: Hello! - String 2: TEST
```

- If we execute the code with input `BUFFEROVERFLOW`, we get the following output

```
[exyss@exyss ~]$ ./test.out
BUFFEROVERFLOW
String 1: ERFLOW - String 2: BUFFEROVERFLOW
```

- The space on the stack get allocated downwards, meaning that `str1` was allocated above `str2`, while instead the `gets` function fills the given buffer upwards
- Thus, by making the buffer overflow, the buffer of `str1` was overwritten

To exploit a buffer overflow, an attacker needs to identify a **vulnerable buffer** in some program that can be triggered using externally sourced data under the attacker's control, requiring knowledge on **how that buffer is stored** in memory and ability to determine potential for corruption. Identification of vulnerable programs can be done by **inspecting the source code**, by **tracing the execution** of the program as they process oversized input or by using tools such as **fuzzing** to automatically identify potentially vulnerable programs.

Older programming languages are typically low-level, meaning that problems such as memory management are under the programmer's responsibility. Thus, an unexperienced programmer is more likely to make programming mistakes, exposing the software to buffer overflows. Instead, modern languages are typically high-level, meaning that such problems are managed by the language itself. These languages tend to be more secure, but also more resource expensive.

The most common type of buffer overflows are **stack overflows**, where adjacent buffers declared in the same stack frame get overwritten, such as the previous example. Some of the C language standard library functions are actually common stack overflow vectors, such as `gets`, `sprintf`, `strcat`, `strcpy` and `vsprintf`.

Definition 17: Shellcode

We define as **shellcode** any machine code specific to the processor and operating system used

A very common way to exploit buffer overflows is through the injection of **shellcode**, which then gets executed directly by the program. To be effective, the shellcode must be **position independent**, meaning it must be able to run no matter where it is located in the memory. The attacker generally cannot determine in advance exactly whe-

re the targeted buffer will be located in the stack frame of the function in which it is defined.

Example:

- Consider the following C code:

```
int main(int argc, char* argv[]){
    char* sh;
    char* args[2];
    sh = "/bin/sh";
    args[0] = sh;
    args[1] = NULL;
    execve(sh, args, NULL);
}
```

- The equivalent position independent x86 assembly code is the following:

```
        nop
        nop
        jmp find
cont:   pop %esi
        xor %eax, %eax
        mov %al, 0x7(%esi)
        lea (%esi), %ebx
        mov %ebx, 0x8(%esi)
        mov %eax, 0xc(%esi)
        mov $0xb, %al
        mov %esi, %ebx
        lea 0x8(%esi), %ecx
        lea 0xc(%esi), %edx
        int $0x80
find:   call cont
sh:     .string "/bin/sh"
args:   .long 0
        .long 0
```

- Once compiled, the assembly code gets translated to the following hexadecimal shellcode

```
90 90 eb 1a 53 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 2f 73 68 20 20 20 20 20 20
```

- Through a buffer overflow, an attacker can inject this shellcode ad force the program to execute it, spawning a system shell

2.3.1 Buffer overflow countermeasures and variants

Ideally, any programmer should be able to write safe software, preventing exposition to buffer overflows. However, in order to protect unexperienced developers, modern languages developed **built-in countermeasure** to these type of attack.

The first countermeasure is **compile-time defenses**. As we already discussed, modern high-level languages aren't vulnerable to buffer overflow attacks, requiring however additional built-in code to be executed at runtime to impose safety checks and additional resources, limiting their usage. For example, programs such as device drivers and embedded software cannot be written through these languages due to the necessity of manually managing delicate operations and/or resource usage.

Additionally, modern languages are slowly replacing old libraries with **safer libraries**, allowing programmers to handle complex things such as dynamically allocated memory more easily. Another powerful addition is **stack protection**, which adds entry and exit codes to each function in order to check the stack for signs of corruption, such as the GCC Stackshield and Return Address Defender (RAD).

The second countermeasure is **run-time defenses**, such as:

- **Executable address space protection**: uses virtual memory support to make some regions of memory non-executable, requiring support from the Memory Management Unit (MMU) and for executable stack code
- **Address space randomization**: manipulates the location of key data structures (stack, heap, etc...) using a random shift for each process. Due to the large address range on modern systems, this mechanism requires wasting some space, having however negligible impact
- **Guard pages**: places a guard page between critical regions of memory and, on some extent, between stack frames and heap buffers. Any attempted access to these regions immediately aborts the process execution

Depending on the way it's executed, we can distinguish four **variants** of buffer overflow:

- **Replacement stack frame**: overwrites buffer and saved frame pointer address, which gets replaced with a dummy stack frame, making the current function return to the replacement dummy frame and transferring control to the shellcode in the overwritten buffer.

  Common defenses include: stack protection mechanisms that can detect modifications to the stack frame or return address, usage of non-executable stacks and randomization of the stack in memory and the system libraries

- **Return to system call**: replaces return address with a standard library function that uses pre-constructed suitable parameters

  Common defenses include: stack protection mechanisms that can detect modifications to the stack frame or return address, usage of non-executable and randomized stack

- **Heap overflow**: same as stack overflows, affecting the dynamic memory, including various data structures.

  Common defenses include: usage of non-executable and randomized heap

- **Global data overflow**: same as stack and heap overflows, affecting the global data memory, including function pointers.

  Common defenses include: guard pages,usage of non-executable and randomized global data region

2.4 Database security