ICT Risk Assessment - Appunti

Francesco Lorenzoni

September 2023

Contents

Chapter 1

Introduction

19 - Settembre

1.1 Info and Contact

Info on the exam...

Question time Monday from 16:00

1.2 ICT Security Fundamentals

A system security policy should preserve:

1. **Integrity** - Only users allowed to *update* certain information are actually able to update it
2. **Availability** - Users who want to *use* the system must be able to use it in a finite and reasonable amount of time
3. **Confidentiality** - Only users allowed to *read* certain information are actually able to read it

Depending on the adversary, the terminology changes

$$\text{Natural events} \Rightarrow \textit{Safety}$$

$$\text{Malicious and intelligent} \Rightarrow \textit{Security}$$

There are other properties regarding systems and their security:

1. **Robustness** evaluates how well the system resists and it is not violated by an attack
2. **Resilience** evaluates how well a system recovers and resumes its normal behaviour after it has been violated
   Resilience is preferred to robustness because it is more cost effective
3. **Vulnerabilities** are defects in the system which reduce robustness, hence safety and/or security. NOT every *defect* is a vulnerability, but every vulnerability is a *defect.*

Secondarily there are properties derived from *security*:

1. Traceability - Discover who has invoked a given operation
2. Accountability - Those who use a resource should pay for it
3. Auditability - Whether the security policy is enforced and satisfied
4. Forensics - Proving that some action has occurred and who has executed it
5. Privacy/GDPR - Who can read and update a certain information

Forensics is distinguished by *Traceability* because it is related to what can be proved in a court of law, not only who performed a single operation on the system. Forensics refers to a set of information able to convince a non-expert that something

21 - Settembre

1.3 Security policy - Glossary

First of all an **Asset analysis** is required. It is mandatory for a business to determine which resources are critical for their system to work, allowing to focus security efforts on specific assets.It is also crucial to determine the **impact**:

- A business process is stopeed (integrity or availability)
- A resource has to be rebuilt ex novo (integrity)
- Attacker discovers the information in the resource (confidentiality)

Asset *discovery* is usually done through an application installed in specific assets and discovers all the assets in the company network.

An **externality** is a cost or benefit incurred or received by a third party that has no control over the creation of that cost or benefit. It's important also to consider this because often the security of an ICT system may depend on third parties factors and entities, whose security isn't controlled by the owner of the system.

Security is a job *shared* by many individuals, hence there may be some **free riders**, i.e. individuals who tend to shirk and be negligent, and whose lack of effort affects security.There may be three prototypical case which define on which individuals depends the security of a system:

1. **Weakest-link**: security depends on agents with the *lowest* benefit-cost ratio. (worst-case scenario)
2. **Best shot**: security depends on agents with the *highest* benefit-cost ratio.
3. **Total effort**: security depends on the sum of efforts of **many** agents.

Chapter 2

Security Policy

21 - settembre

2.1 Definining Security Policy

A **Security Policy** is a set of rules that an organization adopts both to minimize cyber risk and to define the goals of security; it must:

- Define goals of security : assets and resources to protect assets
- Define the correct behaviour of all users
- Forbid dangerous behaviours and components
- Imply the definition of:
    - ...
- Avoid violating the legislation that concerns ICT systems

2.2 Terminology

2.2.1 Subject and Object

- **Subject**: entity which can invoke an operation on an object
  Subject or *Principal*
  ; e.g. User, application, program, process, ...
- **Object**: Instance of abstract data type; e.g. function, variable, logical or physical resources

An *object* which invokes operations on other object is both an object and a subject.

2.2.2 Access Rights

If subject S is entitled to invoke operation $\mathcal{A}$ on object Obj, then S owns an access right on $\mathcal{A}$ on Obj.
Access rights can be directly or indirectly deduced from the security policy and from the adopted implementation:

- **Direct**
$$S\, can\ read\ file F \Rightarrow S\, owns\ a\ read\ right\ on\ F$$

- **Indirect** Any program P - executed by S which reads the memory segment MS in which F is stored - owns a read right on MS

2.3 Composite policy

A whole security policy is the result of the composition of 9 more specific policies.

2.3.1 AUP - Acceptable Use Policy

2.3.2 ACP - Access Control Policy

2.3.3 Change Management Policy

Refers to formal process for making changes to the ICT system, including software and hardware updates, third parties dependencies...

2.3.4 Information Security Policy

Critical one, determines which users/applications can invoke object operations that reads and manipulates system information.

2.3.5 Incident Response Policy

2.3.6 Remote Access Policy

Defines acceptable methods for accessing remotely assets in the system internal network

2.3.7 Email/Communication Policy

Defines how employees can use business communication medias, and what contents they can share through them.

2.3.8 Disaster Recovery Policy

Defines how to behave if an event has a significant business impact

2.3.9 BCP - Business Continuity Plan

2.4 ISP - Information Security Policy

Determines which subject can invoke object operations that reads and manipulates system information.
Owner may choose to structure the policy in two ways:

- Default **allow**: policy defines *forbidden* operations
- Default **deny**: policy defines *legal* operations

Secondarily, one must decide the owner's degree of freedom:

- **Discretionary** Access Control: no constraints, the owner is free (commercial world)
- **Mandatory** Access Control: some contraints the owner cannot violate (military/defence world)

2.4.1 Six Dumbest Ideas in Computer Security

1. Default Allow
2. Enumerate Badness
3. Penetrate and PAtch
4. Hacking is Cool
5. Educating Users
6. Action is Better then Inaction

The first two points are strongly related. The reason for 1. is that dangerous behaviours, i.e. things to forbid, are much more than legitimate ones, so the choosing *Default allow* implies to enumerate badness (2.), i.e. things to be forbidden.

2.4.2 Discretionary Access Control

For each object there is an:

- owner of the ICT system
- of the business process that uses the object

The owner is unconstrained and decides the rights of all other users on its own objects.

2.4.3 Mandatory Access Control

All objects and subject must be partitioned into partially ordered[1] classes (possibly the same set of classes for obj and subj, it would simplify). S may be granted the right to invoke $\mathcal{A}$ on Ob only if the classes of S and the one of Obj satisfy a predefined condition that does not depend upon S or Obj.
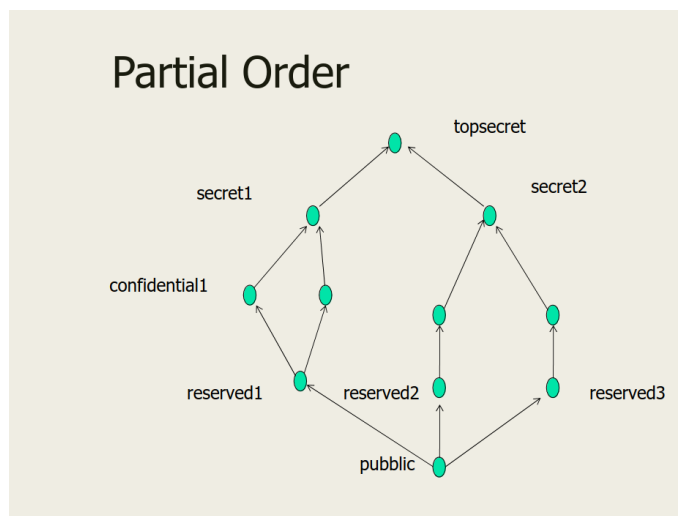
22 - Settembre

2.5 MAC Policies

2.5.1 Bell-LaPadula

Any subject S in class C is **allowed** to:

- Read any file in a class $D \leq C$
- Write any file in C class
- Append to any file in a class $D \geq C$
- Grant rights if S is the owner and the previous contraints are satisfied

This a *no write-down* policy, which preserves confidentiality and prevents an information flow from a higher to a lower level, but may lead to information clustering in higher levels

2.5.2 Biba

- Read any file in $D \geq C$
- Write any file in $D \leq C$

Also called *no write-up* policy, guarantees integrity but sacrificing condidentiality.

2.5.3 Watermark

Time-dependant

2.5.4 No interference property

2.5.5 Clark-Wilson

2.5.6 Chinese Wall

As soon as S invokes and operation on $Obj \in C$, then:

- S cannot invoke operations on Objects $\notin C$
- S can invoke operations on Objects $\in C$

---

[1]Check image below for an example

2.5.7 Overall Policy

In a real world context, organizations merge several of the above-mentioned policies. For a subject, the may be two distinct levels, one for *confidentiality* and one for *integrity*

2.6 Trusted Computing Base

TCB includes any modules involved in the implementation of the security policy. TCB modules are critical, because any bug in one of these represents a vulnerability.

It is important to note that the security level (and the trust in it) is *inversely proportional* to the **size** of the TCB, since more lines of code imply higher chance to hide bugs, hence vulnerabilities.

2.7 Representing Security Policy

It is possible to build an Access Control Matrix, where `ACM[i,j]` indicates which operations subject `I` can invoke on object `j`.

Some kind of representation of ACM is a **necessary** and **not sufficient** condition to consider a policy valid and system actually secure.

	O	
S		
	Rights	

Table 2.1: Access Control Matrix

Chapter 3

Vulnerability, Attack, Intrusion

3.1 Vulnerability

A **vulnerability** is a defect in a person, a component or a set of rules which enables a threat agent to execute an **attack**: action that grants access rights that violate the security policy.

In short,

*A **vulnerability** is a bug that enables an **attack***

Every vulnerability is a bug, but *not* every bug is a vulnerability

3.2 Attack

An **attack** is an action and/or the execution of some code that may grant to the person or the module that executes it some illegal access rights, and it is related to a vulnerability that enables it.
The output of an attack is **stochastic**, it may fail according to a probability distribution.

3.3 Threat Agent

A **threat agent** is a source of **attacks**, it may be *natural* (floodings, earthquakes...) or *man-made* (adversary with a goal). *Man-made* may be malicious or random (employee which clicks on something dangerous accidentally).

It is possible to **assess risk** only if assets, vulnerabilities and threat agents are known for a given system.

3.4 Intrusion

An **intrusion** is a sequence of *actions* and *attacks* of a threat agent to reach its goal, which initially owns its legal access rights and aims to gain illegal ones, hoping to control — a subset of — an ICT/OT system. Some actions may be actuals attacks, while others may collect information to discover possible attacks. Such actions (and attacks) can be implemented by a program called *exploit*.

Once a threat agent gained control over an ICT/OT system:

- Collect and exfiltrate infomration from the system
- Update any information in the system
- Prevent access to any resource/information in the system

Steps of an intrusion= how a hacker behaves

1. The threat agent collects information about the target system
2. Discover vulnerabilities in the system that enable an initial access
3. Intrusion = sequence of actions/attacks
    initial access;
    repeat
        1. information discovery and collection about system modules
        2. Vulnerability discovery in system modules
        3. Build/Buy Exploit
        4. Attack ⇔ Exploit execution+ Human Actions if required
                    (manage the output of the attack)
    until agent goal is reached
4. Install tool to remain in the system = persistence
5. Remove any trace of the intrusion in the target system
6. Lock, encrypt, delete, steal a subset of the information in the system
    1. Exfiltrate some information
    2. Manipulate some information

The steps of an intrusion include a recursive phase highlighted in red in the picture; it appears clear that an attacker cannot plan an entire intrusion in advance before starting it, since an attack reveals information and (possibly) vulenon the system which the next attack will be based on.

3.5 Initial Access

A set of techniques that adversaries may use in an intrusion as entry vectors to gain an initial foothold within an ICT/OT environment.

Informations gathered through initial access are sold on the deep web to hackers team who aim to penetrate a system.

3.6 Countermeasure

The **attack chain** is the sequence of *useful* attacks in an intrusion. A defendant wants to increase the number of *useless* attacks to slow down an intrusion. Besides, it is not mandatory to remove *all* vulnerabilities to prevent an in intrusion, but even only one may be sufficient to interrupt te *attack chain*, thus preventing the attacker from collecting information that would lead to further attacks.

There are two main approaches when considering security:

- **Unconditional security**: Assume that any vulnerability will be exploited regardless of costs and complexity
- **Conditional security**: Consider who is interested in attacking the system and which vulnerabilities their intrusion can exploit.

3.7 Risk assessment

To wrap up, let's define what **Risk assessment and management** involves, keeping in mind that *cyber risk* resembles the average loss for instrusions.

1. Asset analysis
2. Threat agent analysis
3. Vulnerability analysis
4. Adversary emulation
5. Impact analysis
6. Risk evaluation and management: *compute and minimize loss*
    - Compute the risk
    - Accept some risk
    - Reduce some risk (countermeasures + scheduling)
    - Transfer residual risk (insurance)

Chapter 4

Vulnerabilities

In this chapter we'll take a deeper look into vulnerabilities and the related attacks, providing some examples and details.

A dummy but instructive vulnerabilties classification distinguishes two major classes:

1. **Local vulnerability** → vulnerability in a single *module*. Even if the vulnerability needs other modules to be exploited, the defect it mostly depends on is in a single module, and such kind of vulnerabilities can be removed by updating the modules they're dependant on.
2. **Structural vulnerability** → a vulnerability which arises as a result of the composition of multiple modules. Also called *emerging* vulnerability.

4.1 Local vulnerabilities

A basic example of a local vulnerability, is **memory overflow**, particularly easy to exploit in software written in C, due to its memory management. Basically, it is possible to *inject code* in a small memory area by inserting more data than the actual available space. It is common for many attacks to inject *code* where the software expects *data*. Another known example is SQL injection.

It is possible to inject code exploiting **stack overflow**. stack memory areas start at the highest memory addresses and grow backwards, towards lower addresses.
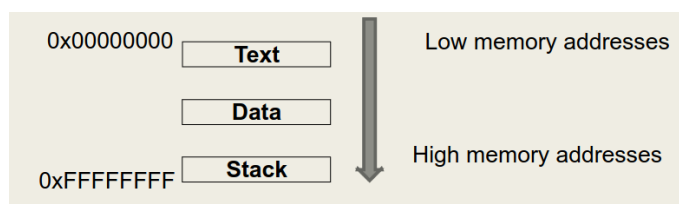
Thus it is possible to inject fake stack frames and pointers to them in the *data* area, using overflow and preventing segmentation faults.

4.1.1 Address Space Layout Randomization ASLR

(...)

4.2 Structural Vulnerabilties

Many attacks exploiting structural vulnerabilities were aimed to discover alive nodes in a network. There is *no control* on the fields of IP packets, thus senders are not authenticated. A threat agent may send *ECHO* messages using a broadcast address or to specific nodes to find out which are alive.

The *Distributed **Denial of Service*** is performed by flooding a network with *echo* messages, making the bandwidth occupied by such messages and relative replies, stopping the services running in such network.

A *Slow Denial of Service*, instead, exploits collisions in a hash table to fill up memory and slow down performance of a Module.

Is this a local vulnerability? Maybe not because it also depends by the possibility for the attacker to push input into the hash table.

4.3   Security Partial Views

4.3.1   Encryption

According to Baiardi, **Encryption** *simplifies* some **security** problems, but does **not** *solve* them. Encryption guarantees *Confidentiality*, and sometimes *Integrity*, but **not** *Availability*. Some claim that encryption solves security, but it must be taken into account that the operating system of a module may not provide a way to protect the encryption keys properly.

4.3.2   Authentication

Most security problem require three problems to be solved:

1. User identification
2. Resource identification
3. Analysis of access rights

User identification is not sufficient, *as some may say*, because it indicates which row of the **Access Control Matrix** to consider, but not how to actually use the matrix.

To provide authentication three classes are considered, and the well known two-factor authentication should pick factor from two of these:

1. *Something you **know***: password, PIN, pet name
2. *Something you **own***: smartphone, credit card, token
3. *Something you **have***: biometric features, i.e. fingerprint, voice, retina

Chapter 5

Discovering Vulnerabilities

5.1 Classification

To address the problem of finding out vulnerabilities many classifications have been proposed, and each one has its own purpose: module affected, how to discover it, enabled attacks, ...
It is mandatory to understand a classification goal before using it.

It is possible to consider where the vulnerability resides to provide some sort of classification:

1. **Procedural**: the actions executed are not correct
2. **Organization**: actions well defined but wrongly executed
3. **Tool**: actions well defined and correctly executed but by bad tools, e.g. OS, compiler, run time support...
   Password transmitted in clear, missing checks on boundaries...

About **tool vulnerabilities**, it is important to pay attention to **code reuse**. It must be considerend that reusing code may mean re-enable a vulnerability in such code. Code reuse is ok, but only along with **code Hardening**, which means, removing instructions and libraries which are not needed.

About the implementation, it is relevant to avoid missing controls on stuff like user input, function parameters, confused program-flow... Generally a strong type system may aid to address these kind of problems.

Besides, to avoid structural vulnerabilities, it is crucial to check whether certain modules depends on the security checks performed by others.

Searching for Vulnerabilities

Aside from the distinction between *Local* and *Emerging* vulnerabilities, it is also important to distinguish between **standard** modules (OS, web servers...) and **specialized** ones (dynamic pages produced by the server).

5.2 Vulnerability Life-Cycle

1. Born when someone does something wrong
2. Known when someone discovers the error
3. Public when its presence is revealed and it is inserted in some public database
4. Some look for a remedy/fix while others search for an exploit[1]
5. Vulnerability might become exploited
6. If existing, the fix should be applied ASAP

Note that this life-cycle doesn't take into account a **zero-day** vulnerability, which is a vulnerability not made public, whose discovery is shared only among few teams or people

Historically the most dangerous vulnerabilities are public and exploited ones; even the oldest ones are exploited because attackers are lazy and defenders even more.

---

[1]A program to implement an attack that exploits it

5.3 Attacker vs Owner POV

Considering the point of view of an **owner** who wants to search for vulnerabilities to improve their system's security.

In order to search for vulnerabilities, an inventory of **all** the system modules is required. This is not a trivial task, but it is necessary.

You cannot protect what you don't know.

The opposite view is the **attacker**'s one. Usually vulnerabilities of standard modules are **known**, thus an attacker may only need to know which *modules* compose the system. An attacker may acquire knowledge on the vulnerabilities from public or private (by paying) databeses, or by buying such information in the deep web. It's rare for an attacker to look by himself for vulnerabilities in a module.

5.4 Scanning

5.4.1 Fingerprinting

Active fingerprinting is a (set of) tool which exploits the fact that modules communicate through ports, and appears quite appealing from both the mentioned views: given a range of IP addresses, it sends packets on each port and analyzes the replies to *fingerprint*[2] the module listening on the port.
The *owner* might want to run the tool on the entire network, while an attacker may target single (or a few) hosts at a time.

Active fingerprinting is noisy and might considerably slow down the network performance, which in some systems, e.g. *ICS* (*Industrial Control System*), must be avoided.
(TODO - CAPIRE MEGLIO)

A **Passive fingerprinting** does not imply direct interaction with modules, but acts as a **sniffer**, analyzing packets the modules exchange in a transparent way. It exploits info in TCP and IP headers to fingerprint modules. The counterpart is that in networks with low noise/packets exchange, passive fingerprinting may take long to discover all the features of interest. Usually cannot be used by attackers.

It is important to note that a scanner may not know whether a **patch** has been applied or not to a module, hence it may report vulnerabilties which in fact have been patched, generating a **false positive**. A scanner may also generate **false negatives**, since some vulnerabilities for a given module may not appear in the DB the scanner uses to map modules to related vulnerabilities. There are also some **breach and simulation** tools which besides scanning also execute an exploit to check whether given vulnerabilities have been patched or not. Even though interesting, it may be dangerous to run such software in low-tolerance systems.

To evaluate vulnerabilities discovery methods it is common practice to use a **confusion matrix**[3], which provides, amongst others measures, *accuracy*, *precision*, *recall*[4], *specificity*.

5.4.2 Stealth scanning

Clearly, the owner is interested in discovering if anyone aside from him is currently scanning the system. An attacker may configure message frequency and the number of nodes to scan, to reduce the chance to be detected.

5.4.3 More on scanning

An owner may combine:

- External vulnerabilities scan: Try to access the sytem from outside, to understand what can an attacker discover before starting an intrusion.
- Internal vulnerabilities scan: Aims to test and fingerprint devices and modules inside a network. It might be ran by either owner or attacker after the initial access.
- Intrusive scans: i.e. breach and simulation. These are the most stringent scans, but may be disruptive.

Anyway note that it is crucial to **periodically scan** a system, due to its eventually mutable nature but way more importantly, because about 20 new vulnerabilities get published every day, along with new potential attackers and

---

[2]i.e. "discover the identity of'
[3]wikipedia.org/wiki/Confusion_matrix
[4]sensitivity

attack techniques.

5.5   Searching in a Module

Vulnerabilities can be searched and assessed when designing the system. Modules can be standard (e.g. OS), and thus be affected by public vulnerabilities, or specialized modules, whose vulnerabilities are unknown.

2 - Ottobre

Static Application Security Testing (SAST) indicates tools of static analysis, which has advantages like scalability and easy patching, but is limited on other aspects, such as:

- Authentication Issues
- Dangerous Management of access rights
- Unsafe use of cryptography
- Many False negatives
- Cannot evaluate runtime values

Opposed to static analysis, there are *Dynamic Analysis* techniques on which big companies generally rely on. The most common technique is **Fuzzing**.

5.5.1   Fuzzing

Fuzzing is associable to *"Chaos Monkey"* test design paradigm: the idea is to send *malformed inputs* to a module. If the system responds with a crash to malformed inputs, such crash may indicate a bug, i.e. a *vulnerability*.
A fuzzing tool results from the composition of three modules:

| Malformed input generator | Input scheduler | Monitoring crashes tool |
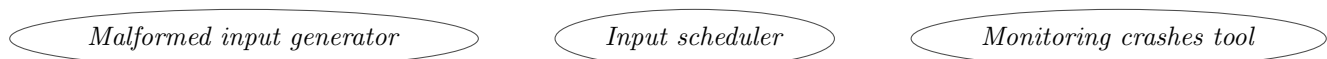
Before executing a fuzz test, a **tainting analysis** is perfomed. Its aim is to computer for each input the set of variables that could be affected by the input. Usually this is done along with a coloring analysis, i.e. *TODO*.

(...)

Black-box fuzzing makes no assumptions on the implementation details of modules. It has grown a lot in popularity since it is faster and, more importantly, nowdays many modules, especially in IoT sector, do not provide open source code.

Some rules to efficiently apply fuzzing should be considered: First of all, input-format knowledge should be kept in mind especially for black-box debugging; in general the longer you run a test, the more bugs you may find, until a saturation point is reached; It is advisable to use different fuzzers, since they may find different bugs.

5.6   Web Vulnerability Scanner

Despite the name, it doesn't work as other scanners: its focus is onto discovering vulnerabilities in a website whose behaviour is determined by the dynamic generation of pages. Passwords and credentials may be given as input to look deeper into the website. Such scanners work much more like to *breach and simulation* then to regular scanners.

- SQL Injection: inserting or deleting information from a database
- XSS: inserting a malware on a website to be later downloaded and executed by the end user (*cross site scripting*)
- CSRF: forces an end user to perform unwanted actions on a website he is authenticated. (*cross site request forgering*)

3 - Ottobre

5 - Ottobre

Chapter 6

Attacks

3 - Ottobre

6.1   Attacks and Vulnerabilities

Following the discovery of a vulnerability v there's an analysis to evaluate which attacks are enabled by v. **Attacks** can be described as a set of attributes:

1. Precondition
2. Postcondition
3. Success Probability
4. Know how, abilities, tools required
5. Noise = Probability of being discovered
6. Automated/Potentially automatable/manual
7. Local/Remote
8. Actions to implement attack[1]

Even though some attack evaluation proposals map to each attribute a number and combine them into a value, such evaluations do not consider that **risk** resides in *intrusions*, not individual attacks, because they have a considerable impact on the system, and keep in mind that are composed by:

- Exploration and information collection
- Persistence
- Attack *chain* for privilege escalation

6.2   Attack Classification

The actions need to implement an attack may be used to define a **taxonomy** of attacks:

1. buffer/stack/heap overflow
2. *sniffing* → Illegal access to info in travel
3. *replay attack* → Repeated exchange of legal messages
4. *Interface attack* → Illegal order in the invocation of API functions
5. *Man-in-the-middle* → Interception and manipulation of info in travel
6. Diversion of an information flow
7. *Race-condition* → Time-to-use time-to-check
8. *Cross site scripting* → XSS
9. SQL injection
10. *Bell-Lapadula policy* → Covert channel
11. Masquerading as
    - user
    - machine (*IP/DNS spoofing, Cache poisoning*)
    - connection (*connection stealing/insertion*)

---

[1]See following Section on attack taxonomy

6.3   Examining attacks

Replay attack

Suppose a user asks the bank to transfer some money to Y account with an M message. Y may sniff and record M, and before the secure channel S gets deleted, Y sends M several time.
Note that the attack may work even if encryption is used.

Man-in-the-middle

If A and B communicate, E may pose itself in the middle, acting as if it were B to A and A to B. Such attack is possible when no authentication is required.

XSS

A website allows users to upload contents to be later (possibly) downloaded by users. Thus a malicious user may upload hidden scripts to damage or steal information from the user who download their content. To avoid this the website must check the content uploaded by users.
A well known attack of this type targeted BBC.

SQL Injection

An input may insert a malicious query (i.e. `DROP TABLE` USERS) in a credentials field. The best way to avoid this is to whitelist using RegEx.

Cryptography attacks

These are a category on their own, there are many types, with different variations and features.

Side-channel attacks

Any attack that measures some physical value to discover an encryption key. Currently it is popular due to the capabilities of machine learning in exploiting large number of pairs to deduce a function.
Such measures may be:

- Electromagnetic emissions
- Energy consumption
- Execution time to discover inner status
- Execetion time to discover cache usage and prediction mechanisms.

Virtual Machines & Blue Pill

Cyber system may be composed of many virtual machines onion-like organized. Thus, attacking a low-level VM may grant access rights to higher ones.
Besides, an attacker may insert a new VM in the hierarchy: this is called *Blue Pill* attack, it's hard to discover and has a high impact. A new VM may return to higher VMs fake information on the status of the underlying machines and/or send malicious commands to the underlying machines.
Stuxnet was a malware which used to send commands to uranium enrichment centrifuges to destroy them, and meanwhile told the operator that everything was going well.

Chapter 7

Patching

3 - Ottobre

5 - Ottobre

7.1 Patching

Patching is **slow** and **expensive**. This is due to many factors: first of all there's the need to run **regression** tests, to check correctness of standard behaviour and of he bug to be corrected; besides, new behaviours and problems may arise because of the new code; in case of a complex problem requiring N patches, the **scheduling** of such patches must be taken into account; in general is advised to patch using an automated process exploiting patching agents and environment.

7.1.1 Common Vulnerability Scoring System

Note that, for instance, *Industrial Control Systems* **cannot be patched**, because it would imply for the production to be *suspended* and for the whole system to be *certified* again.
This forces an admin to decide whether "*to-patch or not to-patch*.
About this matter a **Common Vulnerability Scoring System** (*CVSS*) has been developed. Its aim is to consider the main features of a vulnerability and compute a score based upon them; in the initial idea such score should have allowed to define a score threshold to decide whether to patch or not. However, such idea doesn't work for two main reasons:

1. Single vulnerabilities are not of interest, while *intrusions* are i.e. chaining and exploiting multiple vulnerabilities
2. *CVSS* totally *ignores* the **system**, but the context in this topic is fundamental

Even if it cannot be helpful as a guide for an admin to perform the above mentioned decision, it can be truly instructive to understand how impactful a vulnerability can be, and it can provide an approximation of how *difficult* may be for an attacker to exploit such vulnerability. The CVSS provides three *metrics* to evaluate vulnerabilities risks:

- **Base** fundamental characteristics constant over time and user environments. Such metric aims to provide an intuitive and clear vulnerability representation.
- **Temporal** the characteristics that change over time but not among user environments
- **Metric** the characteristics relevant and unique to a particular user environment

...Specs on metrics...

7.1.2 CVSS revisions

Dragos in 2022 proposed a revision of scores in the CVSS considering the attacker point of view, claiming to have better ones, but still raising up many doubts.

Later on the same experts team created the **EPSS** as a measure of exploitability: it is a *Neural-Network* based system which estimates the probability that a vulnerability will be exploited. It is unclear on which data the AI has been trained, accuracy, tuning, etc... EPSS does not consider risk, context or whatsoever, it's just a probability estimator.

SSVC *Stakeholder-Specific Vulnerability Categorization* aims specifically to produce an **action**. Imagine a decision tree 5 levels deep. An admin must make 5 "decisions"[1] about a vulnerability, and then a leaf of the tree can be one among 4:

1. Track
2. Track
3. Attend
4. Act

Considering the decisions to be made by the admin:

1. **State of Exploitation**
   - *None*: No evidence of atcive exploitation and no public *Proof of Concept* (PoC) on how to exploit the vulnerability
   - **Public PoC** Sites like ExploitDB or Metasploit contain PoC on such vulnerability v or v has a well-known method of exploitation
   - **Active** Credible sources claim that v is shared, observable and has been exploited in the past.
2. **Technical Impact**
   - **Partial** control of the software given to the attacker e.g. DoS attack
   - **Total** control of the software or total information disclosure given to the attacker.
3. **Automatable**
4. **Mission Prevalence & Public Well Being**
   - Does the vulnerable component provide support for the (attacker?) mission? Is it essential? Is it not so useful?
   - Which kind and how much harm the attack may cause and if it is irreversibile or not. Physical, psychological, financial, environmental
5. **Mitigation**
   This is not included in the decision tree!
   - Fix available/unavailable
   - System change difficulty
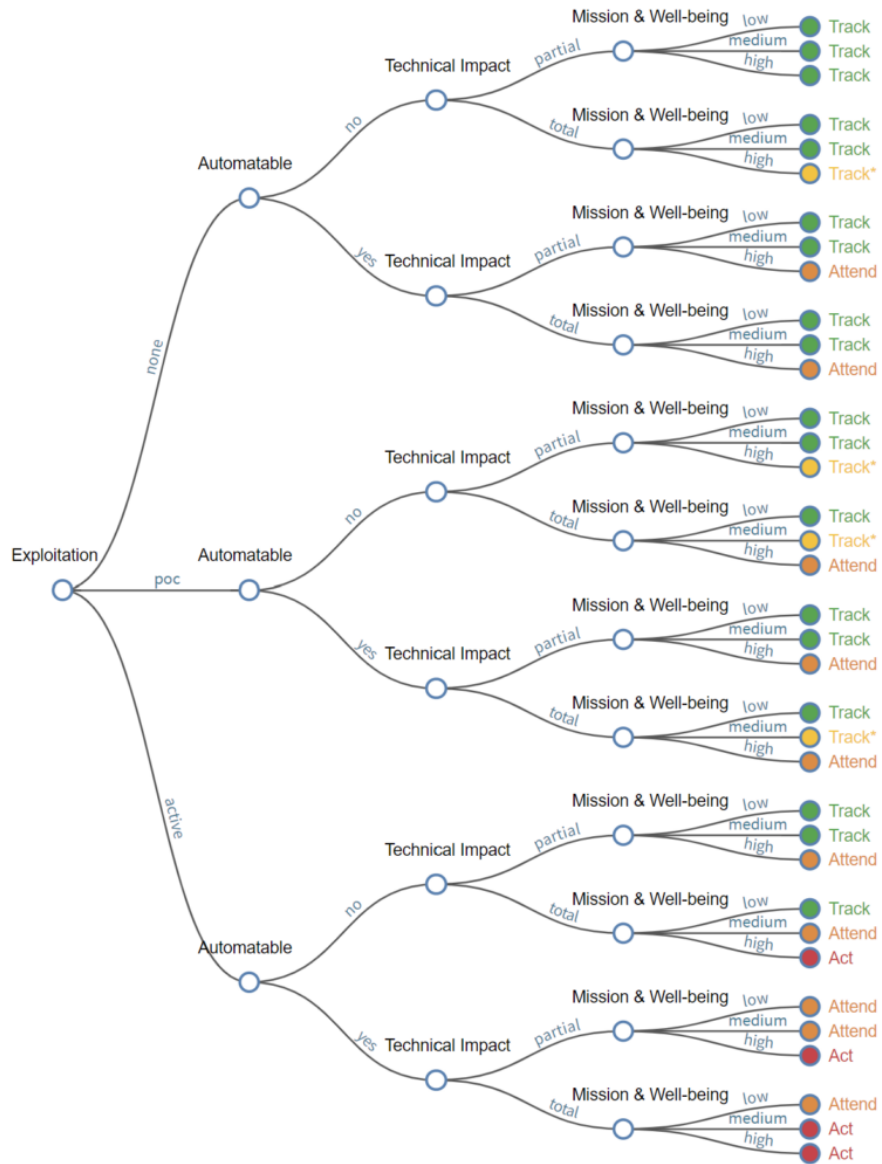   - Actual Fix or Workaround

---

[1]In the sense of answering 5 questions

Figure 7.1: SSVC Decision tree

Chapter 8

Countermeasures

10 - Ottobre

8.1 Introduction

- **Proactive** Patching a vulnerability before being vittim of an attack
- **Dynamic** Countermeasure applied during an intrusion to prevent the attacker from reaching its goal e.g. dropping connection
- **Reactive** Patching applied after an intrusion to prevent the success of the next one.
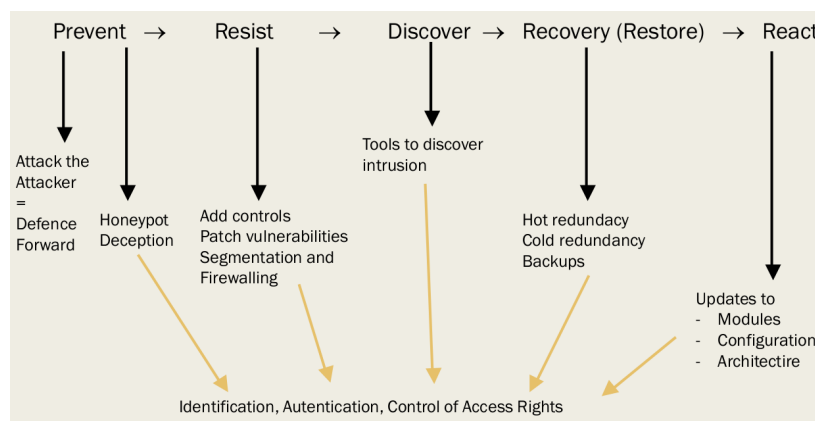

Figure 8.1: Countermeasure more detailed classification

8.2 Robustness and Resilience

Robustness refers to strength and effectiveness, even in adverse conditions. The more robust a product is, the less its performance is affected by disruptions or input changes, because such changes have been predicted and contingency plans have been developed and built into the product.

Resilience instead is the ability to bounce back after disruption. Unlike robustness, which is proactive, resilience is reactive, following incidents in which system performance has already been affected. Resilience is measured in terms of the time a system takes to recover to its original state of performance or better. Both can exploit **redundancy** and **heterogeneity**; **heterogeneity** means using modules from distinct suppliers to avoid a catastrophic failure due to a single vulnerability, increasing robustness.

Redundancy can clearly highly increase *resilience* to faults, there are many kinds of redundancy:

- *Cold* redundancy: spare and idle instances of modules are started only when working instances are unavailable due to faults or attacks.
- *Hot* redundancy: multiple active instances that work simultaneously to tolerate loss due to attack without a recovery e.g. multiple DB copies, or nodes in a network/cloud
- *Triple Modular* redundancy: hot redundancy where three instances of a module receive the same input, execute the same computation and vote the result. Vote can be decentralized or centralized. Space systems use five

copies.

Increases safety but maybe not security

- Overall oversized system in general may allow resource loss

Anytime resilience is based upon reconfiguration, system monitoring is fundamental. System monitoring should discover how close the current system behaviour is close to a boundary and fire the reconfiguration action to remain or return to normal behavior. The largest the amount of information on the current behavior, the higher the performance of monitoring.

8.2.1   Minimal system

One way of dealing with intrusion is to has a **minimal system**, i.e. a subset of the system of robut and heterogenous modules, which can act as a starting point to restore a consistent status. It is crucial not to lose control on the minimal system, since doing so might mean not being to restore the normal behaviour.

From this point of view we can also consider a **minimal behaviour**, i.e. the smallest set of behaviours that is acceptable for the final user. The minimal behaviour requires some features to ensure robustness which can also be used to restore the normal behaviour after an attack.

Consider an ATM and its behaviours as an example, and notice which behaviours compose the minimal one:

1. Protect money
2. Interact with a central system
3. Distribute money
4. Distribute information about accounts

8.3   Authentication

$$\langle subject, object, operation \rangle$$

Keeping in mind this triple, there are two kinds of controls which can be done on it; subject *identity* controls and *access rights* ownership controls (i.e. the subject owns the access right).

Most of the mapping of identity into a set of access rights is a task that usually is delegated to the OS, which can also handle authentication, but typically specialized components are preferred.

Authentication can be classified as follows:

- Weak Static: passwords and similar strategies which can be easily defeated by a sniffing atacker
- Weak not Static: cryptographic techniques to produce information that is not repeated
- Strong: mathematics and encryption to produce information that is not repeated and that may be validated by a distinct channel

8.3.1   Authentication Mechanisms

1. Something the user knows e.g. a password hashed on the server
2. Something the user owns
3. Something the user "is" e.g. Biometric Authentication through fingerprint/retina/face

8.4   Kerberos

Strong authentication network protocol

- A user password must never travel over the network
- must never be stored in any form on the client machine
- must be immediately discarded after being used
- should never be stored in an unencrypted form even in the authentication database

A user enters a password only once per session so that it can transparently access all the services it is authorized for without having to re-enter the password during this session. Authentication information management is centralized on the authentication server. Application servers must not contain authentication information. Such centralization guarantees no redundancy and possible consistency problems, allows an admin to perform edits on the auth DB in a one-time action.
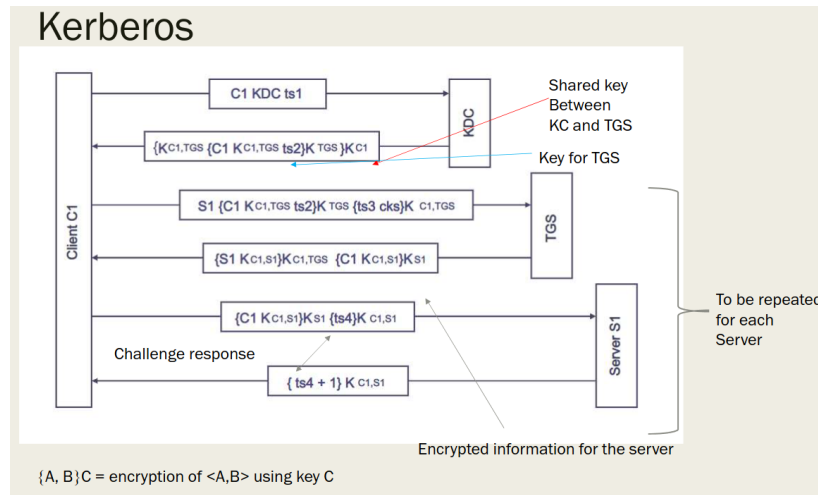
Figure 8.2: Kerberos messages exchange

Kerberos provides a three-sided authentication with a shared key for **symmetric encryption**. The agents are the following:

1. Client
2. Server
   offers a service but want the users to be authenticated
3. KDC Key Distribution Center
4. TGS TIcket Granting Service

Base Principle: *If you know the shared key then you have been authenticated by the KDS*
A **ticket** allows a client to prove its identity to a server to access the service one offers. A ticket is valid in a time window only. However, the **counterpart** is that centralization introduces both *single-points-of-failure* and potential *performance bottleneck*.
Master Keys are encrypted with the *master key* of the **KDC** and with the one of **TGS**. The passwords of these two modules are the last line of defence.

8.5 Zero Trust

A key point of zero trust is that authentication and authorization involve both a subject and a used *device*. The focus of zero trust is to protect resources instead of network segments since the network location is no longer seen as the prime component of the security posture of the resource.

8.6 Control and Management of Access Rights

As said before access rights are represented as a matrix which is highly dynamic.
Note that any OS has an implementation of this matrix to protect physical resources, logical resources and memory areas. Besides each application may have its own matrix to protect the resources it manages.

Basically, an OS matrix determines which users can interact with a given application, and also which operations such users can invoke.

A whole matrix is inefficient due to centralization, thus it is advised to decompose on *rows* or *columns*.

8.7 Rows - Capabilities

A possible implementation is the one with capabilities: this solution stores the access rights in the subject that then extracts and presents the one that enables the operation of interest. An example of usage is the map that translates virtual addresses on physical addresses, each entry in this map contains some bits that represent the permission for that memory region.

On **distributed** systems, we need some way to protect capabilities information from tampering because hardware features can't certify that. To solve the problem we use a shared secret among the systems and we use that secret as

a key to build check digits that are a hash of the capability produced using the key. The receiver of the capabilities and of the check digits can use the digits to verify the capability is authentic.

In **centralized** systems, instead, each pointer corresponds to a capability.

8.8 Cols - Access Control List

This solution provides storing the matrix in colums, one column for each object. An object implementation also stores the information to control the accesses to the object.

An instance of ACL is the packet routing in Linux made with `iptables`, which allows to define rules for each packet "chain":

- Input chain
- Output chain
- Forward chain

Besides it is possible to define default policy PASS/DROP. For each packet it is possible to perform various actions:

- DROP/PASS (route)
- goto/return i.e. call/return packet to a chain
- Queue i.e. handle packets queue using user's code
- Log
- Reject
- dnat/snat/masquerade

In general, filtering packets going *outside* a node, is called **egress filtering**; in other words, before allowing an *outbound connection* a user-defined *rule* must be checked. Such control allows to *discover malware*, stop contributing to *attacks*, or to *block local users* from using illegal services.

8.9 Role Based Access Control

Basically pairing *access rights* with a *professional* **role**. When representing such access rights, there is a simple matrix for each role, making room for scalability and easy management. Clearly, also a mapping between users and roles is needed.

Roles may be partially ordered, leading to $R_1, R_2 \Leftrightarrow (R_2.access\_rights \in R_1.access\_rights)$

8.10 Attribute Based Access Control

Access rights are granted or denied according to values of 4 key attributes:

1. **Subject**
2. **Action**
3. **Object**
4. **Contextual**

This is a flexible approach, but strongly relies on how attributes resist manipulation. Besides it must be noted that *ACL* and its derivations can grant access even to unknown subjects, which instead is not possible with capabilities since they are distributed only to entities we already know.

Chapter 9

Windows authentication

The key concept is the relationship between **logon sessions** and **access tokens**. A logon session represents the *presence* of a user on a machine and begins with a successful authentication and ends when the user logs off.

When a user logs in they provide a pair of $\langle username, password \rangle$ which is checked by *Local Security Authority* (LSA). If the credentials are valid, LSA will create a new logon session and produce an **access token**; multiple access tokens may be associated with a session, but one token can only be linked to one session, typically the logon that generated it. However Windows can change the logon session (and cached credentials) a current token is associated with.

9.1 Access token

Access tokens cache some attributes regarding the user security context, i.e. the privileges and permissions of a user on a specific workstation (and across the network).

- The security identifier (SID) for the user
- Group memberships
- Privileges held
- A logon ID which references the origin logon session

An *access token* act as proxy or stand-in for the logon session. When making security decisions, Windows never interact with the logon session itself ("hidden" in `lsass`, the process implementing LSA), but with an access token which represents it.

9.1.1 Mandatory Integrity Control

Aside from access tokens, there is another secury level in Windows: security *principals* and securable *objects* are assigned **integrity** levels that determine their levels of protection or access. MIC is a mechanism for controlling access to securable objects in addition to discretionary access control and evaluates "integrity" access before evaluating access checks via an object's DACL[1]. For instance, a principal with a low integrity level cannot write to an object with a medium integrity level, even if the DACL of the object allows write access to the principal.

9.1.2 Access Control List(s)

ACLs are the lists in a **security descriptor** with information on actions users, groups, or objects can perform on the file or folder to which the descriptor is applied.
A security descriptor may contain different two types of lists:

1. **DACLs** Discretionary ACL - the list of SIDS for the users, groups, and computer objects allowed or denied access to perform actions on files or folders
2. **SACLs** System ACL - the list of SIDS for the users, groups, and computer objects for which successful or failed auditing events are logged

ACEs are individual entries in either DACLs or SACLs for particular users, groups, or computer objects

---

[1]D? Access Control List

9.1.3 Sandboxing Tokens

Applications e.g. browsers, have historically been victims of attacks. An attacker who successfully exploits a browser, then the attacker's *payload* shares the same *access token* of the browser, allowing it to perform any action the browser is allowed to do.

To mitigate such kinds of attacks, browsers' code has been moved into lower-privilege processes by creating a smaller and restricted security context; in the Unix Documentation, such context is called a **sandbox**.

The key idea is to limit the extent of an attack to only the resources accessible to the sandbox maliciously exploited.

9.2 Remote Hosts AC

A logon session is unique to a workstation and users cannot send an access token over the wire because it would be meaningless as it does not correspond to a valid logon session on the remote host. Furthermore, this is a target for replay attacks. Thus, the user needs to **re-authenticate** and establish a new session on the remote host.

In order to establish a new logon session, the **SMB** server has to authenticate the client over the network. In Windows domains, network authentication is performed via **Kerberos** or the **NTLM** challenge-response protocol. Regardless of the auth method, network logins do not cache credentials and this token cannot be used to authenticate to another remote host. This is the *"double hop"* problem.

Kerberos is **default** authentication method today, NTLM acts a backup in case Kerberos authentication fails.
In NTLM, passwords stored on the server and domain controller are not ***"salted"***, which means adding a random string of characters is not added to the hashed password to further protect it from cracking techniques. Besides NTLM doesn't support many modern encryption algorithms and techniques.

9.2.1 MS Kerberos and MIT Kerberos

In a standard authentication, a user asks its Kerberos key distribution center (KDC) for a session ticket for a specific host. In Windows instead, once authorized to enter, the user must still show his rights for the resource requested, such as a shared file or network printer. In this way the user's security access token in the application-specific data field in a message protocol

9.3 Impersonation/Delegation

In multi-threaded applications, complex race conditions may arise if different threads start enabling/disabling different privileges or modifying default token DACLs. By default all threads will inherit the same security context as their process's primary token. However, impersonation allows a thread to switch to a different security context. Impersonation enables threads to have their own local copy of a token: an **impersonational token**. Such process allows, for instance, an SMB server to handle each incoming request in a separate thread and *impersonate* the access token representing the *remote client*. Thus, **locally**, since the thread is associated with an impersonational access token, any *access checks* will be performed with such token.

What does this mean?
"as this impersonated token may be linked to a different logon session with different cached credentials the thread's security context remotely is also different"

Unless some mechanism protects the token, a thread running as SYSTEM can modify it. To avoid too impactful exploitation of such feature, there is an undocumented feature called *"trust labels"*, which is an optional component of every security descriptor, restricting specific access rights to some types of protected processes.

Chapter 10

Deception

10.1 Honeypot

A **honeypot** is a system designed exclusively to be attacked an to **collect information** about the attacker and its tactics, techniques and procedures; the other focus of an honeypot is also to possibly slow down an attack to a system by **diverting** it on itself.

The scaled version of a honeypot, is a **honeynet**, which is an entire network attached to a real system designed to be targeted instead of the main system.

10.1.1 Classification

1. **Interaction**-based
   (a) *Low* - e.g. simple port listener
   (b) *Medium* - emulation of a network service that analyzes the inputs and returns some replies similar to those the real service would return.
      - Simulates just some **features** of the service
      - Easy to implement, **low risk**
      - Can collect a low amount of information
      - Tools $\longrightarrow$ OS + Honeyd
   (c) *High* - built around real services that run on real machines to fool the attacker
      realistic but dangerous due to the large amount of vulnerable software
      - Simulates **all features** of the service and of the underlyng OS
      - The attacker may fully compromise and control it
      - **High risk**
      - A larger amount of information
      - Tools $\longrightarrow$ Honeynet.
2. **Implementation**-based
   (a) *Virtual*
   (b) *Physical*
3. **Goal**-based
   (a) Production
   (b) Research on attacker behaviour

10.2 Honeyd

Honeyd is a daemon which creates virtual nodes in a network. It is highly configurable and is able to reproduce even large and complex networks; besides it can integrate with virtual and physical real-existing networks.
Honeyd provides many features, we can list some of the main ones:

- It detects illegal activities in a network by monitoring the IP addresses that are not withing a **range** named *"dark space"*. Any attempt of connection to or from the *dark space* is assumed to be an attack or a vulnerability scan.
- It monitors activities related to TCP and UDP ports and ICMP traffic.
- It can emulate network services using script in Perl, shell or other way of interacting with the attacker.

10.2.1 Architecture

- *Configuration database* - Queried to discover the model paired with the destination IP address
- *Packet dispatcher* - analyzes input packets and checks correctness and integrity. Anything different from TCP, UDP and ICMP gets *discarded.*
- *Protocol manager* -
- *Personality engine* - computes a reply packet and updates it to guarantee coherence with the OS that the destination is expected to use
- *Optional routing component* - allows the routing of a packet to a real application

10.2.2 Research results

Honeypots provided an important amount of data to perform research on. Many statistics have been computed to produce estimations and interesting results, a ***UniPi*** *student presented a thesis on the topic* ☺.

Chapter 11

Countermeasures

20 - Ottobre

11.1 Robust Programming

Ideally it indicates a programming style focused on minimizing vulnerabilities and the impact of any vulnerability still exploitable.

Robust programming can be summarized with a few guidelines:

1. *Validate* program inputs aka input is evil
2. *Prevent* buffer overflow aka check sizes
3. A robust implementation minimizes any *information leaked outside* e.g. module, object, function ...
   - Logical pointers rather than physical ones
   - Validate any information that is exchanged
4. Check values transmitted to other functions (egress filtering)
5. Check returned results

Besides, it is important to focus also on **interaction controls**, robustness must be enforced on both malicious and erroneous behaviour.

11.1.1 Input validation

Usually input validation is achieved with a form of *default* **deny** by defining a legal input structure and discarding every input which doesn't satisfy it.

In case of string this may be done through RegEx, max length, ...

It is important that the checks to validate the input should be specified when the program is designed rather than after an attack; besides a check should be designed in an simple and readable way, to easily ensure its correctness. Some examples of input which usually must be validated are:

- Environment variables
- File names (blanks, .., /)
- Email addresses
- URL
- HTML headers/body
- Data

Memory allocation and strings length is a crucial aspect: only library functions with an explicit string length specified should be used[1], and in general, it is appropriate to allocate only the memory actually needed by a data structure according to its sizeto avoid leaving space to store dangerous values or inputs.

Speaking of functions, attention must be paid to a rigorous **interfaces definition** and to avoid making assumptions on relationships between input and output values of function; in other words, if a function A takes as input the value

---

[1]e.g. `strncpy()` instead of `strcpy()`

x returned by B, it must not be *asserted* that x is for sure a **valid** value, B should check the correctness of the input regardless of knowing how it was generated.

11.1.2 CWE - Vulnerabilities Ranking

[This article] by **CWE** (*Common Weaknesses Enumeration*) lists the most dangerous and frequent software weaknesses of 2023, based on data provided by *NIST*.

The scoring formula to calculate a ranked order of weaknesses considers the **frequency** a CWE is the root cause of a vulnerability with the **severity** of its exploitation. Both frequency and severity are *normalized* relative to the minimum and maximum values seen. **Frequency** is obtained by counting weaknesses occurences in the *National Vulnerabilities Database* (NVD), while **severity** is the average computed on the Vulnerabilities score in the $CVSS^2$ a given weakness is mapped to.
The **final weakness score** is computed by multiplying frequency and severity scores.

Biases and limitations

There are two biases which CWE doesn't take into account, which somehow negatively affect how valid CWE's scores are:

1. **Metric bias**
    (a) Indirect prioritization of implementation faults over design flaws
    (b) Prefers frequency over severity due to distributions of real-world
2. **Data bias**
3. (a) Only uses NVD data based on publicly-reported CVE Records
    (b) Many CVEs do not have sufficient details to assign a CWE mapping, omitting them from ranking
    (c) There may be an over-representation of certain programming languages, frameworks, or weakness-detection techniques

There also a few aspects which this scoring system cannot represent and should be taken care of. First of all, weaknesses that are rarely discovered will not receive a high score, regardless of the consequence of an exploitation. Weaknesses that begin with a root cause of a mistake leading to other mistakes, create a chain relationship. As we have seen, chains of mistakes/vulnerabilities/attacks are a key point in security, but CWE's scoring system treats any $\langle V_1, V_2 \rangle \land V_1 \rightarrow V_2$ as if V_1 and V_2 were independent i.e. $V_1 \nrightarrow V_2$.

11.2 Firewall

A **firewall** is a module to filter all the messages exchanged by *two* networks with a distinct security level; *all and only* the messages travelling on the wires connecting the two networks cross the firewall and therefore get filtered. A firewall works correctly under the assumption that a network has been split (*segmented*) into two **subnets**, and that it *correctly implements* a security policy, which should **not** define the policy by itself. Firewalls are usually **classified** on the known and manageable **protocols** and on their **implementation**.

11.2.1 Segmenting

Firewalling goes along with **segmenting** a network, which results in multiple subnets with different security levels whose interaction is determined by firewalls inbetween them. This architecture increases **robustness** by preventing an attacker from having **initial access** on an entire system and from freely performing **lateral movements**; besides this architecture perfectly integrates with **honeypot** deception mechanisms.

11.2.2 Classification

Firewall may operate on different levels of the TCP/IP stack and in different ways:

- Packet filtering firewall
- Circuit-level gateway
- Application-level gateway (aka *Proxy Firewall*): firewall which recognizes application level protocols and can make assumptions on it
- Stateful inspection firewall *Stateful* means that the firewall inspects also the contents of a communication and the properties related to the status of a connection.
- Next-generation firewall (NGFW)

---
[2] *Common Vulnerabilities Scoring System*

At level 3 (IP Packet Inspection) the firewall can check only the header of IP packets, while at level 4 (circuit level firewall) ...

TODO

11.2.3   Pros & Cons

- A Single device can filter traffic for an entire network
- Extremely fast and efficient in scanning traffic
- Inexpensive
- Minimal effect on other resources, network performance and end-user experience

- Since traffic filtering is entirely based on IP addresses and ports, it lacks broader context that informs other types of firewalls
- Doesn't check the payload and can be easily spoofed
- Not an ideal option for every network
- Access control lists can be difficult o set up and manage