COMPUTER SECURITY

Notes for the course of COMPUTER SECURITY at Politecnico di Milano [2023-2024]

Luca Gerin

Disclaimer

The content of this document is the result of the collection of the notes for the course Computer Security at Politecnico di Milano (academic year 2022-2023).

These notes may contain inaccuracies and are not guaranteed to be error-free, comprehensive, or up-to-date. Users are encouraged to cross-reference the information with other academic resources and consult their instructors or course syllabus for the most accurate and current information.

Some images used in these notes may have been sourced from various materials, including course materials and the internet. The inclusion of these images is intended for illustrative and educational purposes only. The creators of these notes do not claim ownership of the images and acknowledge the respective copyrights of the original creators.

Summary

1. Computer security basic concepts

Computer security, often referred to as cybersecurity or information security, is a field that encompasses practices, technologies, and measures designed to protect computer systems, networks, and data from unauthorized access, attacks, damage, disruption, or theft.

Two are the main characteristics that computer security aims to achieve:

- The **security** of a system, as the ability of that system to resist to internal and external threats, so not to be vulnerable to them and protect its assets.
- The **safety** of the people or entities from the environment and from the system itself, that if safe can be used without harm.

To engineer a secure system, we need to enforce security properties to every building block of it and consider it both in its components and as a whole, considering the potential consequences for the assets and users.

To protect a system, we need to satisfy basic security requirements, that can be summarized with the so-called **CIA Paradigm** for information security, that states three requirements:

- **Confidentiality**: information can be accessed only by authorized entities, while non-authorized attackers cannot access the data.
- **Integrity**: information can be modified only by authorized entities, and only in the way such entities are entitled to modify it.
- **Availability**: information must be available to all the parties who have a right to access it, within specified time constraints.

The first two requirements go in the direction of restricting access to some system, while the third goes in direction of ensuring easy access to resources. Availability is not less important than the other two requirements, as even the life of a person may depend on it (e.g., the availability of medical equipment in emergency situations).

There is a trade-off between Availability and the other two requirements, as improving it often requires giving up something else for what concerns Confidentiality and Integrity. The engineering problem consists in finding a good balance in order not to compromise the usability of the system while granting the needed security properties.

When evaluating the security of a system, it's important to consider the system not only in general, but to think about whether the system is secure enough in the context in which it is. The security of a system depends on its context, environment and on all the elements composing it.

An example is a vault door:

The system seems secure, until we look at the keypad, that controls the opening of the door. If, for example, there are numbers that are visibly more used or even pen signs on them to remember the code, we can easily conclude that the system is not secure, despite the thickness of the walls and door.

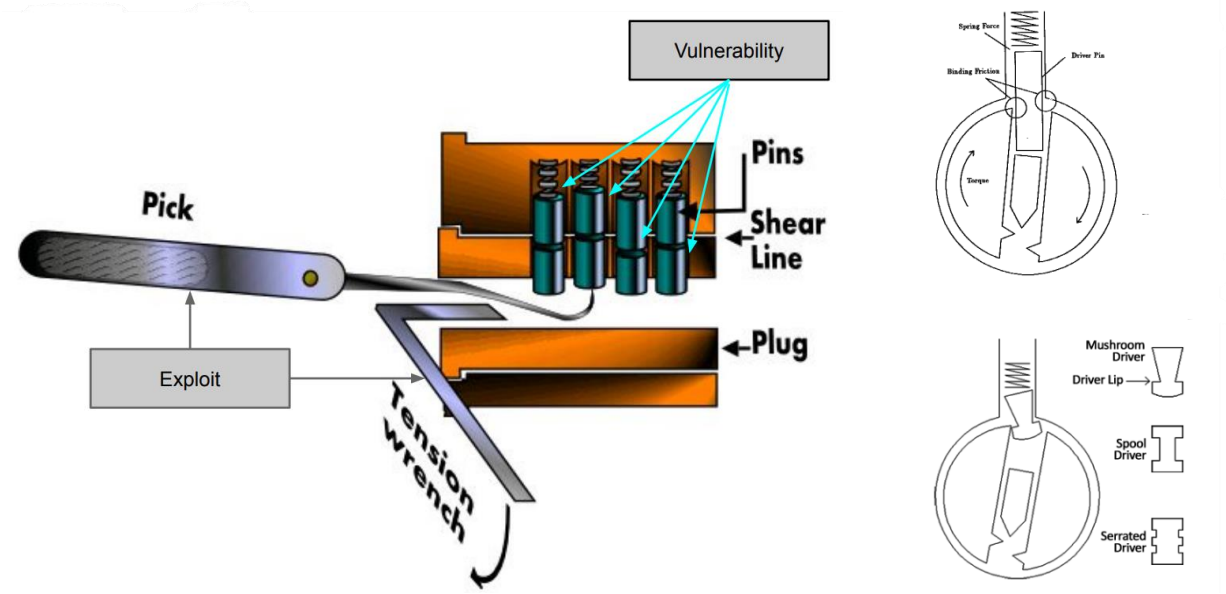
One small detail, that may lead to a way to attack the system, can compromise the entire system and jeopardize and make vain all the other defensive measures, no matter how strong they are. An attacker will most likely attack the weaker element of the system, and a system is thus as secure as the weakest path to breach it is.

When the system itself is instead apparently secure, it's important to take into account how it is used and the context in which it is inserted. A numpad without signs is useless if there are notes around with the code written on.

A **vulnerability** is something that allows to violate one of the constraints of the CIA paradigm, so something that allows to break the security of the system. Examples are pins in physical locks or a software failing to do something properly.

A vulnerability is not always an error in the defensive strategy but may just be something intrinsic to the system under analysis. For example, a vulnerability allowing to attack the availability of a smartphone is its battery.

An **exploit** is a specific way to use one or more vulnerabilities to accomplish a specific objective that violates the security constraints. Usually there are more possible exploits for the same vulnerability. Examples are the act of lockpicking or an input to a software leveraging a vulnerability.



In a lock, the vulnerability allowing lockpicking is intrinsic to how the lock works. As a consequence, it cannot be patched, and we cannot create a lock that cannot be lockpicked (otherwise it wouldn't be a lock). Some vulnerabilities cannot be patched.

But it is possible to make it extremely difficult for an attacker to exploit a vulnerability. We are trying to mitigate the problem given by the inevitable presence of a vulnerability by making the exploitation as hard as possible.

Using different kinds of pins that give no feedback to the attacker about the correctness of the position of each pin, using longer keys to leave less room to the attacker movements, or other similar measures, are all examples of how to make lockpicking more difficult.

Software vulnerabilities instead can usually be patched:

```
int i;
unsigned short s;

i = atoi(argv[1]);// parse command line parameter as int

if (i == 0) {     // check
    printf("Invalid number: value must be > 0\n");
    return -1;
}
s = i;
if (s == 0) {     //security check
    printf("Access GRANTED!\n");
}
```

```
$ gcc -o ex1 ex1.c
$ ./ex1 0
Invalid number: value must be > 0
$ ./ex1 10
$ ./ex1 65536  <~ exploit = the number "65536"
Access GRANTED!
```

Vulnerability:

- we check input on `int i` with `if (i == 0)`
- `int i` is guaranteed to be encoded in at least 32 bit (standard C)
- but `unsigned short s` can be encoded in 16 bits only ⇨ $2^{16} = 65536$
- then we (implicitly) convert an `int` to an `unsigned short`
- and do our "authentication check" on `s`

- **TODO:** can you find a **different** exploit for the **same** vulnerability?

The **security level** of a system depends on the threats to which the system is exposed and on the context, while the **protection level** refers to the measures put in place to protect a system and to how well an asset is protected.

For example, soldiers have a higher protection level than civilians, because they are armed and trained, but have a lower security level, because they operate in hostile environments. Civilians have fewer protective means but are exposed to less threats, so they have lower protection level than soldiers but higher security level.

When analyzing the security level, the context must be considered, starting with the possible threats insisting on the assets to protect, and the value of the assets.

An **asset** identifies something that is valuable for an organization, and needs to be protected. Examples of IT assets are hardware, software, data, reputation of a company.

A **threat** is a potential violation of CIA. A threat is everything that may eventually cause harm to the system, any attack that may compromise the security of it. Examples of attacks that may insist on IT systems are denial of service, identity theft or data leaks.

A **threat assessment** is the analysis of the possible threats that can configure attacks to the system, with the aim of identifying and mitigating potential harm.

An **attack** is an intentional use of one or more exploits with the objective of compromising a system's CIA property. Examples of attacks are attaching a malicious file to an email or picking a lock to enter a building.

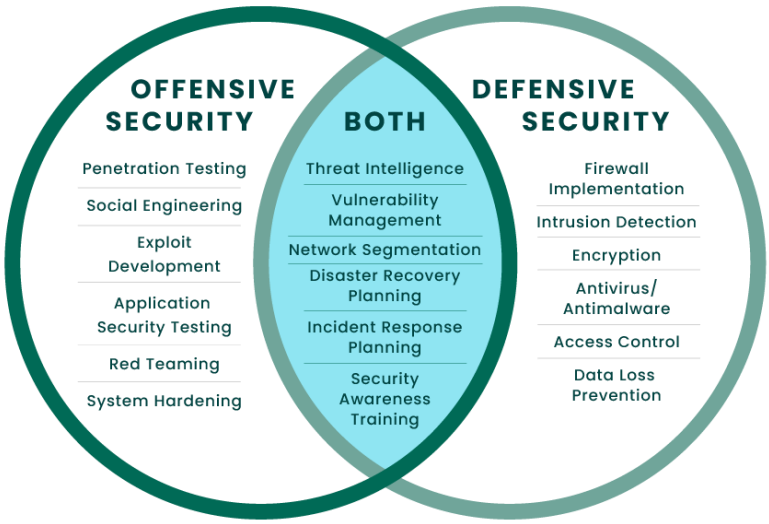
A **threat agent** (attacker) is whoever or whatever may cause an attack to occur. Examples of threat agents are malicious software or thieves.

An **hacker** is someone with an advanced understanding of computers and computer networks, and with willingness to learn "everything". Hacker does not necessarily mean a malicious attacker.

Malicious hackers are called **black hats**, from black and white movies in which bad guys used to wear a black hat.

Security professionals are instead called **white hats** (or sometimes ethical hackers), and they:

- Identify vulnerabilities,
- Patch vulnerabilities,
- Develop exploits,
- Develop attack-detection methods,
- Develop countermeasures against attacks,
- Engineer security solutions.



In any case, it is not possible to build perfectly secure systems. What a security professional aims to do is to build a system that is secure enough with respect to the vulnerabilities that there may be, the exploits that already exist, the assets to protect, and the threats that may insist on the system itself.

All these concepts can be summarized by the concept of risk. The security problem becomes then the problem of evaluating and mitigating risks, so the problem of risk management.

A **risk** is a statistical and economical evaluation of the exposure to damage because of the presence of vulnerabilities and threats.

Every time we want to build a secure system, we need to assess the risk by evaluating and managing the risks associated to it.

$$Risk = Asset \times Vulnerabilities \times Threats$$

To a high value of the three variables corresponds a high risk.

The first two variables for the risk, the asset value and the entity of the vulnerabilities, are controllable variables, meaning that we can act on them to lower the risk. Vulnerabilities can in fact be patched or made more difficult to exploit. The asset value cannot instead be reduced, but some other precautions can be taken, such as for example split the asset in more parts and dislocate them.

The threats variable is independent, and there is not much to do about it, so it is important to properly work on the first two.

An important aspect to consider is the cost of the defensive measures put into place. Security implies finding a balance between reducing the vulnerabilities and preparing for damage containment and the cost of doing such things. The goodness of this trade-off can define how secure a solution is.

Every security measure has in fact a cost, not only in economic terms, but consisting of both direct and indirect costs. Direct costs include: management, operational, equipment costs. Indirect costs, that are more relevant, include: less usability, slower performance, less privacy due to security controls, reduced productivity (users are slower).

Direct costs	Indirect costs
Management costs Operational costs Equipment costs	Less usability Slower performance Less privacy Reduced productivity

A security measure that is too strong can be counter-productive, because it might even push the users not to respect it! Spending more money in more theoretically advanced security measures against some attacks does not necessarily imply having more security.

For example, having a very expensive but unconfigured firewall is useless, it would be better not to have it instead of having it but not designing and configuring it properly. Having a complex authentication mechanism may slow down users, that might write passwords on stickies or use other strategies to make less effort, creating new vulnerabilities.

The cost of a system must be proportional to the risk evaluated.

In airport security, for example, security measures are in place against most known attacks but some exploits are intentionally ignored because they are very rare and with affordable cost, so it is not worthy to face the problem slowing down controls and everything else.

When deploying a system, it is of paramount importance to estimate the threats very well, and then be ready to update the threat assessment to adapt to new emerging threats.

When doing threat assessment, it is not possible to consider all the infinite number of threats, but, at some point, we need to stop evaluating possible scenarios. We are defining boundaries of trust, so that part of the system will eventually be assumed secure. In other words, at some point there needs to be a **trusted element**.

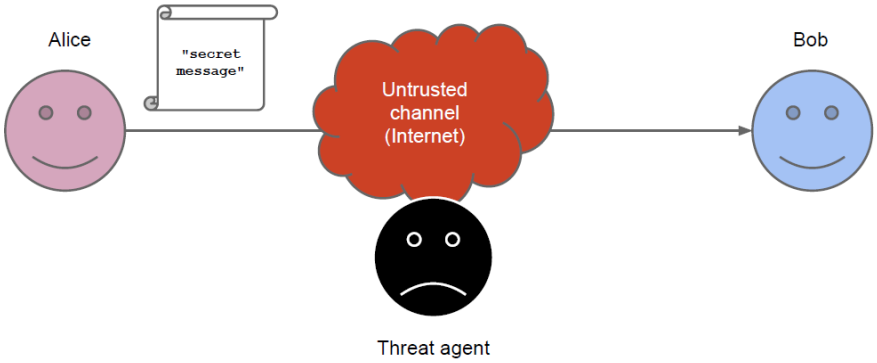
For example: we evaluate whether to trust the security officer, then the software, then the code of the software, then the compiler, then the bios, then the hardware, so the manufacturer, and so on. At some point, we have to define a boundary, where the system is considered secure, is trusted, even if theoretically it is not 100% secure. We cannot go on forever to not trusting stuff.

If the trusted element is compromised, then the entire security will e compromised, but this element is the one that we estimate to have the lower risk. The trusted element must also be as small as possible, so to be easily and continuously checked in order to verify that it has not been compromised.

2. Cryptography

Cryptography is the study of techniques to allow secure communication and data storage in the presence of attackers.

The problem to solve is that of two entities wanting to communicate (over an untrusted channel) but knowing that there is an attacker interested in the communication, or that of the user protecting its own storage and its data.



Features that cryptography is meant to provide include:

- **Confidentiality** of data: data can be accessed only by chosen entities.
- **Integrity/freshness**: detect/prevent tampering or replay attacks (when data is captured and re-used for example to impersonate someone else by re-enacting something that happened).
- **Authenticity**: data and their origin are guaranteed.
- **Non-repudiation**: data creator cannot repudiate the created data.
- Advanced features such as proof of knowledge or proof of computation.

The basic properties for cryptography are Confidentiality and Integrity.

The **proof of computation** consists in demonstrating that an entity has effectively done something because the energy he has spent is coherent with the task he has performed and what has been produced.

The **proof of knowledge** is instead a demonstration that an entity knows something, a certain secret, without explicitly saying it. This proof is at the base of the challenge and response protocol.

Challenge and response protocol in brief:

A: “I want to communicate”
B: “tell me the secret computed along with some random data I provide you to cover it (in a way it can be retrieved)”
A sends the secret with the computation required.
A: “Now you compute something based on the secret, the data you provided me and some other random data I now provide you”
B sends the secret with the computation required.
Now A and B, based on the received data, know if the interlocutor knows the same secret they do.

Evolution of cryptography

The word *cryptography* comes from the Greek word “kryptos”, hidden, and “graphein”, to write, and so is born as the art of secret writing.

In really ancient history writing itself was already a secret technique, as not everyone coming across something written could understand its meaning. Cryptography was born in Greek society when it became more common for people to read and hidden writing became a need. The primary uses were military and commercial uses.

Algorithms to hide some secret were designed by humans for other humans to decipher by hand.

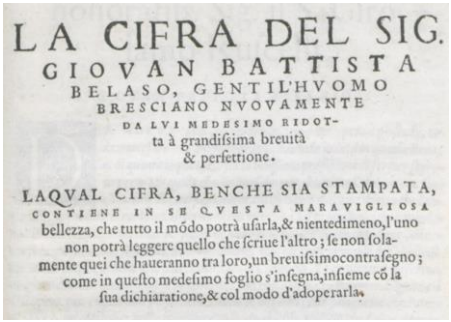
One of the first methods was the **scytale**, that is a tool used to perform a *transposition cipher*, consisting of a cylinder with a strip of parchment wound around it on which is written a message. A strip of leather is put around a prism and a message is written on it. Then, to read the same message from the leather, a prism of the same diameter is needed to put the strip around.

The ancient Greeks, and the Spartans in particular, are said to have used this cipher to communicate during military campaigns.

With this technique, there is no secret known by the two communicating parties, but what is secret is the technique that is needed to get the hidden message. Once the technique is known, it doesn’t take much effort to discover the secret message, even without having the right instrument (the prism) but going by trial and error.

In the ancient view, cryptography methods are just exercises from smart people trying to obfuscate messages. Once a certain cryptography method is beaten, the solution was just to change the method for a new one. It was a battle between cryptographers and cryptanalysts: the cryptographer ideates a secret method to obfuscate a text, and the cryptanalyst tries to figure out the method, breaking the cipher, so a new one is needed.

A turning point is reached by Bellaso (1553) when he separates the encryption method from the key. This way, also if the method is discovered, there is no need to change it, because an attacker still needs to discover some secret key to get to the hidden message.



LA CIFRA DEL SIG.
GIOVAN BATTISTA
BELASO, GENTILHVOMO
BRESCIANO NVOVAMENTE
DA LVI MEDESIMO RIDOT-
ta à grandiſsima breuità
& perfettione.

LAQVAL CIFRA, BENCHE SIA STAMPATA,
CONTIENE IN SE QVESTA MARAVIGLIOSA
bellezza, che tutto il módo potrà uſarla, & nientedimeno, l'uno
non potrà leggere quello che ſcriue l'altro; ſe non ſola-
mente quei che haueranno tra loro, un breuiſsimo contraſegno;
come in queſto medeſimo foglio s'inſegna, inſieme cõ la
ſua dichiaratione, & col modo d'adoperarla.

A consequence of this approach is that algorithms can be shared, because the strength of the cryptography heavily depends in the secrecy of secret key. The attacker will then know the algorithm, but not the secret.

A secret, non-public, algorithm (also based on a secret key) has more short-term security, until a vulnerability is found in it. If the algorithm is instead public, it can be studied and standardized by the community and it is easier to find vulnerabilities and patch them.

During medieval and renaissance times, there were studies about cryptography: Gabriele de Lavinde wrote a manual in 1379, Leonardo da Vinci used mirror writing. Cryptography has still mostly a military interest. Italian army general Luigi Sacco wrote a famous “Nozioni di crittografia” book in 1925, one of the last “non-formalized” exercises in cryptography.

In 1883, **Kerckhoffs** stipulated the **six principles of a good cipher**, that still apply to modern cryptography:

1. It must be practically, if not mathematically, unbreakable.

A cipher is unbreakable or perfect if it is a system that does not leak any secret independently from the time and the strength put in place by an attacker in order to decrypt it.

2. It should be possible to make it public, even to the enemy.

As previously discussed, this opens up to the possibility that the cipher is extensively tested and so vulnerabilities are discovered in advance.

3. The key must be communicable without written notes and changeable whenever the correspondent wants.

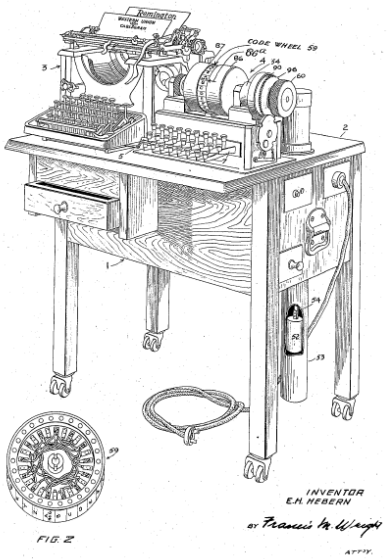
This principle refers also to the usability of the key, that must be simple enough to be remembered (so that it is not needed to write it somewhere) and must be easily manageable.

4. It must be applicable to telegraphic communication.
5. It must be portable and should be operable by a single person.
6. Finally, given the operating environment, it should be easy to use, it shouldn’t impose excessive mental load, nor require a large set of rules to be known.

The first two principles take into consideration the security of the algorithm, while the next four principles take into account the usability of the algorithm.

In modern history, the advent of machines allowed mechanical computation to change cryptography, because now machines could perform quickly computations that are very complex for humans.

A **rotor machine** was a system composed by a typewriter connected to a pin board where it exploited a rotor in order to substitute each letter with another one, in a revertible process.



The design of the rotor machine became popular in the second world war when it was utilized by the Germans for the Enigma machine.

Enigma was beaten in two ways:

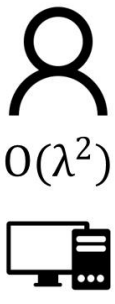
- Using algebraic permutation analysis by looking at the ciphertext
- By reverse engineering of the enigma machine

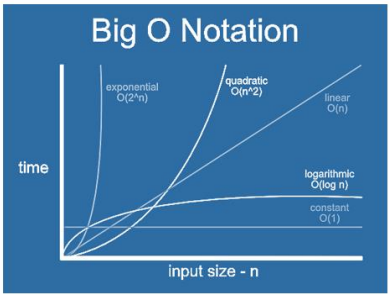
Two are the principles of modern cryptography.

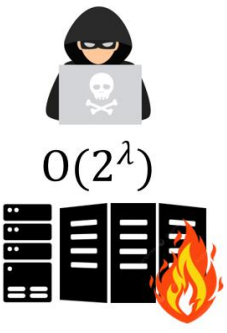
In 1949, Shannon proved that a mathematically unbreakable cipher exists.

In 955 Nash stated that computationally secure ciphers are enough. It's then enough that a cipher is secure from a computational complexity point of view, so that too many resources are needed to break it that it is not worth the cost or that is practically impossible.

Considering a cipher with a λ bit long key. The attacker effort to break the cipher would be $O(2^{\lambda})$, while the owner of the key takes only $O(\lambda^2)$ to compute the cipher. For large λ, this computational gap is unsurmountable.







What is known as **Kerckhoffs's principle** states that "The security of a cryptosystem relies only on the secrecy of the key, and never on the secrecy of the algorithm."

This means that:

- In a secure cryptosystem we cannot retrieve the plaintext from the ciphertext without the key.
- Also, we cannot retrieve the key from analyzing ciphertext-plaintext pairs.
- Algorithms must always be assumed known to the attacker (no secret sauce!).

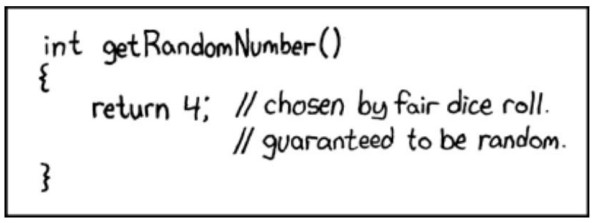
Key concepts in cryptography

A **random number** is a number that was extracted from a set with a random process. A single value cannot be "random", but random can only be the process generating it. So, randomness characterizes a generative process related to unpredictability.

A **random process** is one that cannot easily be predicted or replicated, in which it is not possible:

- To understand the mechanics of the generative process itself
- To guess the next element of the sequence or any information about it

If a value is generated from a random process, then it is impossible to distinguish this random process from a random distribution of values.



A **cryptosystem** is a system that takes in input a message (known as plaintext) and transforms it into a ciphertext with a reversible function that usually takes a key as a further input.

The use of "text" is historical, and today we mean "string of bits".

The **plaintext** space P is the set of possible messages $ptx \in P$. Plaintext is the message before it is hidden, and consisted of words in old times, while nowadays in a sequence of bits of length l.

The **ciphertext** space C is the set of possible ciphertext $ctx \in C$. Ciphertext is the string output of the cipher, the hidden message that is transmitted over a channel. The length of the ciphertext l' is not necessarily $l = l'$,ciphertexts are usually larger than the corresponding plaintexts because systems add random bits to the plaintext before encrypting it.

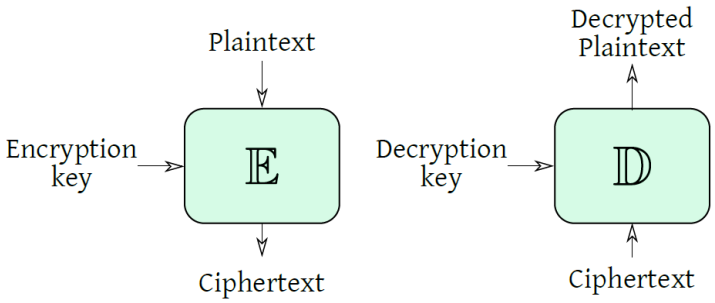
The **key** space K is the set of possible keys, of length λ.

Usually: $\lambda < l < l'$

An **Encryption function** $E: P \times K \to C$ is a function using the key to cipher the plaintext, obtaining a ciphertext.

A **Decryption function** $D: C \times K \to P$ is a function using the key to recover the plaintext from the ciphertext.

The encryption key can be different from the decryption key.



The property to ensure when evaluating these functions is correctness: the plaintext obtained by decrypting the ciphertext must be equal to the plaintext given as input to the encryption process.

Providing confidentiality

Confidentiality of the data is the first property that cryptography has the goal to guarantee, by ensuring that anyone not authorized is prevented from being able to understand the data.

In order to achieve confidentiality, a threat assessment must be carried out and all the possible attacks need to be analyzed in order to make the system resilient.

The possible attacker models are the followings:

- Plain eavesdropper (**ciphertext only attack**), in which the attacker reads the encrypted messages without knowing anything else.

Weak cryptography methods are vulnerable to this kind of attack as the attacker can search for patterns in cryptography and reverse-engineer it. For example, if a cipher uses only substitution, an attacker can recognize patterns in the text and use them to decrypt the message, possibly also retrieving the key.

- The attacker knows a set of possible plaintexts (**known plaintext attack**), so he knows some words that are probably encrypted and can try to reverse engineer the cryptography.

A known plaintext attack could be for instance when the attacker intercepts encrypted HTML messages like the *ping* message. If the attacker sniffs the conversation, he knows that at some point a message with the format *ping* + *host* will be sent.

A limit case for the known plaintext attack is the **chosen plaintext attack**, in which the attacker gains control of a machine sending messages and can decide what plaintext to send under the encryption, to observe the outcome. This way the attacker may deduce which ciphertext corresponds to which plaintext.

- **Active attacker**: the attacker can tamper with the data and observe the reactions of a decryption-capable entity.

An attacker tampering with the transmitted data can observe the changes in the stream and deduce information about the encryption; also, he can insert a chosen word in the ciphertext and see what happens.

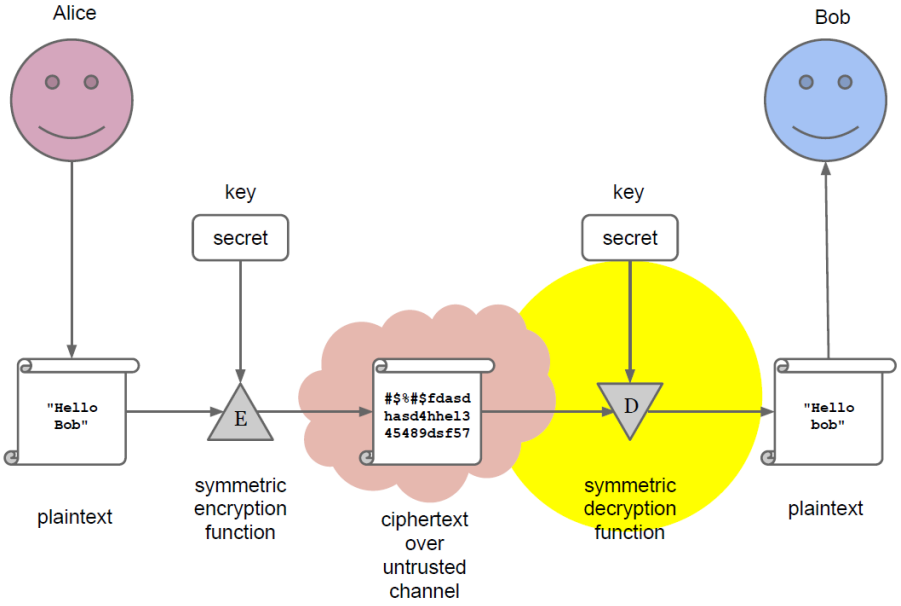
The limit case for the active attacker scenario is when the attacker can see the actual decrypted value, and so he can for example insert a word in the ciphertext and see how it is decrypted to deduce the encryption key. In order to counter this attack, the ciphertext must not change deterministically with the change of the plaintext.

Symmetric encryption

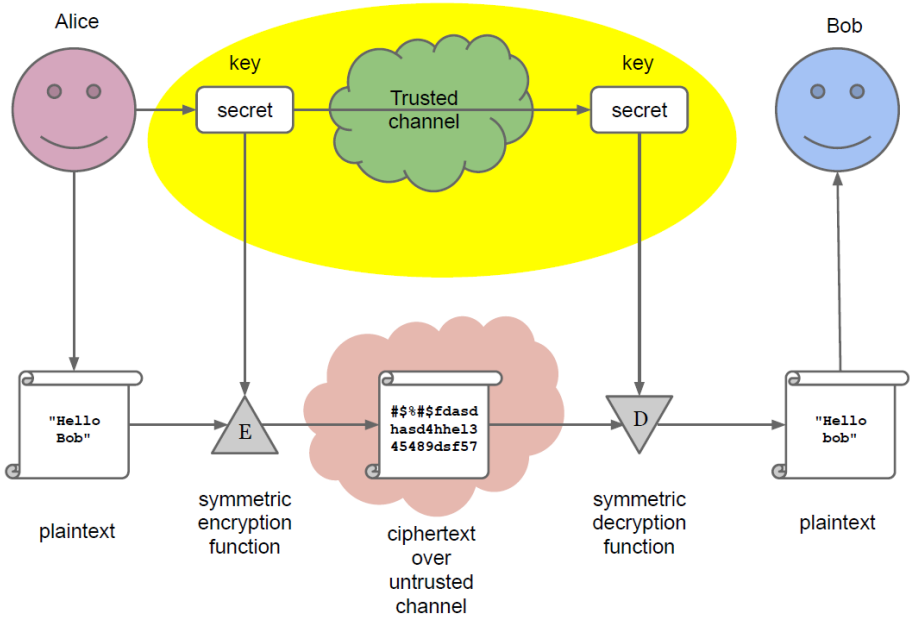
Symmetric encryption, or shared key encryption, or secret key encryption, is a method where the same key is used for both encryption and decryption of data.

A key is generated, and the plaintext is combined with it using an encryption algorithm, producing the ciphertext that is transmitted. The received ciphertext can be decrypted using a decryption algorithm and the same key.

An attacker not knowing the key should not be able to retrieve the plaintext.



Both parties involved in the communication must share the same key beforehand, and doing so can be an issue. A trusted channel is needed to communicate the key to the interlocutor without the possibility that this secret key is stolen.



A channel is always trusted only “for someone”; we cannot be sure that a channel is secure in general, but we can assume it is secure enough considering the current threat elements.

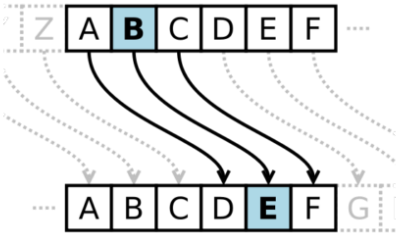
There is in any case a key management problem, as sending the key on the same channel that will be used for encrypted communication would not make sense and not be secure, so a costly off-band transmission mechanism is probably needed.

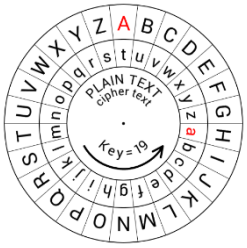
Another issue is the scalability issue: what if there are more than two elements between which to guarantee confidentiality by using a symmetric encryption scheme? Every pair of communicating elements would need a key for their communication, so with n element we would need $n-1$ pairs of keys.

A symmetric encryption algorithm is a “cocktail” made from a mix of two main ingredients: substitution and transposition.

Substitution is the practice of replacing each byte of a message with another according to some set of rules.

A toy example is the **Caesar cipher**, in which each letter in a sentence is replaced with the one following it by K positions in the alphabet (e.g. “secure” become “vhfxuh” with $k = 3$).



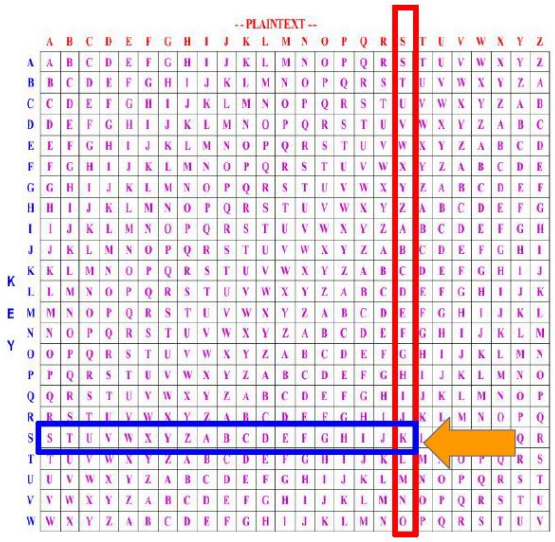


There are two main problems with the Caesar cipher, and in general with mono-alphabetic ciphers:

- The structure of the plaintext is visible in the ciphertext, so an attacker can observe it and so try different combinations to discover the plaintext. If the cipher is known (ciphertext only attack) then with 25 attempts at most, like the number of letters in the alphabet, and with 13 attempts on average, the attacker can discover the key and retrieve the plaintext. This is due to the fact that the keyspace is too small.
- The repetitions of the plaintext are visible in the ciphertext, because the same letter is always substituted with the same one. An attacker can study the frequency of substitution of letters to infer the plaintext, for example comparing the frequency of appearance of letters in the ciphertext with the one in the dictionary, and looking at combinations of letters appearing together with the same frequency.

Letter	Relative frequency in the English language[1]			
	Texts		Dictionaries	
A	8.2%		7.8%	
B	1.5%		2.0%	
C	2.8%		4.0%	
D	4.3%		3.8%	
E	12.7%		11.0%	
F	2.2%		1.4%	
G	2.0%		3.0%	
H	6.1%		2.3%	
I	7.0%		8.6%	
J	0.15%		0.21%	
K	0.77%		0.97%	
L	4.0%		5.3%	
M	2.4%		2.7%	
N	6.7%		7.2%	
O	7.5%		6.1%	
P	1.9%		2.8%	
Q	0.095%		0.19%	
R	6.0%		7.3%	
S	6.3%		8.7%	
T	9.1%		6.7%	
U	2.8%		3.3%	
V	0.98%		1.0%	
W	2.4%		0.91%	
X	0.15%		0.27%	
Y	2.0%		1.6%	
Z	0.074%		0.44%	

To solve the problem of repetition and structure that are visible in the ciphertext, multiple alphabets can be used together, in the **polyalphabetic ciphers**, or Vigenère cipher.



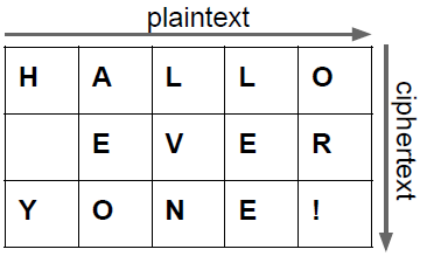
m=SECURE m=SECURE
k=SECRET k=SECRET
c=K c=KIELVX

By putting on the rows the alphabet of the key and on the columns the alphabet of the plaintext, we can use the combination key-plaintext to get in every cell the alphabet of the ciphertext to use for the encryption.

Ciphertext only attacks are no longer feasible on a polyalphabetic cipher, and it is not possible to determine which letters in the ciphertext come from the same letter in the plaintext as repetition and structure are now hidden.

The second practice that is the second ingredient of symmetric encryption is **transposition**, or diffusion, consisting in swapping the values of given bits.

A toy example is to write the plaintext in the rows of a matrix and extract the ciphertext by reading the columns.



m= HALLO EVERYONE!
k=(3,5)
c=H YAEOLVNLEEOR!

The key would be constituted by the dimensions of the matrix, so $K = (r, c)$ and we would need

$$r * c \sim len(\text{message})$$

so that the number of cells of the matrix is more than and similar to the number of bits (or letters) of the message to transmit. If the length of the message is small compared to the number of cells of the matrix, we would get bad results.

Repetitions and structure are hidden by this technique, but the keyspace is still small and so the key could be guessed.

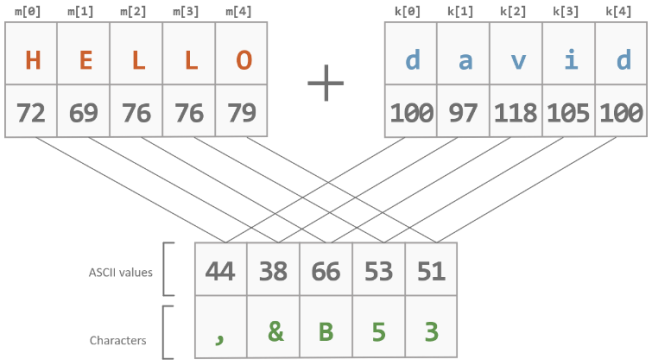
Perfectly secure cipher

As proved by Shannon, a mathematically unbreakable cipher exists. In a **perfect cipher**, the fact that the attacker sees the ciphertext does not change the probability that he guesses the plaintext.

$$\Pr(ptx\ sent = ptx) = \Pr(ptx\ sent = ptx | ctx\ sent = ctx)$$

In other words, seeing a ciphertext gives the attacker no information on what the plaintext corresponding to it could be, and the plaintext cannot be deduced from the ciphertext in any way.

If a perfect cipher is brute-forced, so all the possible combinations of keys are tried, the attacker can find a potentially infinite number of possible plaintext solutions all corresponding to the ciphertext, without being able to distinguish which is the right one.



In other words, with two different ciphertext, there would be the same probability of guessing the plaintext of one or the other, without being able to understand which plaintext corresponds to which ciphertext.

Theorem (Shannon 1949)
Any symmetric cipher $\langle P, K, C, E, D \rangle$ with $|P| = |K| = |C|$ is perfectly secure if and only if:

- Every key is used with probability $1/|K|$
- A unique key maps a given plaintext into a given ciphertext

$$\forall (ptx, ctx) \in P \times C, \exists! \, k \in K \; s.t. \; E(ptx, k) = ctx$$

The theorem says that a perfectly secure cipher is a cipher in which the key-space is as big as the plaintext and the ciphertext ones and the key is extracted random and fresh new for every encryption, with the same probability to be chosen as other keys and chosen independently of the plaintext. Moreover, a ciphertext obtained from a plaintext with a key cannot be obtained using another key and the same plaintext.

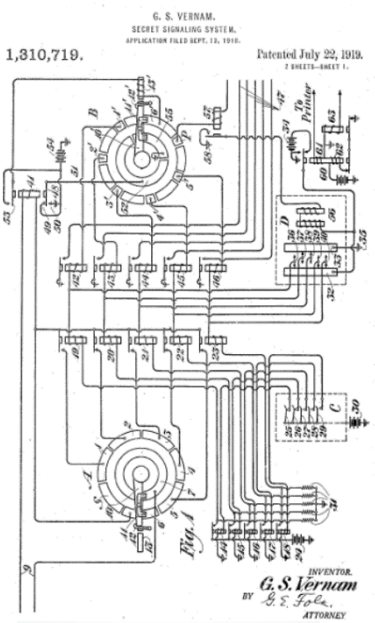
One implementation of a perfectly secure cipher is the **Vernam cipher** or **One-time pad**.

In it, the encryption function draws a uniformly random and fresh key out of the set of possible keys each time an encryption is needed and computes:

$$ctx = ptx \oplus k$$

So, the plaintext is *xored* with a random key with the same length to obtain the ciphertext. The one-time pad is a “minimal” perfect cipher because the key length is the same of the length of the message.

Vernam patented a telegraphic machine implementing $ptx \oplus k$ in 1919 and Mauborgne suggested the use of a random tape containing k, realizing a perfect cipher with two inputs: the plaintext from the user and some info randomly extracted to use as key.


G. S. VERNAM.
SECRET SIGNALING SYSTEM.
APPLICATION FILED SEPT. 13, 1918.
1,310,719. Patented July 22, 1919.
2 SHEETS—SHEET 1.
Fig. 1.
INVENTOR.
G. S. Vernam
BY
ATTORNEY

This cipher is however extremely costly to use and extremely inefficient.

Key management is a big problem because a key of the same length of the message is needed and for every message a new key must be pre-shared and then never re-used. If the same key is reused, information about the plaintext is leaked. However, due to the difficulty of changing key for every message, key reuse was common. Storing key material, changing keys and then getting rid of them is a nightmare and key theft or reuse led the cipher to be broken.

Another issue is that of generating keys that are random.

The name one-time pad comes from the fact that the key used to be written on a piece of paper called pad that could be used only once.

Such kind of perfect cipher cannot be broken but has the problem of the key storage and communication, and because of the usability cost it is used only in critical fields like the military field and the diplomatic one.

Computationally secure cryptography

Perfect ciphers do exist, however real-world algorithms are not perfect ($|K| < |M|$) and so can be broken, because every ciphertext-plaintext pair potentially leaks a small amount of information.

Brute-forcing is possible for any real-world cipher, when an attacker tries all the possible keys until one produces an output that seems the correct one.

Instead, perfect ciphers (one-time pads) are not vulnerable to brute force, because trying all the random keys will yield all the possible plaintexts, which are equally likely, so an attacker cannot decide which is the correct one.

Toy non perfect cipher	Toy perfect cipher
M=UGO K=+3 C=XJR Brute-forcing: try k=-1,-2,-3,…,-26 for each key will produce: k=-1 … M1= WIQ k=-2 … k=-3 … M3=UGO … The attacker gets to the secret by trying all possible keys	M=UGO K=+1+2+3 C=VIR Brute-forcing: for each letter try k=-1-1-1, -1-1-2 ,…,-26-26-26 will produce: M1= ADA M2=XKR M3=UGO …. The key is uniformly random extracted for each message and performs a different shift for each letter of the plaintext. Every possible string obtained with brute-force attack is equally probable of being the plaintext. It is not possible to know which plaintext corresponds to the one decrypted.

As said, all the real algorithms are vulnerable to brute-force, but this doesn't mean they are considered broken.

A real (non-perfect) cryptosystem is broken if there is a way to break it that is faster than brute forcing.

With “faster” we mean that the attacker can reduce the security margin of the apparatus of at least a lot of attempts. Then, the security margin of a system depends on how many time or attempts are needed to break a system.

Example of broken cipher:

Given a zip-compressed file encrypted with a secret 4-bytes key.
The (non-secret) encryption algorithm is to xor the plaintext with the key repeated many times in order to cover all the length of the file.

```
K(hex) =  AA BB CC DD .. .. .. .. .. .. ..   (repeat the key)
M(hex) =  50 4B 03 04 BA DA 55 55 .. .. ..   (and so on)
          XOR
C(hex) =  FA F0 CF D9 10 61 99 88 .. .. .. .. .. .. ..
```

The cipher apparatus is clearly not perfect, because the key is not at least as long as the plaintext and moreover it is always the same and not changed every time.

Brute-force could be a solution to discover the plaintext, but there is a faster way.
We can notice that different files’ ciphertexts all start in the same way. This is because sip files all have the same extensions and flags and the first bytes are always the same to identify the type of file,

```
michele@starkiller ~/Desktop
➤ xxd test.zip | head
00000000: 504b 0304 1400 0808 0800 bb74 3150 0000  PK.........t1P..
00000010: 0000 0000 0000 0000 0000 1400 0000 6974  ..............it
00000020: 2d36 3931 3439 3678 3038 3931 3635 2e70  -691496x089165.p
00000030: 6466 ccbb 6554 5ccb ba36 8abb bbbb 5be3  df..eT\..6....[.
00000040: eeee ee6e 8dbb bb6b 20b8 8510 9ce0 ee2e  ...n...k .......
00000050: 0182 bb3b c125 b806 bb49 d65e 7baf 7dce  ...;.%...I.^{.}.
00000060: face ddf7 8cef c71d 73d4 acaa 77be 56d2  ........s...w.V.
00000070: b3bb 9ef1 34a5 b2b8 2423 0b13 3b1c e5b7  ....4...$#..;...
00000080: 9dc9 5938 0e12 6612 4753 1b12 387e 7e38  ..Y8..f.GS..8~~8
00000090: 80ba b713 9004 20e1 e526 a5e6 66e2 0684  ...... ..&..f...
```

So, we have the ciphertext and we know the plaintext of the first bytes and can perform a known plaintext attack to recover the key or at least a portion of it if it is longer than the known plaintext (in this case, retrieve the rest of the key with brute-force)

```
K(hex) =  AA BB CC DD .. .. .. .. .. .. ..   (repeat the key)
M(hex) =  50 4B 03 04 BA DA 55 55 .. .. ..   (and so on)
          XOR
C(hex) =  FA F0 CF D9 10 61 99 88 .. .. .. .. .. .. ..
```

$K = M \text{ xor } C$

$K = 50\ 4B\ 03\ 04 \text{ xor } FA\ F0\ CF\ D9 = AA\ BB\ CC\ DD \rightarrow$

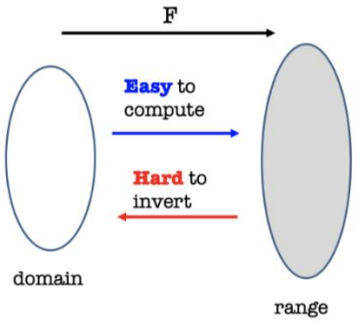
A **computationally secure**, or cryptographic safe, cryptography is one that in theory can be broken but is unbreakable in a practical perspective (for example it could require 10 000 years of computation to solve).

Remember: the attacker will always follow the path of less resistance

We are making a practical assumption, to consider secure ciphers built so that a successful attack is also able to solve a hard computational problem efficiently.

The time needed to break a cipher then depends on the length of the key and on the problem to solve.

On the other hand, these problems are very easy to verify if the solution is available. One-way functions are functions that are easy to compute from one side, but practically impossible to solve starting from the other side, so functions that are hard to solve but easy to verify.



NP problems like the ones employed are difficult to solve but easy to verify using a Turing machine equivalent in polynomial time.

The 3 hard to solve problem families employed in cryptography are the followings:

- Solve a generic nonlinear Boolean simultaneous equation set. It is required to solve a set of non-linear equations with Boolean values and OR and AND operations, basing the solution on outputs but without knowing nothing about the input. Partial knowledge of input and output leaves only enumeration of solutions as possible way to solve the problem. There is no way of solving this problem without enumerating all the possible inputs, and the time to solve the problem is exponential. If lambda bits are missing, an attacker requires 2^{λ} computations, if λ is the number of missing bits.

- Factor large integers / find discrete logarithms
  - Factor large integers: if p and q are two large primes, then computing $n = p * q$ is easy but given n it is very slow to find p and q, because all the primes must be tried until you get to the smaller between p and q. The problem requires brute-forcing over all possible values until the right one is found.
  - Discrete logarithm: if $y = a^x$ then $x = \log_a y$. Given x, a, p it is easy to compute $y = a^x mod(p)$ but knowing y it is difficult and really computationally expensive to compute x.
- Decode a random code/find shortest lattice vector. Decoding a random code requires the enumeration of all possibilities, while finding the shortest lattice vector is the problem of finding the shortest distance in a multidimensional space with n spaces, and the attacker would need to analyze all the dimensions.

The advent of quantum computer requires different problems that are hard for them.

Proving computational security

Modern cryptography standards require a **proof of computational security**.

The method to prove that a cipher is secure is the following:

1. Define the ideal attacker behavior, defining the main types of attackers to consider, their degree of knowledge and their granularity of action and what they have access to.
2. Assume a given computational problem is hard, so that the problem used by the cipher is not solvable faster than enumerating all the possible inputs.
3. Prove that any non-ideal attacker solves the hard problem, so that any attacker that does at least something can break the system.

An attacker with a different degree of knowledge would be able to break the cipher, but only by solving the hard problem. We are in practice trying to demonstrate that the only way to break the cipher is by solving the hard problem.

An attacker is represented as a program able to call given libraries with their functions. These libraries implement the cipher at hand, so we are simulating the attacker that sniffs the communication (calls a library implementing the cipher and sees the output).

We represent a security property as answer to a given question, for example a property could be "The attacker must not be able to obtain the plaintext".

The attacker wins the game if it breaks the security property more often than what is possible through a random guess (enumeration of solutions). So the attacker wins if he is faster than brute-force.

This way we can demonstrate the equivalency between a cipher under analysis and an ideal cipher. If it is the case, then the attacker cannot break the cipher faster than brute-force.

Number generators

If we were to use a Vernam cipher, we would need a key as long as the plaintext, whatever the plaintext. This is equivalent to say that we would need an infinitely long key that we can cut to the point needed for the plaintext.

To transform a Vernam cipher into something more practical, we aim to use a finite length key and to expand it as much as we need. So, we need a procedure that, given in input a finite key with a key length, will generate an arbitrarily long key such that nobody can distinguish it from an apparent random key.

We need to assume that the attacker can only perform $poly(\lambda)$ computations, so that the mechanism cannot be broken by an attacker in less than polynomial time.

This assumption is of paramount importance, because, if we remove it, then any algorithm to enlarge the key would be easy to break (Attacker has infinite time!). To distinguish between Random and pseudorandom I have to see all the possible outputs, and pseudorandom has 2^{λ} possible outputs (like input), while random has more.

A **Cryptographically Safe Pseudorandom Number Generators** (**CSPRNG**) is a deterministic function $prng: \{0,1\}^{\lambda} \rightarrow \{0,1\}^{\lambda+l}$ whose output cannot be distinguished from a uniform random sampling of $\{0,1\}^{\lambda+l}$ in $O(poly(\lambda))$.

l is the CSPRNG *stretch*.

So, with a CSPRNG we can obtain a longer key starting from a finite length one, and the obtained key is not distinguishable from a random string unless to enumerate all possible cases. If we have a CSPRNG with a given stretch that is not sufficient, we can repeat the procedure.

The *random function in C* is not computationally secure, as it has an integer seed as an input (so, 2^{32} possible inputs) and it can be statistically studied observing the output. These functions are

often used because they are fast, but the systems using them are not computationally secure and are always broken.

In practice, there are only candidate CSPRNGs, as we have no proof that a function Pseudo Random Number Generator exists. Proving that a CSPRNG exists would imply directly that $P \neq NP$, which is a hard problem.

There are two ways to build CSPRNGs. The first way is to build them "from scratch", which is possible, but it is not efficient because they are this way built as atomic functions.

The second and more practical solution is to build them by employing another building block, that is **Pseudo Random Permutations** (**PRP**s). Also for PRPs, there is no formal proof of their existence, because again this proof would imply to prove $P \neq NP$, which is a hard problem. PRPs are defined starting from Pseudo Random Functions (PRFs).

To understand PRFs, we need to see what a Random Function is. **Random Functions** (RFs) are randomly sampled functions from a set of all possible functions.

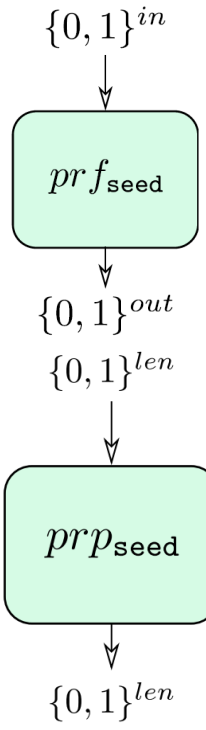
Consider the set $F = \{f: \{0,1\}^{in} \rightarrow \{0,1\}^{out}; in, out \in \mathbb{N}\}$
A uniformly random sampled $f \leftarrow F$ can be encoded by a 2^{in} entries table, each entry out bit wide.
$|F| = (2^{out})^{2^{in}}$

The set of functions is extremely large, because we need to consider also the domain and codomain of the variables. Unfortunately, storing the set of possible functions is impractical an impossible, but RFs are result of a random process.

The solution is to use **Pseudo Random Functions** (**PRF**s). Instead of using the infinite set of possible functions to pick one, PRFs employ a limited set of functions defined by a seed of length λ.

So, a Pseudo Random Function is a function $prf_{seed}: \{0,1\}^{in} \rightarrow \{0,1\}^{out}$ taking an input and a λ bits seed. The extraction output depends on the seed and is described by the value of the seed, but a PRF cannot be told apart from a Random Function in $poly(\lambda)$.

Unfortunately, Pseudo Random Functions are not easy to use.



Finally, **Pseudorandom Permutations** (**PRP**s) are bijective Pseudo Random Functions $prf_{seed}: \{0,1\}^{len} \rightarrow \{0,1\}^{len}$ that are cryptographically safe, as it is not possible to tell them apart from a random function RF in polynomial time $poly(\lambda)$. The function is uniquely identified by the value of the key (the seed) and the output is indistinguishable from random unless we have the key, as it is like having a permutation of all the possible $\{0,1\}^{len}$ strings.

A PRP can be seen as a cipher in which the seed is the key.

There does not exist any formally proven Pseudorandom Permutator (yet). The existence of a PRP would imply $P \neq NP$.

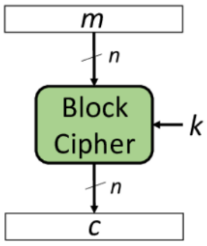
In practice, until 1997 PRPs were declared secure by the NSA. After that year, modern PRPs are the outcome of public contests in which a PRP is proposed and tried to be broken. What is tested

is whether the cipher can be broken faster than brute force, by identifying biases in the outputs, to which good designs are immune.

Block ciphers

Theoretical PRPs do not exist, so we consider a concrete PRP secure if it is computationally very expensive to break.

Concrete PRPs go by the historical name of **block cipher**.



They are considered broken if, with less than 2^{λ} operations, they can be told apart from a PRP, via:

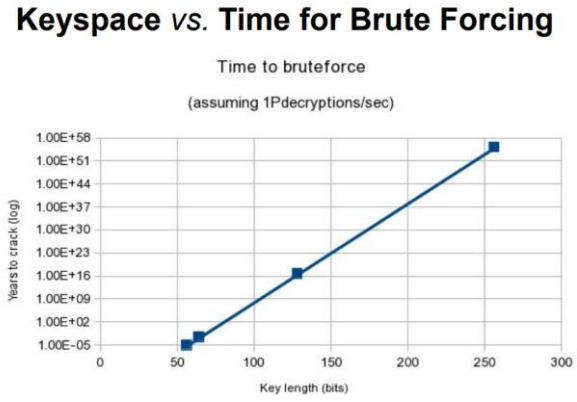
- Deriving the input corresponding to an output without the key
- Deriving the key identifying the PRP, or reducing the amount of plausible ones
- Identifying non-uniformities in their outputs

The key length λ is chosen to be large enough so that computing 2^{λ} guesses is not practically feasible.

The key length needs to be chosen carefully, because attack time is exponential in the number of bits, but we need to balance this number with the computational power we have and need to store the key, share it, and is available from the PRP that becomes inefficient for long keys.

To calculate the strength of a key, we can use the time needed to break a block cipher of length λ needing 2^{λ} operations to be broken:

- Legacy level security: at least 2^{80} Boolean operations
- 5 to 10 years security: at least 2^{128} Boolean operations
- Long term security: at least 2^{256} Boolean operations



Or we can use as metric the energy needed to crack a cipher, that would be equal to the energy needed to bring from $20℃$ to $100℃$:

- 2^{65} operations ≈ an Olympic swimming pool, that is a feasible amount

- 2^{80} operations ≈ the annual rainfall on the Netherlands, which is not feasible for most people
- 2^{114} operations ≈ all water on Earth, that is impossible today

Being the block cipher implemented in blocks, we need to choose the key length by following multiples of the base key length, at the cost of computational power and time.

Two are the widespread block ciphers:

The modern PRPs are based on **Advances Encryption Standard** (**AES**), which is the result of public contest ended in year 200 and is implemented in the hardware of a lot of processors (ARMv8 and AMD64m have dedicated instructions accelerating its computation).

AES is a very well optimized 128 bit block, with three possible key length: 128, 192, or 256 bits.

From 1976 instead, **Data Encryption Algorithm** (**DEA**, or DES) was widely used. It did not come from a contest but from recommendation of IBM and NSA.

Its core function is an S-box (key dependent substitution) using a 56 bit long key, that at the time was the security margin, while today it is easy to break with machines. So, DEA is not secure anymore.

DEA is still present in legacy systems, like satellites hardware, but officially deprecated. A patch to obtain a $\lambda = 112$ equivalent security margin consists in encrypting the data 3 times, providing a good mitigation but requiring more computation. Note that even encrypting three times, the security margin is only doubled.

History of Data Encryption Algorithm
- 1976: it becomes a standard in the US and its S-boxes are “redesigned” by the NSA.
- Late 1980s: differential cryptanalysis is discovered.
- 1993: it is shown that the original S-boxes would have made DEA vulnerable to differential cryptanalysis, whereas the NSA designed S-boxes were specifically immune to that.

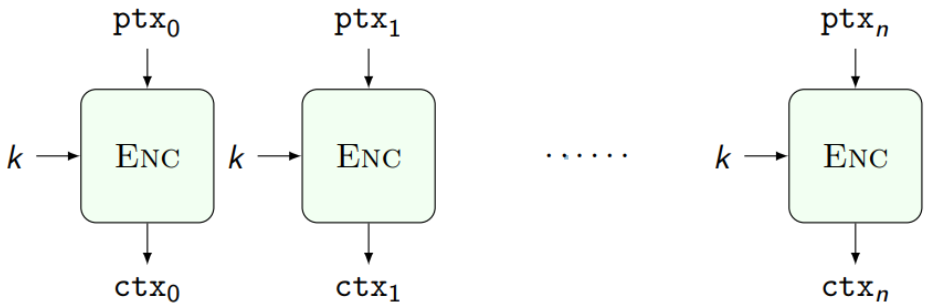
A theory is that the NSA knew about differential cryptanalysis since the 1970s.

Electronic Codebook

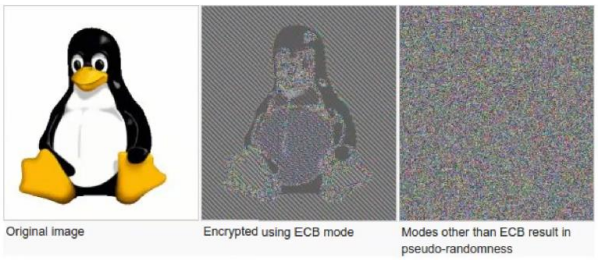
It is possible to use a 128-bit cipher to encrypt a plaintext long only 56 bits, so shorter than the block size of the block cipher.

If instead the plaintext is longer than the block size, extensions like split-and-encrypt would not be valid (not equivalent to CSPRNG). This bad strategy is the one used by the **Electronic Codebook** (**ECB**).

Using more block ciphers, we split the plaintext in parts and give them in input to the block ciphers using always the same key. Then, we concatenate the results to obtain the final ciphertext.



The problem lies in the fact that the same key is used for all the blocks. Consequently, the same input is deterministically translated by all blocks into the same output, so giving as input two equal strings to concatenate would result in two equally encoded ciphertexts to concatenate. This situation would not happen in a Vernam cipher.



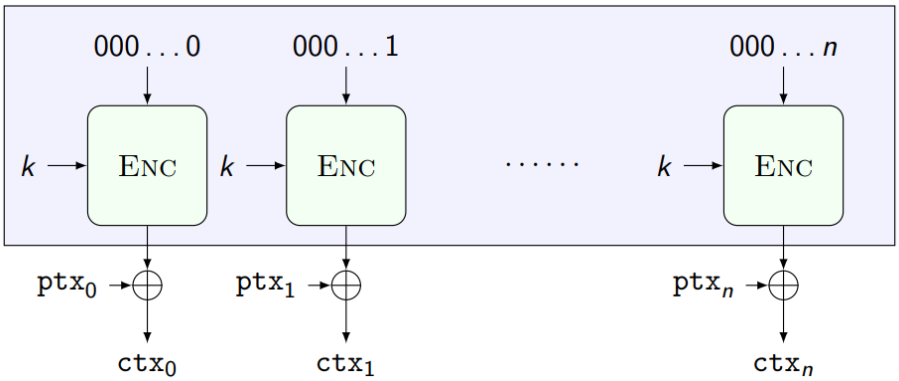
Patterns in the plaintext remain in the ciphertext, which is of course unwanted.

To solve this problem using the same scheme, we would need more keys, that is unpractical.

Counter (CTR) mode

If, instead of a plaintext, we provide as input to an electronic codebook a counter, we will obtain as output a random string. We can then XOR this random string with the actual plaintext to obtain a ciphertext.

This encryption scheme is called **Counter** (**CTR**) **mode**.



Confidentiality against Chosen Plaintext Attacks (CPAs)

Until now, we have protected confidentiality against ciphertext only attacks, but in practice an attacker can know some of the possible plaintext and may want to understand which is being sent.

Now we will consider an attacker model capable of carrying out chosen plaintext attacks, in which the attacker knows a set of plaintexts which can be encrypted and wants to understand which one is being encrypted.

We do not want the attacker to be able to tell which plaintext was encrypted out of the N he chose.

Even with counter CTR mode, to the same input corresponds the same output. Being the encryption deterministic, the CTR mode is insecure against chosen plaintext attacks.

Given the function $CTR(str1, str2)$, if the attacker knows $str1$ and $str2$,
he can call $CTR(str1, str1)$ to see the output of str1.

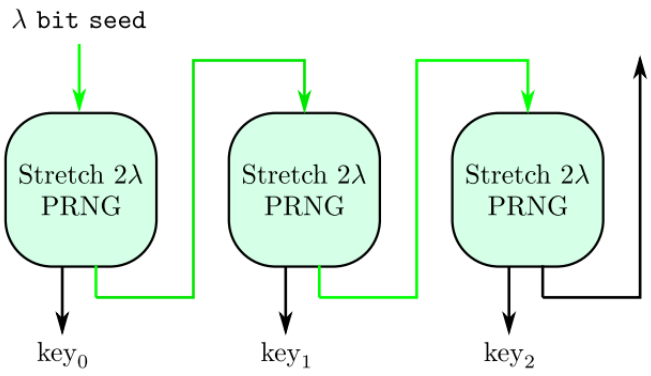
We want the ciphertext to be different at each encryption, even for the same string, in order to avoid CPA attacks, keeping the fact that the ciphertext must be decryptable.

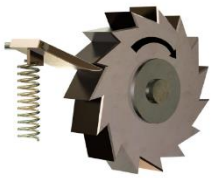
There are 3 possible solutions to obtain non-determinism but still have a decryptable result:

1. Rekeying: change the key for each block with a ratchet
2. Randomize the encryption: add (removable) randomness to the encryption (change mode of employing PRP)
3. Numbers used ONCE (NONCEs): in the CTR case, pick a NONCE as the counter starting point. NONCE is public.

The objective of **Symmetric ratcheting** (first solution) is to change the key for each block of the codebook, avoiding the key management problem given by having more keys. So, the aim is to generate more different keys starting from just one, and then use each one of these keys for the different blocks of the CTR mode.

In order to do so, the first shared key is inserted in the mechanism, and the PRNG stretches the key producing an output with doubled size. The first half of the output of each PRNG is used as key for encryption, xored with the plaintext, while the second half is used as input key for the next block.





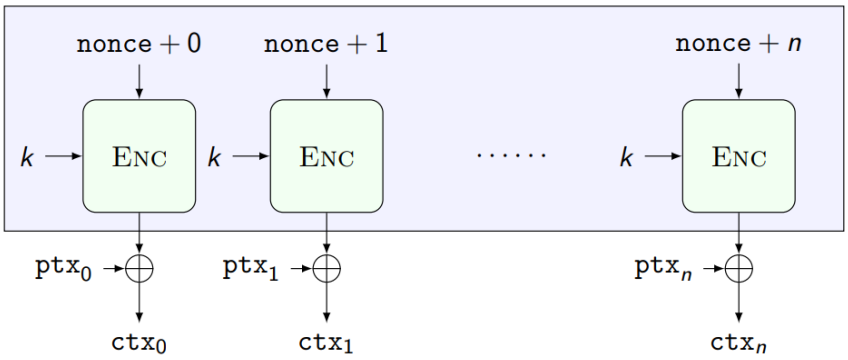
The key generation is deterministic, but it is impossible to reverse the procedure once the carried value (from one block to the next) is deleted, because the key is consumed once used. If an attacker retrieves a key, he is not able to retrieve the keys from the past (forward security).

The trusted element of the mechanism is the original starting key.

The third solution is called **CPA-Secure Counter (CTR) mode**, and exploits random numbers that can be used only once (NONCE).

A NONCE is a number used once and is randomly or pseudo randomly generated. Usually NONCEs are huge in size, so for it to be unlikely to extract the same NONCE more times.

The counter of counter mode starts as a NONCE and generates each time different bitstreams to be xored with the plaintext. As a result, the same plaintext encrypted twice is turned into two different, random looking, ciphertexts.



The NONCE can be shared in the same channel of the communication and be public (if not shared, it would not be possible to decrypt a message). The important thing is that the same NONCE is not used more than one time, so for different messages.

Providing integrity

It is possible for an (active) attacker to tamper with the ciphertext transmitted over a channel. This operation of changing the ciphertext without knowing the key causes evident changes to the plaintext.

Making changes to the ciphertext maps to predictable changes in the plaintext. A cipher in which a change in the ciphertext means determinist changes in the plaintext is *malleable*. This is a vulnerability and can be abused to build decryption attacks.

By flipping some bits one can obtain deterministic changes and play with the ciphertext to understand what belongs to the block and what to the padding by analyzing responses.

Malleability can also be used to make computation on the ciphertext and gain statistics without knowing the key.

Malleability can also be turned into a feature, the **homomorphic encryption**, allowing to make operations directly on the ciphertext without knowing its meaning and so preserving the privacy.

In any case, to avoid malleability it is possible to:

1. Design an intrinsically nonmalleable scheme (but it is non-trivial).
2. Add a mechanism ensuring data integrity (against attackers) so to know when an active attack has happened and know that the ciphertext has been tampered.

In order to guarantee confidentiality, we need to guarantee also integrity.

Confidentiality does not provide integrity (that needs an additional mechanism), but compromised integrity could compromise also the confidentiality. The two requirements C and I are orthogonal.

MAC

A mechanism to provide integrity is **Message Authentication Codes** (**MAC**) (contrarily to what the name suggests, MAC does not provide authentication, but integrity).

MAC works by adding a small piece of information, a tag, to a message, allowing to test for the message integrity.

Mac can be used independently of whether a message is encrypted or not, as it would be possible to send a plaintext along with a tag to check the integrity of that plaintext.

To allow to guarantee also confidentiality beside integrity, the tag of MAC should be computed based on and added to the ciphertext. Adding it to the plaintext and then encrypting is not good, as it may produce a leakage that may be used by the attacker. For example, in a known plaintext attack. Moreover, if the tag was computed based on the plaintext and the ciphertext is tampered, there would be no way to retrieve the plaintext to compute the tag and verify it.

The recipient of a message takes the ciphertext and computes the tag using the secret key, then compares the computed tag with the one received to see if they are equivalent or not, and knows if the data has been tampered with.

A MAC is constituted by a pair of functions:

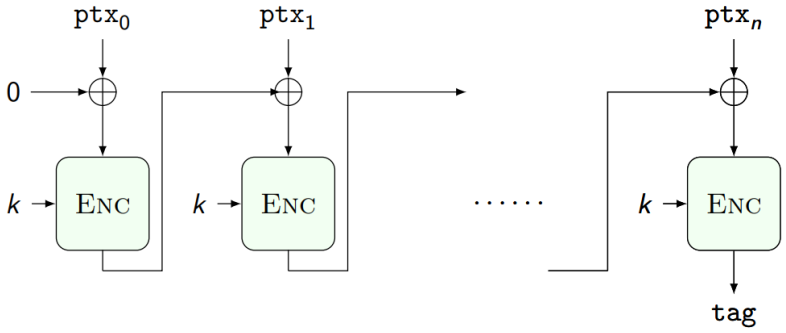
- COMPUTE_TAG(string, key): returns the tag for the input string, used by the sender of a message.
- VERIFY_TAG(string, tag, key): returns true or false, depending on whether the received tag is correct or not, so the integrity is preserved or not. It is used by the receiver of a message.

In an ideal scenario, the attacker is able to retrieve as many message-tag pairs he wants, but cannot forge a tag for any message, even for the messages whose tags he knows already. Moreover, he is not even able to separate a tag from a message and use it, for example attaching it to another message.

In order to work, MAC requires that both the tag creating entity and the verifying entity know the same secret key, which is the trusted element of the system. The only way in which an attacker can modify a tag would be by knowing the secret key. However, the presence of a secret shared key means that key management is an issue.

Both the tag creator and verifier are able to create a valid tag starting from a message and the key, and this means that non-repudiation property is not guaranteed by MAC, that is not good for digital signature (in general, every algorithm based on a shared secret cannot be used to grant authenticity and non-repudiation properties).

In practice, MAC is implemented by **CBC-MAC**, that uses block ciphers in Cipher Block Chaining (CBC) mode to generate a fixed-size output tag.



The plaintext or ciphertext, to which we want to add the tag, is split in parts of n bits each. The first part is XORED with the initialization vector, that usually is a sequence of zeros but can be a random sequence in some variations, and the result is fed to a block cipher that encrypts it using the secret key. The output of this first phase of the process is used instead of the initialization vector to XOR the second part of the message of which to compute the tag, and so on.

The entire text is read this way, but its hash can be used instead, in order to reduce computation.

Unfortunately, CBC-MAC guarantees integrity and is secure only for prefix-free messages, because the used PRPs are deterministic, and messages with the same prefix or so that one is prefix of the other are vulnerable. In facrt, CBC-MAC does not bind the MAC to the length or position of the message blocks. This means an attacker can prepend arbitrary data to a message and still produce a valid MAC.

Independence of Initial Chaining: In CBC-MAC, the chaining process starts with an initialization vector (often zero), and the first block of the message. If an attacker can control or predict this initial value, they can craft new valid MACs for modified messages.

Forging New Messages: An attacker can prepend any message P to an already authenticated message M. Suppose the MAC for M is known. The attacker can generate a new message $P \| M$ (where P is any prefix and $\|$ denotes concatenation) and compute the CBC-MAC for this new message if they can control the initialization vector or know the MAC of M.

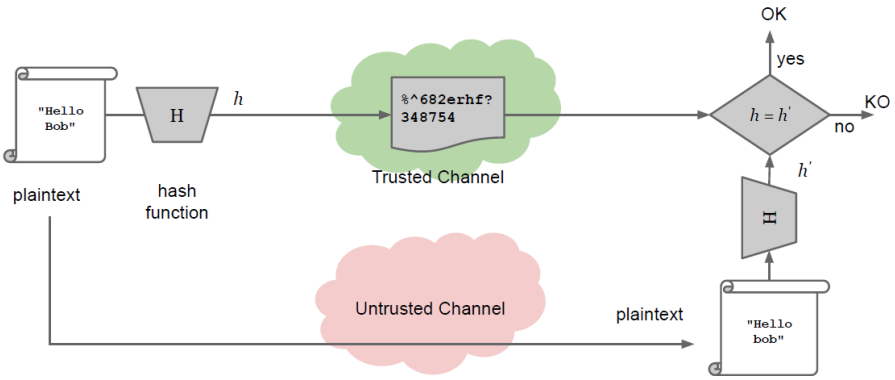
The partial solution to this problem is to encrypt the tag one more time (Encrypt-then-MAC), doing another pass of the CBC-MAC.

CMAC (Cipher-based Message Authentication Code) and HMAC (Hash-based Message Authentication Code) are more robust versions of CBC-MAC that include additional processing to ensure security against types of attacks related to the presence of a prefix or to the initialization vector.

A practical use of MAC is in **browser cookies**. HTTP is stateless, and cookies contain information useful to maintain a session opened. Of course, we want to avoid that a cookie is tampered with, otherwise an attacker can, for example, impersonate someone else or modify some information about the session.

MAC is used to check the integrity of cookies: the server runs COMPUTE_TAG(cookie, k) and stores both the cookie and the tag. This way, every time the cookie is sent back to the server, it can check its integrity (the user must not be able to modify or generate cookie and/or the tag).

Other uses of MAC include mitigating SYN-based denial of service attacks (SYN Cookies) and time-based two-factor authentication mechanism (TOTP/HOTP).



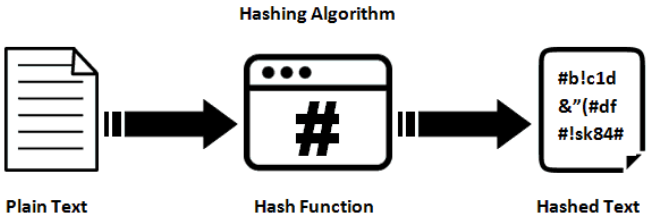
Cryptographic hashes

Testing the integrity of a file requires us to compare it bit by bit with an intact copy or read it entirely to compute a MAC, which is in both case not efficient. The solution to this problem is to use a short, fixed length string independently from the file size, representing the file itself.

By performing such compression, we must take into account that there is a limit, a lower bound, to the number of bits to encode a content without information loss (Shannon information theory), and we cannot go lower than this limit.

Instead of reading all the file to check for its integrity, we can use its hash, that is the result of a computationally hard problem (hashing) that, even If not theoretically unbreakable, is practically safe.

The objective of **cryptographic hash** is to efficiently compress, or map, a bit string of arbitrary length into a fixed length one, called *digest*, in a way such that it must be difficult to obtain the same output from different inputs.



Changing just one bit of the input string causes the change of the whole digest.

The domain of hash functions is bigger than the codomain, so it is theoretically impossible to avoid that two different inputs have the same output (a *collision* happens) but exploiting this fact in practice requires to solve some hard problems.

A cryptographic hash is a function $H: \{0,1\}^* \rightarrow \{0,1\}^l$ for which the following problems are computationally hard:

- given $d = H(s)$ find s (1st preimage):
  if the attacker knows the digest, it must be difficult to retrieve the original value.
- given s and $d = H(s)$ find $r \neq s$ with $H(r) = d$ (2nd preimage):
  given an input and its digest, it must be computationally hard to find another input, different from the one the attacker has, with the same digest.
- find r, s; $r \neq s$, with $H(s) = H(r)$ (collision):
  it must be hard to find two different inputs with the same hash.

The second problem depends on the first: if an attacker is able to solve the first, then he can easily solve the second, but not vice-versa.

In order to be able to say that it is “hard” to retrieve the original string given the digest and to repeat the process, the ideal behavior of a concrete cryptographic hash should respect the three properties:

- finding 1st preimage takes $O(2^d)$ hash computations guessing s.
- finding 2nd preimage takes $O(2^d)$ hash computations guessing r.
- finding a collision (another string with the same hash) takes $\approx O(2^{\frac{d}{2}})$ hash computations.

If an attacker can break one of the 3 properties by solving one of the hard problems (before the upper bound of 2^ ...), then the algorithm is broken.

Attacks to hash function include:

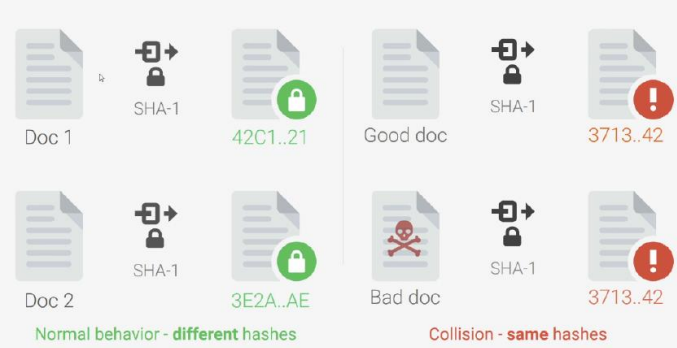
- Arbitrary collision or (1st or 2nd) preimage attack: given a specific hash digest, the attacker can find an input giving that digest or, equivalently, given a specific input the attacker can find another different input with the same hash. With a n sized digest hash function, collisions will happen in 2^{n-1} iterations.
- Simplified collision attack: the attacker can generate couples of inputs that have the same hash faster than bruteforce. Random collisions can happen in $2^{\frac{n}{2}}$ cases because of the birthday paradox (easier to find two people with the same birthday than finding a date and a person with that birthday). With a n sized digest hash function, collisions resulting from this attack will happen in $2^{\frac{n}{2}}$ iterations.

Concrete hash functions used nowadays include:

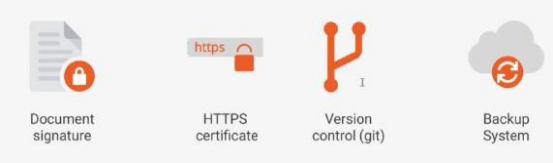
- **SHA-2** was privately designed (NSA), has a digest $d \in \{256, 384, 512\}$. It has not been broken yet, but shares some mechanisms with functions already broken in SHA-1 and MD5, as it is based on the same hard problems.
- **SHA-3** followed a public design contest (similar to AES), selected among ≈ 60 candidates, has a digest $d \in \{256, 384, 512\}$. It was born from the need to replace SHA-2 that will for sure eventually be broken, and is now the current standard.
  SHA-3 is based on different hard problems than SHA-1 and SHA-2, but is slower than SHA-2.

Deprecated hash functions include:

- SHA-1: $d = 160$, collision-broken (demonstrated that collision is obtainable in ≈ 261 operations), still needs enormous resources for current standards to be broken.
- MD-5: $d = 128$, horribly broken. Collisions in 2^{11}, public tools online available to break it. In particular, collisions with arbitrary input prefixes in $\approx 2^{40}$ (By adding the same 64 bit prefix to two documents, they will collide in 2^{40} operations).
  Digest of 128 bits generally means that we can expect a generic collision in 2^{128-1} and a simplified collision in 2^{64} attempts, but it was demonstrated that for MD-5 it is possible to obtain a collisions in 2^{11}. MD-5 can be easily broken by a smartphone in 30 seconds.



Collision broken means that an attacker is able to substitute something with something else malicious with the same hash. Typical targets are document signatures, HTTPS certificates, version control (Git), Backup system.



Hash functions have, in general, different uses:

- **Pseudonymized match**: store/compare hashes instead of values (e.g., Signal contact discovery), for example to compare passwords without knowing the actual password.
- **MACs**: to check integrity. Building MACs: generate tag hashing together the message and a secret string, verify tag recomputing the same hash. [hash(message+key)]
  However, MD-5, SHA-1, Sha-2 should not be used if the tag is simplu appended to the message, because this makes them vulnerable to some kinds of attacks. So, a better way is to use HMAC, a field-proven standardized way of combining message and secret that computes the hash twice.
- **Forensic use**: write down only the hash of the disk image you obtained in official documents. In order to prevent evidence from being tampered, the hash of the evidence files or disks is computed, put on a piece of paper and signed by authorities to be checked later. If data changes, also its hash changes.

Asymmetric encryption

Until now, we have considered confidentiality and integrity based on an algorithm based on a shared secret.

We would like to have some additional features that are convenient:

- Possibility to agree on a short secret over a public channel (a first solution is *Diffie Hellman* algorithm), eliminating among other things the need for a secure key management system (till 1976 the key was exchanged in person).
- Ability to confidentially send a message over a public authenticated channel without sharing a secret with the recipient
- Actual data authentication

The solution is to use an asymmetric cryptosystem.

> Before 1976: rely on human carriers / physical signatures
> DH key agreement (1976) / Public key encryption (1977)
> Digital signatures (1977)

The idea behind an **asymmetric cryptosystem** is to encrypt the plaintext with a public key (with which it cannot be decrypted) and decrypt it with a private key, instead of having just one key, relying on the hardness of a hard problem. This way, the encryption key can be shared and is public.

The first asymmetric system is RSA (Rivest–Shamir–Adleman), released in 1977.

Diffie-Hellman key agreement

The Diffie-Hellman key agreement has the objective to make two parties share a secret value using only public messages.

The attacker is assumed to be able to eavesdrop anything, but is passive and cannot tamper with the data.

Another assumption is the computational Diffie-Hellman assumption (CDH assumption), stating that the problem at the base of the algorithm is hard to solve.

CDH Assumption
Let $(G,\cdot) \equiv \langle g \rangle$ be a finite cyclic group, and two numbers a, b sampled uniformly from $\{0, \dots, |G|-1\}$ $(\lambda = len(a) \approx \log_2 |G|)$
Hard problem: given g^a, g^b finding g^{ab}costs more than $poly(log\ |G|)$
Best current attack approach: find either b or a (discrete log problem). An attack to this mechanism is equal to solve the discrete logarithm problem.
Note: to make the problem hard, the length of g must be big, as with this length growing the problem becomes harder and harder to solve.

In brief, the key agreement between A and B works like this:

- A: picks a from a set $\{0, \dots, |G|-1\}$ and sends g^a to B
- B: picks b from a set $\{0, \dots, |G|-1\}$ and sends g^b to A
- A: receives g^b from B and computes $(g^b)^a$
- B: receives g^a from A and computes $(g^a)^b$
- Being $(G,\cdot)$ commutative, $(g^b)^a = (g^a)^b$, now A and B know the same number and can use it or its hash as the shared secret

It is possible to deduce the secret from knowing g^a and g^b (an attacker can intercept them), but it is computationally very hard. However, breaking the problem at the base of Diffie-Hellman requires half the time that is needed to brute force the solution. The problems at the basis of asymmetric encryption are easier to solve than bruteforce attacks, that take $\mathcal{O}(2^{\lambda})$.

Conditions on the groups used to extract numbers making the problem harder
- A subgroup $(G,\cdot)$ of $(\mathbb{Z}^*_n,\cdot)$ (integers $mod(n)$), breaking CDH takes $\min\left(\mathcal{O}\left(e^{k(\log n)^{\frac{1}{3}}(\log(\log n))^{\frac{2}{3}}}\right), \mathcal{O}\left(2^{\frac{\lambda}{2}}\right)\right)$
- EC points w/ dedicated addition, breaking CDH takes $\mathcal{O}\left(2^{\frac{\lambda}{2}}\right)$

The problem at the base is a one-way trapdoor problem: the discrete logarithm problem. It is easy to compute in one direction, but very hard in the other.

Discrete logarithm problem
Given x, a, p, it is easy to compute $y = a^x \bmod p$
But, given y, it is difficult to compute x

Pick p prime and a number a that is the primitive root of p.
(primitive root = number such that raising it to any number between 1 and $(p-1) \bmod p$, we obtain each number between 1 and $(p-1)$; for example 3 is a primitive root of 7 because $3^6 \bmod 7 = 1, 3^2 \bmod 7 = 2, \dots, 3^3 \bmod 7 = 6$)

p and a are public and can be shared or transmitted over an unprotected channel.

Again, the protocol works like this:

Both parties A and B pick a number in the allowed interval, each one a number.

Secret number (undisclosed): They pick a number X in *[1, 2, ..., (p-1)]*

Alice X_A
Bob X_B

$X_A = 3$
$X_B = 1$

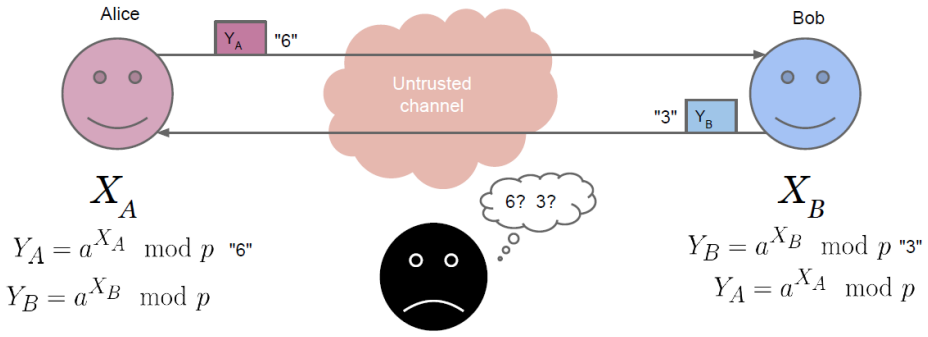
Both parties compute a value with a computation that is easy to compute for them but very hard to reverse.

Public number (disclosed to everyone): They compute:

Alice $Y_A = a^{X_A} \mod p$
Bob $Y_B = a^{X_B} \mod p$

$Y_A = 3^3 \mod 7 = 6$
$Y_B = 3^1 \mod 7 = 3$

This value is shared over a public channel.



An attacker monitoring the line will only see two random numbers.

The two entities A and B, instead, use their own keys and the received number to compute the same number and can use it or its hash as shared key.

- Since $Y_A^{X_B} \mod p = (a^{X_A} \mod p)^{X_B} \mod p = (a^{X_A})^{X_B} \mod p = (a^{X_B})^{X_A} \mod p = (a^{X_B} \mod p)^{X_A} \mod p = Y_B^{X_A} \mod p$
- Alice $K_A = Y_B^{X_A} \mod p = 3^3 \mod 7 = 6$
- Bob $K_B = Y_A^{X_B} \mod p = 6^1 \mod 7 = 6$

$K_B = K_A = K$

Thanks to the Diffie-Hellman algorithm, both A and B have a key to use for symmetric encryption. Everybody can listen to the conversation, but cannot compute the secret K.

Public key encryption

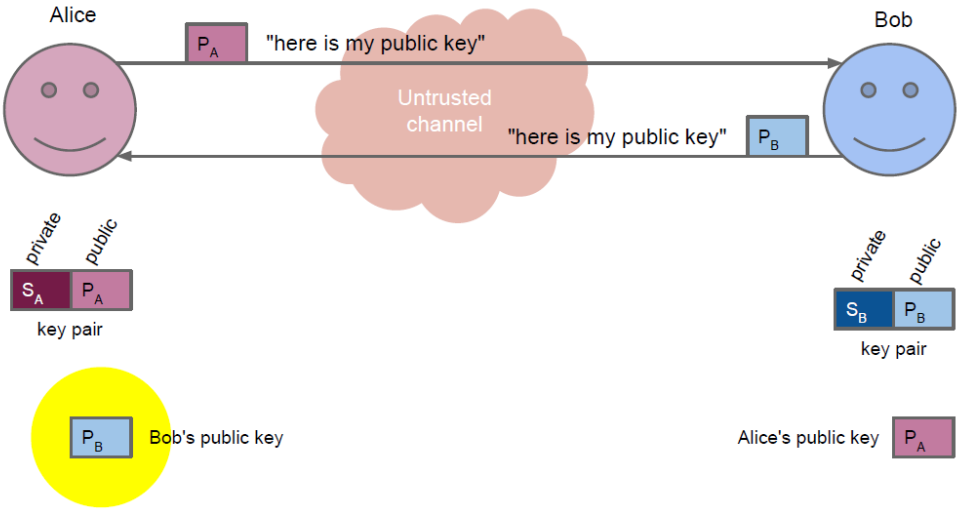
The concept behind asymmetric encryption is that of a cipher that uses two keys: what is encrypted with $key1$ can be decrypted only with $key2$ (and not with $key1$) and vice versa. If one of the two is chosen as public, it can be shared, while the other is kept private. The private key can not be retrieved starting from the public one, unless by solving a computationally hard problem.

This solves radically the problem of key exchange, because it is no longer needed to share a secret, but everyone just needs to share its public key without the need to know some secret key belonging to the other party.

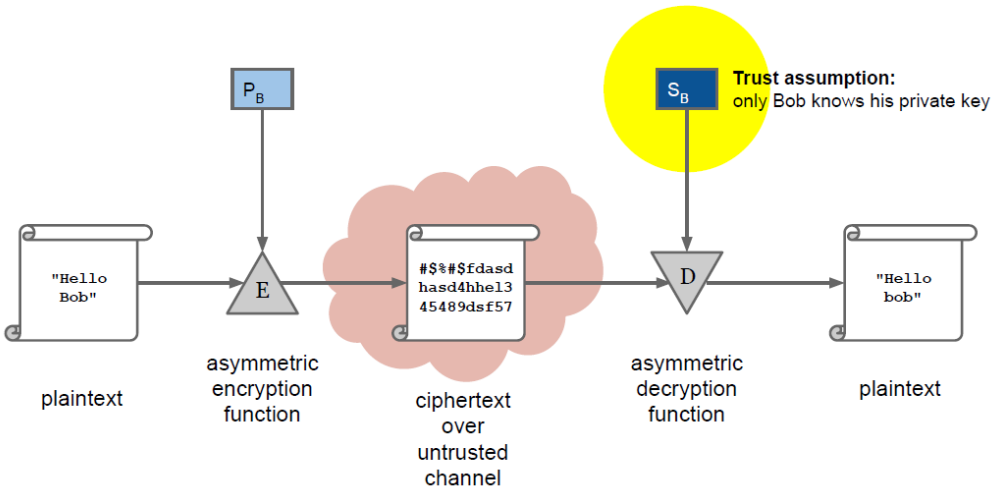
Asymmetric encryption is based on a one-way function with a trapdoor (problem easy to solve in one direction but extremely expensive to compote in the other), but are usually more computation-intensive than symmetric encryption algorithms.

Everyone has a pair of own public and private key.

The public key of everyone is shared, so everyone has (or can retrieve) the public key of the person he wants to communicate with.

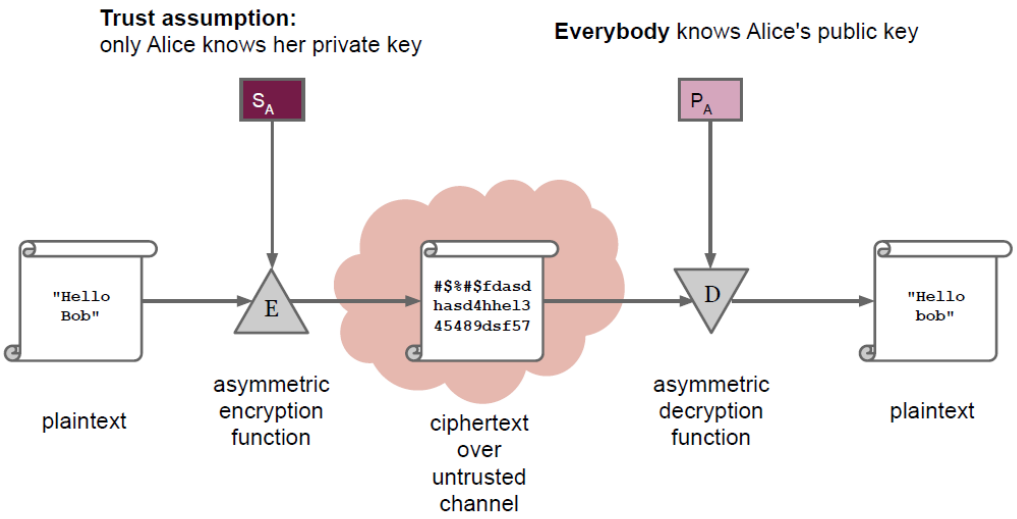


A encrypts a message meant for B with B's public key and sends this encrypted message over a public channel. When B receives the message, he can decrypt it with its own secret private key.

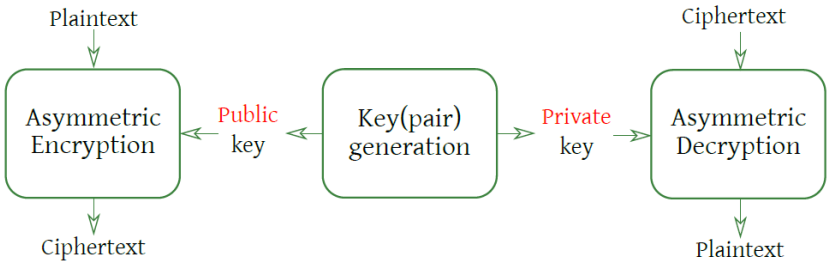


This way, confidentiality is guaranteed.

If we reverse the process and A encrypts something with his private key and sends it, then everyone can decrypt it with A's public key. This way, confidentiality is not guaranteed, but we are providing authentication, because we are guaranteeing that the document or message sent was encrypted with A's private key, that only A knows, so it was A for sure that encrypted it.



In any case, it is not possible to encrypt something with a key and then decrypt it using the same key, but the other key of the pair is needed.



The two computationally hard problems at the base are:

- Decrypt a ciphertext without the private key
- Compute the private key given only the public key

Widespread asymmetric encryption ciphers include:

- **RSA** (Rivest, Shamir, Adleman), 1977: based on the factorization of large numbers and with key sizes between 2048 and 4096 bits, the patent is now expired.
  There is no ciphertext expansion, meaning that input and output of the cipher have the same size, which is a computational advantage making RSA more efficient than other versions, but a disadvantage against know ciphertext attacks.
  Incidentally, if we do not change the public and private key between different encryptions, the encryption with a fixed key is a PRP (it is the only proven algorithm implementing PRP).
- **Elgamal encryption scheme**, 1985: based on the discrete logarithm problem and with key sizes in the range of $kbits$ or hundreds of bits, depending on the variant (and on the choice of the cyclic group), that can be shorter than RSA. It is more spread than RSA because it is not encumbered by patents. The ciphertext is twice the size of the plaintext, which is a disadvantage in terms of storage, bandwidth, and overall efficiency, making it less suitable for applications where resource optimization is critical.

In particular, RSA is, as said, based on the factorization of large numbers problem: if p and q are given, computing $n = p * q$ is easy but given n it is painfully slow to get p and q, especially when large prime numbers are involved. Factoring n is exponential in the number of bits n, while computation time for encryption grows only linearly in the number of bits n. At the moment of writing, a 512 bit RSA can be factored within 4 hours for around 100$, and the key sizes that are considered safe are those longer than 1024 bits, with the typical choices being 2048 or 4096.

The dimensions of the keys needed for asymmetric encryption and the security margin they provide with respect to the expenses and performance make so that we still often rely on symmetric encryption, especially for large amount of data or frequent messages.

Usually, asymmetric encryption is employed to share a secret and then that secret is used as key to perform symmetric encryption for the next messages.

A cautionary note on the security margin of asymmetric cryptosystems is needed. It's demonstrated that it's impossible to have perfect asymmetric cryptosystems, and this is because they rely on hard problems for which brute-forcing the secret parameter is not the best attack. It is in fact possible to achieve better performance than enumeration, that takes $\mathcal{O}(2^{\lambda})$.

For example, factoring a λ bit large number takes $\mathcal{O}\left(e^{k(\lambda)^{\frac{1}{3}}(\log(\lambda))^{\frac{2}{3}}}\right)$, that is easier than bruteforcing the key in terms of complexity.

Hence, the need for larger keys to make the solution of the hard problem more difficult.

Attention must be paid to the fact that comparing the bit-sizes of the security parameters instead of actual complexities is really wrong.

Year	Symmetric	Factoring (modulus)		Discrete Logarithm		Elliptic Curve	Hash
		(1)	(2)	Key	Group		
2032	95	2629	2144	168	2629	179	189
2033	96	2698	2208	169	2698	181	191
2034	**96**	**2768**	**2272**	**171**	**2768**	**182**	**192**
2035	97	2840	2304	172	2840	184	194
2036	98	2912	2368	173	2912	185	195

Key length is measured in bits both in symmetric and asymmetric algorithms. However, they measure different things: in the symmetric algorithms they measure the number of decryotion attempts needed, while in asymmetric algorithms the number of key breaking attempts.

It's possible to compare symmetric algorithms based on the length of the key, but you cannot directly compare asymmetric algorithms based on the key length, nor compare directly asymmetric and symmetric algorithms based on the key length.

So, what is the best key length that guarantees the same security for both symmetric and asymmetric kinds of algorithms? The rule of thumb states that:

- asymmetric encryption algorithms based on discrete logarithm or number factorization require a key length of around 1000 bits
- asymmetric encryption algorithms based on elliptic curve require a key that is at least twice long as the symmetric encryption one

For example, to obtain a security margin of AES 128 bit, we need a 256 bit elliptic curve algorithm.

Date	Security Strength	Symmetric Algorithms	Factoring Modulus	Discrete Logarithm Key	Discrete Logarithm Group	Elliptic Curve	Hash (A)	Hash (B)
Legacy (1)	80	2TDEA	1024	160	1024	160	SHA-1 (2)	
2019 - 2030	112	(3TDEA) (3) AES-128	2048	224	2048	224	SHA-224 SHA-512/224 SHA3-224	
2019 - 2030 & beyond	128	AES-128	3072	256	3072	256	SHA-256 SHA-512/256 SHA3-256	SHA-1 KMAC128
2019 - 2030 & beyond	192	AES-192	7680	384	7680	384	SHA-384 SHA3-384	SHA-224 SHA-512/224 SHA3-224
2019 - 2030 & beyond	256	AES-256	15360	512	15360	512	SHA-512 SHA3-512	SHA-256 SHA-512/256 SHA-384 SHA-512 SHA3-256 SHA3-384 SHA3-512 KMAC256

This report [8] describes recommendations from the German federal office for information security, BSI.

Date	Symmetric	Factoring Modulus	Discrete Logarithm Key	Discrete Logarithm Group	Elliptic Curve	Hash	
2020 - 2022	128	2000	250	2000	250	SHA-256 SHA-512/256 SHA-384 SHA-512	SHA3-256 SHA3-384 SHA3-512
2023 - 2026	128	3000	250	3000	250	SHA-256 SHA-512/256 SHA-384 SHA-512	SHA3-256 SHA3-384 SHA3-512

Key encapsulation

Key encapsulation or key wrapping is a cryptographic method where a symmetric key is securely transmitted using asymmetric encryption. This symmetric key is then used for encrypting the actual data. This way, key encapsulation combines the security benefits of asymmetric encryption for key exchange with the efficiency of symmetric encryption for data encryption.

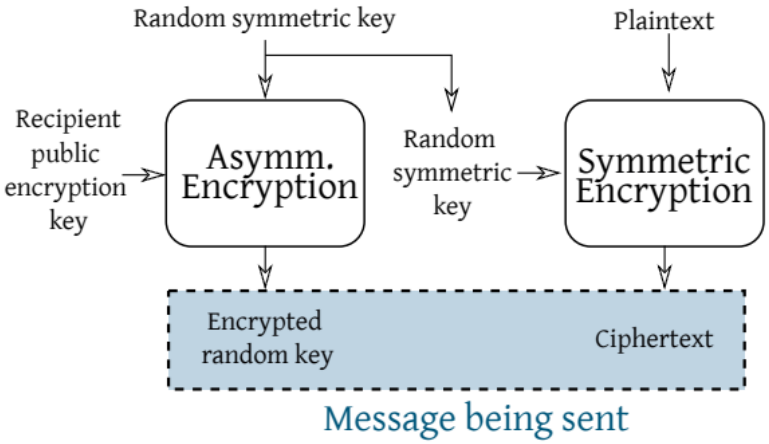
- Alice: generates a keypair (k_{pri} , k_{pub}), sends to Bob the public key
- Bob: gets a random secret $s \leftarrow \{0,1\}^{\lambda}$, encrypts it with k_{pub} and sends the ciphertext to Alice
- Alice: decrypts the ciphertext with k_{pri} and recovers s. Now both A and B know the secrete s

With respect to Diffie-Hellman key agreement, in this case It is one of the two entities to decide the value of the shared secret s, usually the one with a better random generator. To get better guarantees of key agreement, it is possible to repeat the procedure with swapped roles and then combine the two secrets.

Usually Diffie-Hellman is chosen in synchronous contexts, while key encapsulation is utilized in asynchronous situations.

As already mentioned, asymmetric encryption is inefficient compared to symmetric encryption. In principle, two entities could employ only an asymmetric cryptosystem to communicate, but in practice this approach would be extremely inefficient. Asymmetric cryptosystems are from 10 to 1000 times slower than their symmetric counterparts.

Hybrid encryption schemes employ asymmetric schemes to provide key transport/agreement and then symmetric schemes to encrypt the bulk of the data. All modern secure transport protocols built around this idea, for which confidentiality is ok, and integrity can be obtained by adding MAC to this mechanism.



Providing authentication

When relying for example on a public key, we need to be sure that the public key we are using belongs to the person we want to send the message to. An attacker could have substituted the public key with its own, so to be the only one to be able to decrypt a message meant for someone else.

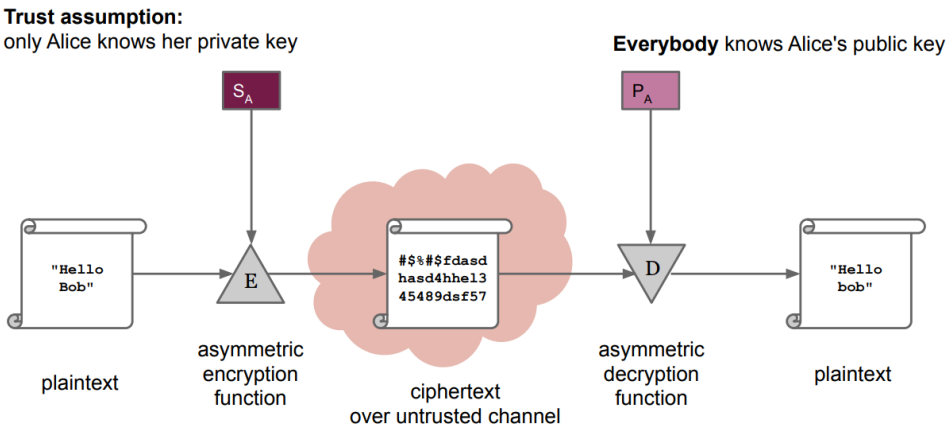
To build a secure hybrid encryption scheme we need to be sure that the public key the sender uses is the one of the recipient. To do so, we want to be able to

To build a secure hybrid encryption scheme we need to be sure that the public key the sender uses is the one of the recipient. To do so, we want to be able to verify the authenticity of a piece of data without a pre-shared secret. In summary, we need **authentication** of the data.

A **digital signature** is a cryptographic technique used to validate the authenticity and integrity of a digital message, software, or document. It provides strong evidence that data is bound to a specific user (authentication), without the need of any shared secret to check or validate the signature. In addition, proper signatures cannot be repudiated by the user (No-repudiation).

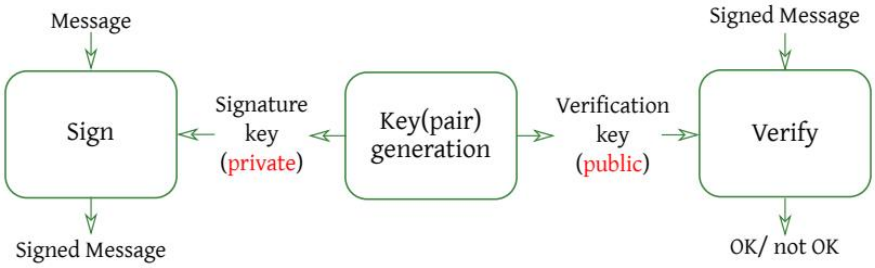
Digital signatures are asymmetric cryptographic algorithms.

The principle to provide authentication through digital signature is to use the private key to “sign” something and then everyone can use the public key to check if that something has been “signed” with the private key that only the owner of that key is supposed to know.



Digital signature is based on computationally hard problems:

- Sign a message without the signature (private) key: this includes splicing signatures from other messages
- Compute the signature key given only the verification (public) key
- Derive the signature key from signed messages

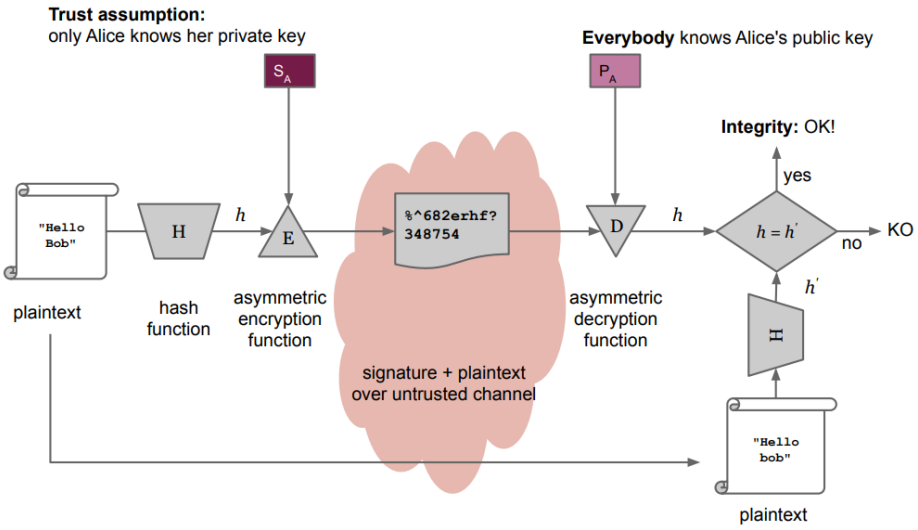


Of course, for this mechanism to work, public keys exchange must be secured. The problem will be tackled by PKI (public key Infrastructure), in which a digital certificate is used to bind the identity of a user with his public key, and is signed by a trusted entity, the certification authority.

In order to also verify integrity, it's possible to digitally sign the hash.

- A: sends the document in clear and the document's hash encrypted
- B: receives the document and the encrypted hash (the digital signature). He can decrypt the signature and compare the received hash and the one he creates from the received document

The digital signature is then a set of data encrypted with the private key, containing the hash of the document sent.



This way, both authenticity and integrity are verified. This is important, because an entity signing something cannot repudiate it, and definitely wants to avoid that the recipient receives something different than what has been sent.

There are two main widespread signature schemes:

- **Rivest, Shamir, Adleman** (**RSA**), 1977: standardized signature, based on the same hard to invert function that is used for the asymmetric encryption scheme RSA. Signing with RSA signature is definitely slower than verification (≈ 300 times) .
- **Digital Signature Standard** (**DSA**): another standardized signature, but the signature and verification take roughly the same time.

The difference between the two schemes lies only in the provided performances.

Digital signatures are used in practice to authenticate digital documents; for performance reasons, only the hash of the document is signed instead of the whole document. Signature properties are however now guaranteed only if both the signature and the hash algorithms are not broken.

Another use is that of authenticating users without the need of a password (a secret) but based on a challenge. The challenge and response scheme utilized works like this:

- The server has the user's public verification key (e.g. deposited at account creation)
- The server asks the client to sign a long randomly generated bitstring (challenge)
- If the client returns a correctly signed challenge, it has proven its identity to the server

Signed documents must not have dynamic parts. What is signed must be static, unable to automatically change, because the signature remains the same.

Horror story 1:
In 2002, several pieces of software performing digital signatures with legal value in Italy (originally DiKe by InfoCamere) allowed to sign MS Word documents containing macros or dynamic fields without notice.
A macro does not change the bit sequence of the document, so the signature does not change with the visualized document, and thus there was a mismatch between what was thought to be signed (the visualized document) and the actual signed object.
Current standard for digital signatures on human-intended documents target PDF/A and XML formats, and software to sign documents with a dynamical part show big alerts.

Both in asymmetric encryption and digital signatures, the public key must be bound to the correct user identity. If public keys are not authentic, a MITM attack is possible on asymmetric encryption, and anyone can produce a signature on behalf of anyone else.

The public key authenticity is however guaranteed with another signature, that needs in turn to be authentic.

A **digital certificate** is used to bind the identity of a user (or a machine) with his public key, and is signed by a trusted entity, the certification authority. They also specify the intended use for the public key contained (encryption or signature) and contain a time interval in which they are valid.

Digital certificate

- Key
- Identity
- Type of use
- Start of validity
- Expiration of validity

It is not possible to delete a signature, but one can invalidate his certificate.

This way, for example, if someone stole a pc with a signature on it, the owner could invalidate the signature, and the entities finding that signature would recognize it as invalid because the certificate was revoked.

A digital signature does not ensure that a file was authored by someone, but only that that file was encrypted with a certain key associated with that someone.

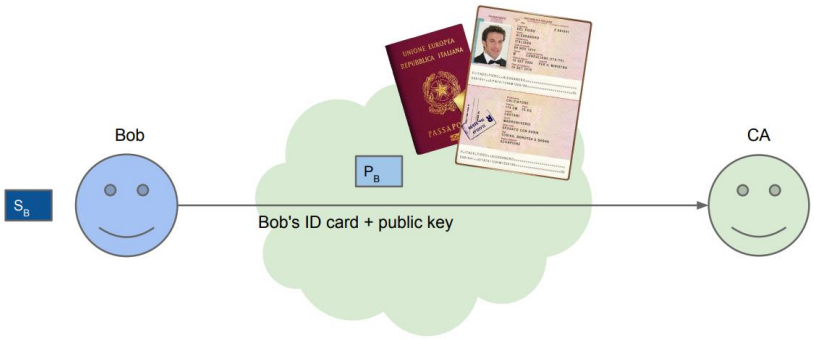
Public keys exchange must be secured, and a **public key Infrastructure** PKI has the task to associates keys with identity on a wide scale.

A PKI uses a trusted third party called a certification authority (CA) that digitally signs the digital certificates binding public keys to identities,

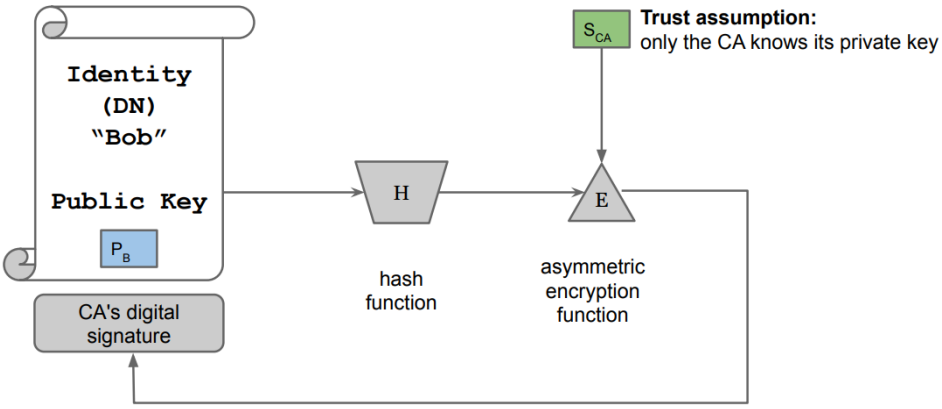
However, at this point, another public key is needed: the one of the Certification Authority, because we need to trust it. We could get it somewhere (from another CA, and immediately have the problem of another certificate for which we need another key. And we can go on and on along the *certification chain*.

We go along the certification chain till we reach a point we consider trusted. We trust only a top-level certification authority or Root certification authority which provides a self-signed certificate.

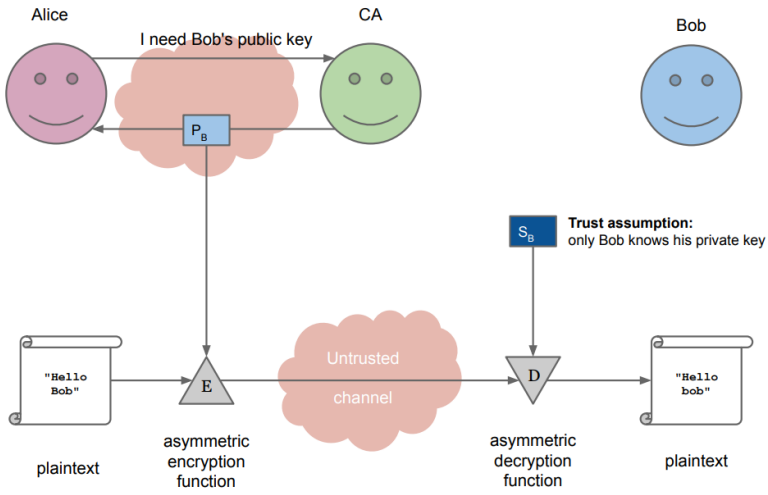
The process more in detail works like this: at the beginning, Bob sends his own ID and public key to the certification authority.



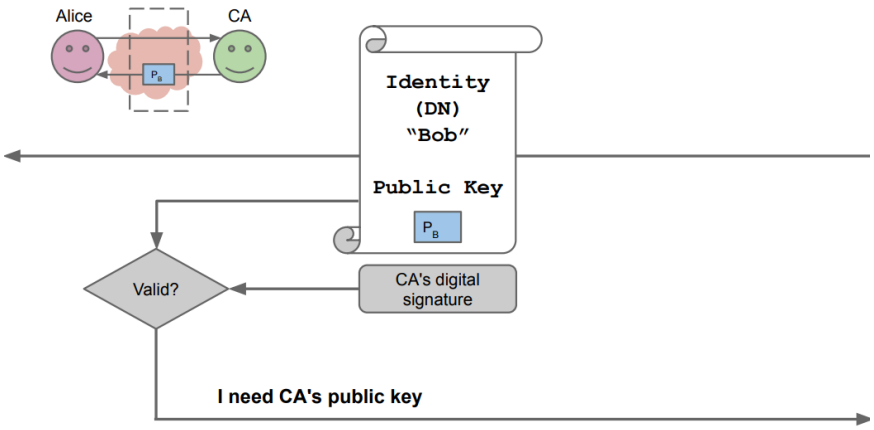
The CA will take this information received (identity and public key binding) and digitally sign it: the hash is computed, encrypted with the private key of the CA and attached to a digital certificate. This certificate is stored and made available for people or machines to retrieve, so they can verify the public key binding with Bob identity.

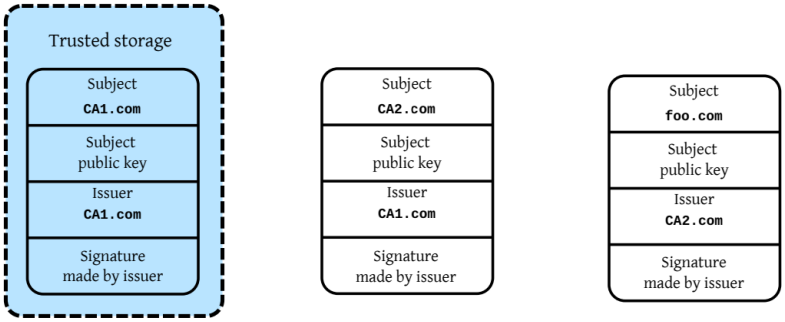


When someone, called for example Alice, needs Bob's certificate, he asks the Certification authority, that is the trusted third party signing the certificate, and obtains the certificate with the public key of Bob.



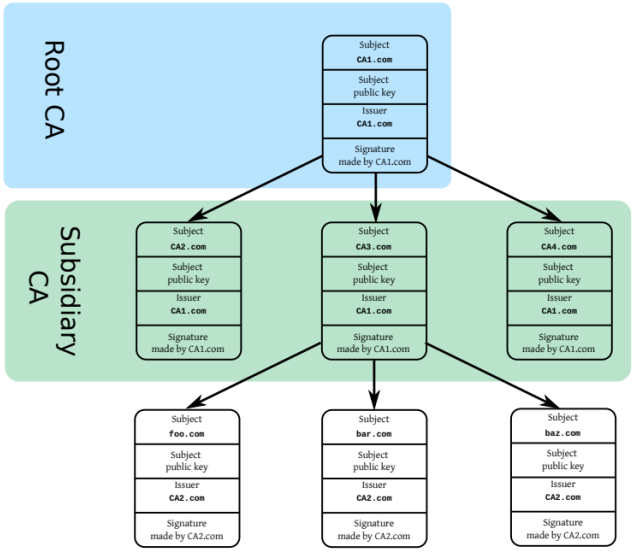
To check the validity of the public key retrieved, Alice needs to verify the integrity of the certificate and authenticate the CA, so she needs the public key of the CA.





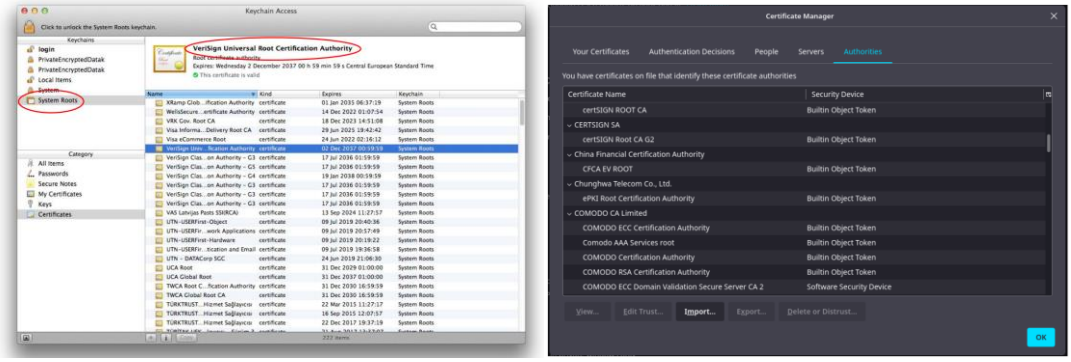
Alice would need to check the public key of the CA, and to do so she must use another CA. And what about this other CA? How to verify its public key? Alice could go on and on potentially forever in the "certification chain".

The solution to avoid searching forever in the certificate chain is to determine a trusted element, that is a certificate signed by a root certification authority. A **top-level CA** (or *root CA* or *source CA*) uses a self-signed certificate and cannot be verified but is the trusted element of the system.

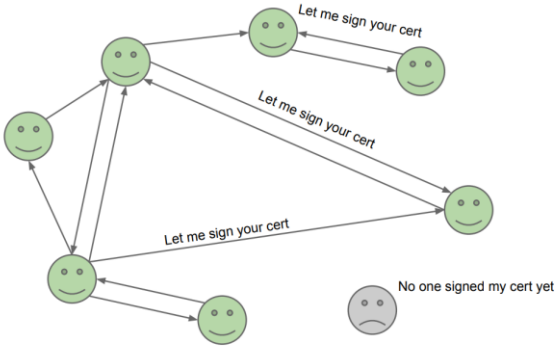


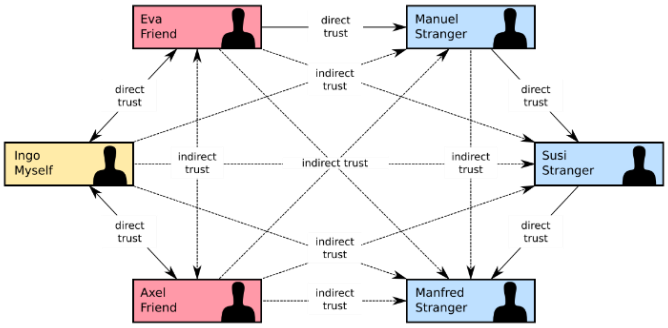
So, the certificates are retrieved from CAs that are authenticated following their hierarchy in a certification chain up to the trusted element, the root CA, that is trusted a priori. By trusting the root CA, we trust all the CA whose identity depends on it.

Usually, the root CA is installed in the systems in a dedicated secure location, but it can be released by an authority such as the state, a regulator or the belonging organization management.



There are solutions in which trust is instead decentralized. The *PGP* (Pretty Good Privacy) *web of trust* is a decentralized trust model used to authenticate the association between public keys and their owners in the PGP encryption system. Unlike a centralized trust model where a single entity, like a Certificate Authority (CA), vouches for the identity of key holders, the web of trust allows users to establish and verify trust through a network of individual endorsements.





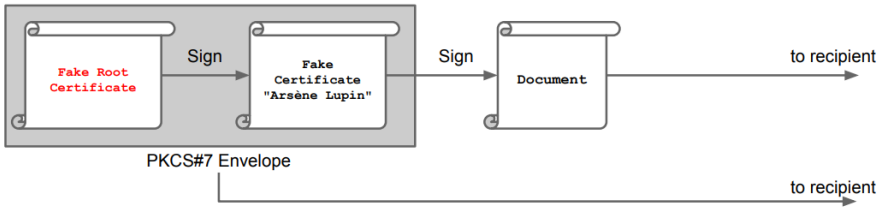
Back to the case in which a signature is stolen, it is not possible to destroy the signatures, but they can be made invalid through the revocation of the certificates. To revoke a certificate, the owner of a signature informs all the Cas that his key is not valid anymore, so they can update their Certificate Revocation List (CRL), that is a list of digital certificates that have been revoked by the issuing Certificate Authority (CA) before their scheduled expiration date, making them invalid for secure communications.

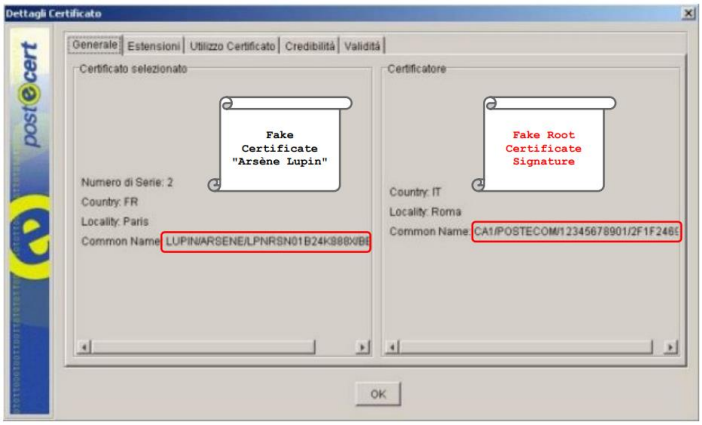
Verification Sequence for Certificates

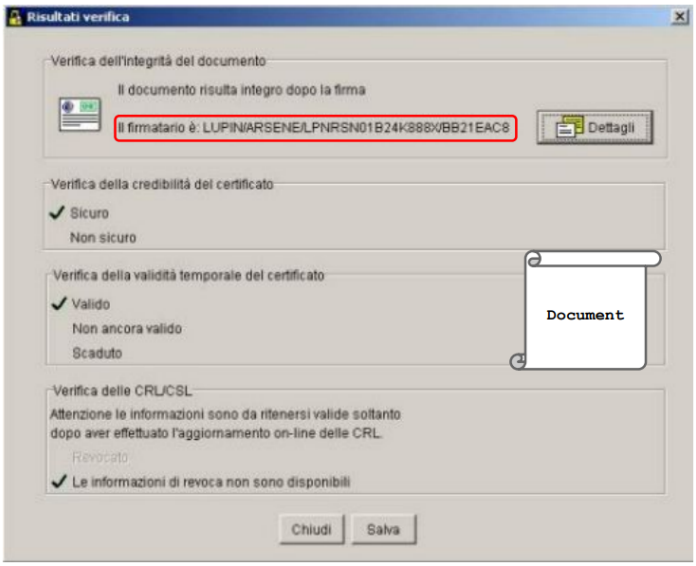
1. Does the signature validate the document?
   Hash verification as we have seen
2. Is the public key the one on the certificate?
3. Is the certificate the one of the subject?
   Problems with omonimous subjects, DN
4. Is the certificate validated by the CA?
   Validate the entire certification chain, up to the root
5. Is the root certificate trusted?
   Need to be already in possession of the root cert
6. Is the certificate in a CRL?
   How do we get to the CRL if we are not online?

Horror story 2:
In 2003, Firma&Cifra was the digital signature application by PostECom.
A vulnerability was found. In order to allow for offline verification, when presented with any certificate bundled with a signed document, it trusted the provided certificate, considered it authentic and added it to its trusted storage.
The exploit, called *Arséne Lupin signature*, worked like this:

1. Generate a fake root CA certificate with the same name as a real one (e.g., PostECom itself).
2. Use this to generate and sign a fake user certificate (in our example Arsène Lupin).
3. Use Arsène Lupin's certificate to sign a document (e.g. theft and burglary confessions).
4. Include the fake root cert to the PKCS#7 envelope (attached to the document).

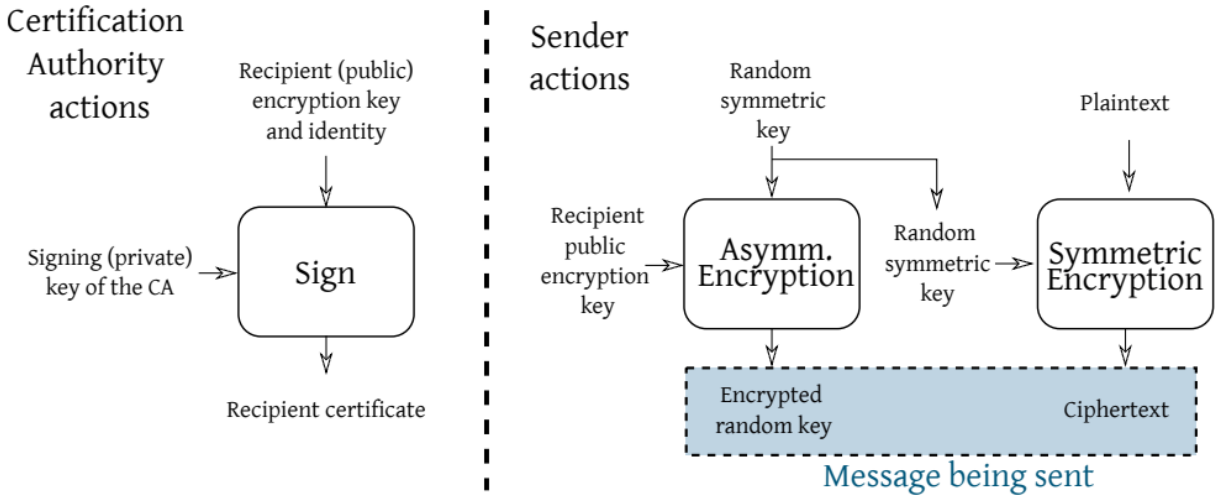






Take away point: which certificates reside in your (applications') trusted storage determine who you trust. Usually, the not secure part of authentication mechanisms is not the cryptographic one (if the right cryptographic algorithms are chosen) but other weaker paths of less resistance in the software.

Putting all together



Modern cryptography

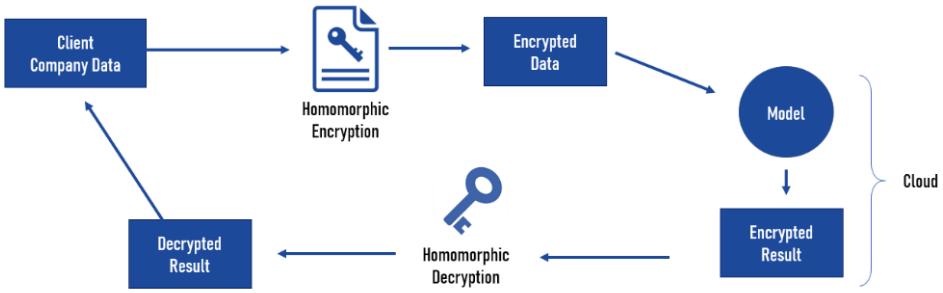
Three are the main research questions defining the directions in modern cryptography.

<u>What if we have a quantum computer?</u>

Some computationally hard problems are no longer hard, so we need to move away from cryptosystems based on hard problems for classic computers (like factoring or discrete log) and start using problems that are hard for quantum computers. Alternatives are available and being standardized (2022-04), public contents for algorithms to consider safe are being held.

What if we want to compute on encrypted data? (homomorphic encryption)

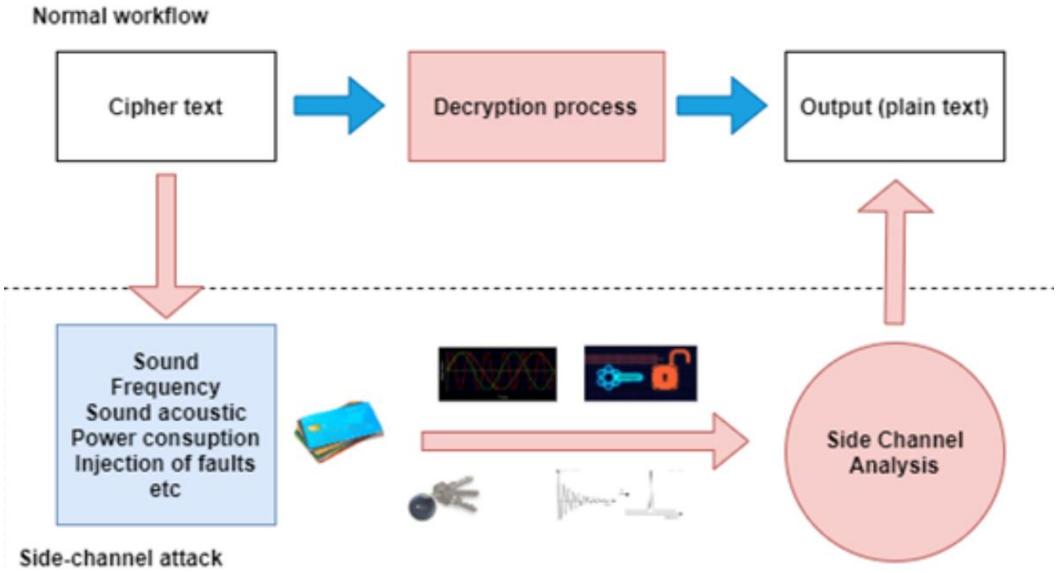
Perform computation of encrypted data is possible, for example to extract statistics from data without knowing the data itself, but it's moderately-to-horribly inefficient, as at the moment these computations allow only simple operations with high cost.



What if the attacker has physical access to the device computing the cipher (or some way of remotely measure it)

We add a new attack surface to our assessments, taking into account side channel information in the attacker model. Instead of solving the hard problem, an attacker can solve a secondary one, for example he can look at the time or power consumption to deduce if a certain key is right or wrong. Information leakage of this sort goes under the area of attacks named **side channel attacks**.

The new threat model includes passive attacks, active attacks, and side channel attacks.

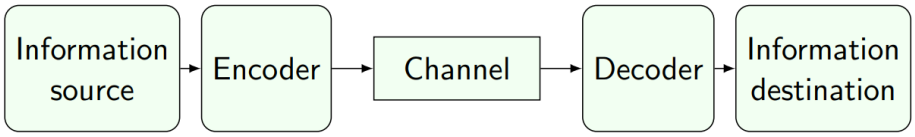


Information theory

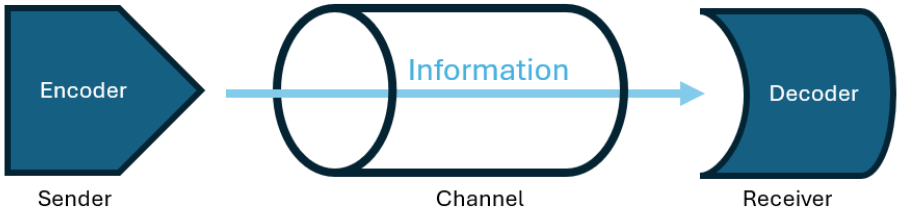
The objective of **Shannon's information theory** is to mathematically frame communication. We want a way to quantify information, measuring for example the amount of information exchanged between two entities.

Information theory is useful in the scope of cryptography to quantitatively frame "luck" and "guessing". For example, we are interested in determining how hard it is to guess a random string.

A *communication* takes place between two endpoints: a sender, made of an information source and an encoder, and a receiver, make of an information destination and a decoder.

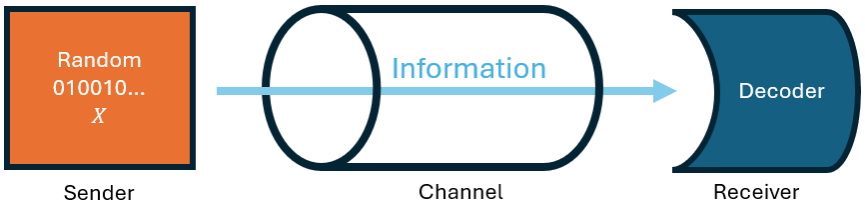


Information is carried by a channel in the form of a sequence of symbols of a finite alphabet. To send information, it must first be transformed into a sequence of bits.



As assumption, acquiring information means lowering the uncertainty about something the information is about.

The receiver gets information (a sequence of bits) only through the channel. The receiver will be uncertain on what the next symbol is, until the symbol arrives, thus, we can model the sender as a random variable (and the receiver as reading the output of this random variable).



Acquiring information is modeled as getting to know an outcome of a random variable X.

Then, the amount of information depends on the distribution of X. The more random is this variable X (the more possible outcomes there are), the more info the receiver can acquire.

The intuition of Shennon is that the closer is X to a uniform distribution, the higher the amount of information we can get get from knowing an outcome.

The uniform variable is the variable with equally probable outputs; hence it is the one providing the higher amount of info.

Encoding maps each outcome as a finite sequence of symbols, so more symbols should be needed when more information is sent. The reasoning is that the higher a source is random, the higher the information you can get from it, the more are the symbols needed.

We have found a relationship between the randomness of a source and the number of bits needed to encode the information. From it, we can go towards a measure of uncertainty: Entropy.

Entropy

Entropy is a non-negative measure of uncertainty. It must be additive, so that combining uncertainties should map to adding entropies.

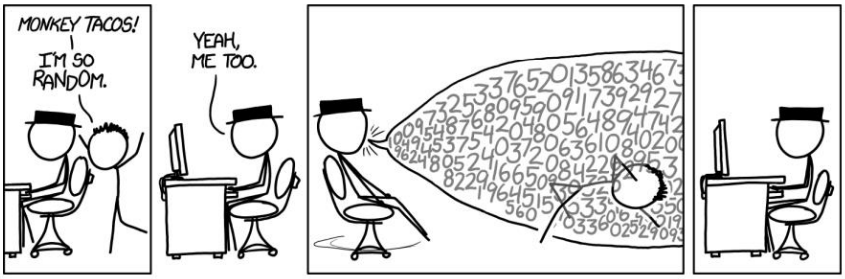
Let X be a discrete random variable with n outcomes in $\{x_0, \dots, x_{n-1}\}$ with $\Pr(X = x_i) = p_i$ for all $0 \leq i \leq n$. (we are taking the probability of each outcome from a source)
The entropy of X is:

$$H(X) = \sum_{i=0}^{n-1} -p_i \log_b(p_i)$$

Because $0 \leq p_i \leq 1$, the logarithm is negative and so we need to put a minus in order to obtain a non-negative entropy. The logarithm has the aim to make the measure additive, by summing all the logarithms of the probabilities.
The measurement unit of entropy depends on the base b of the logarithm: typical case for $b = 2$ is bits.

Entropy can be used as a metric to express the hardness of guessing a string. Higher the entropy, the more the attempts required to guess something.

In fact, the more unpredictable the source, the more information it can provide. Entropy enables to quantify which randomness is really random. For example, humans are not good random generators.



$\mathcal{X}$: Uniformly random 6 letters word

- $\mathcal{X}$ is a sequence of 6 unif. random letters ($6^{26} \approx 3.1 \cdot 10^7$)
  - $H(\mathcal{X}) \approx \sum_{i=0}^{3.1 \cdot 10^7} -\frac{1}{3.1 \cdot 10^7} \log_b(\frac{1}{3.1 \cdot 10^7}) \approx 28.2b$
- $\mathcal{X}$ is a uniform pick from 6-letters English words
  - $H(\mathcal{X}) \approx \sum_{i=0}^{6300} -\frac{1}{6300} \log_b(\frac{1}{6300}) \approx 12.6b$

A source with a higher number of outcomes has higher entropy. Source constrained in the number of outcomes have instead lower entropy.

Min-entropy

It is possible to have distributions with the same entropies, but with a practical mismatch.

Taking passwords for example, there are combinations that are more common than others and can be guessed more easily. Some outcomes are more predictable than others. To account for this fact, we use another measure, that is the min-entropy.

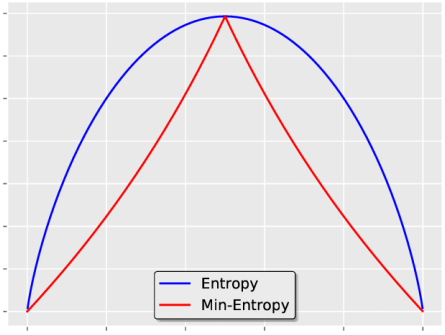
We define the **min-entropy** of X as:

$$H_\infty(X) = -\log\left(\max_i p_i\right)$$

Intuitively: it's the entropy of a random value with uniform distribution, where the probability of each outcome is $\max_i p_i$.

We are measuring how hard it is to guess the most probable outcome.

Guessing the most common outcome of X is at least as hard as guessing a $H_\infty(X)$ bit long bitstring.



A very biased r.v.

Consider $\mathcal{X}$: $\begin{cases} \mathcal{X} = 0^{128} & \text{with Pr } \frac{1}{2} \\ \mathcal{X} = a, a \in 1\{0,1\}^{127} & \text{with Pr } \frac{1}{2^{128}} \end{cases}$

Intuition and quantification

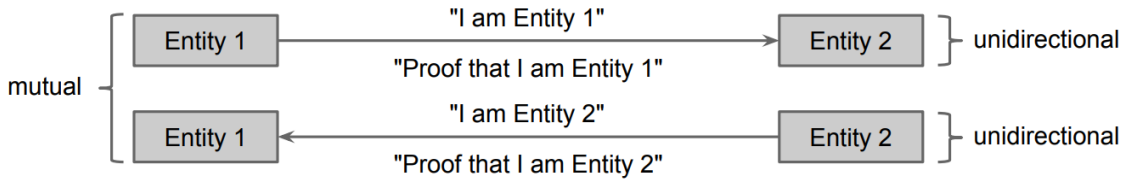
- Predicting an outcome shouldn't be too hard: just say 0^{128}
- $H(\mathcal{X}) = \frac{1}{2}(-\log_2(\frac{1}{2})) + 2^{127}\frac{1}{2^{128}}(-\log_2(\frac{1}{2^{128}})) = 64.5\text{b}$
- $H_\infty(\mathcal{X}) = -\log_2(\frac{1}{2}) = 1\text{b}$
- Min-entropy tells us that guessing the most common output is as hard as guessing a single bit string

3. Authentication

A process to verify the user's identity is needed, because through authentication we can check whether the user is allowed to perform some action or correlate that identity to past events, keeping a record of who did what.

Authentication is the process following identification. During **identification**, an entity declares an identifier. Later, during **authentication**, the entity provides a proof that verifies its identity.

Authentication can ben unidirectional, if just the user authenticates, of bidirectional (mutual), if also the system is required to provide proof to be the actual system to the user.



Unidirectional authentication is vulnerable to a lot of attacks: an attacker could for example impersonate the system and gain info about the user (social engineering, spoofing, …). Authentication should always be bidirectional, with both entities providing proof of their identity.

Authentication can happen between any entity: human to human, human to computer, computer to computer (in this last case usually through generated tokens).

Authentication is the foundation for the subsequent authorization phase.

There are three ways an entity can prove to be what it declares to be, these are the three **factors of authentication**. An entity can provide:

1. Something that the entity knows (to know)
   Example: password, PIN, secret handshake.
2. Something that the entity has (to have)
   Example: Door key, smart card, token.
3. Something that the entity is (to be)
   Example: Face, voice, fingerprints, biometrical factors.

With these factors we are authenticating respectively someone who knows a secret, is in possession of something, or is something. There is never a 100% guarantee that we are authenticating the actual person or entity, but having more factors give a good probability to be right.

Usually, the to be factor is more robust than the to have factor, that is more robust than the to know factor.

Factors of authentications must be combined in order to find a good tradeoff between security, usability and cost. **Multi-factor authentication** uses two or three factors.

Humans prefer to use the *to be* factor (recognize a person) more than the *to have* factor (own a identification document), and lastly the *to know* factor (ask a secret to a person).

On the contrary, machines predilect the *to know* factor over the *to have* factor and leave as last the *to be* factor. In fact, machines cannot, for example, write a password somewhere, and keeping a secret is for them easier. The difference between machines and humans, when coming to choosing what factors to use for authentication, lies in usability considerations.

The “to know” factor

With the “to know” factor, an entity must know some secret in order to prove its identity and be authenticated. The entity needs to provide the secret to a second entity, that in some way know that the secret is the right one (or can just prove to know the secret without explicitly telling it, e.g. Diffie-Hellman).

A password does not authenticate and entity, it just authenticates someone knowing the secret. Everyone obtaining the secret can impersonate the owner of that secret, and there can be no proof the secret (or password) is stolen. Also, if a secret is stolen, the owner may not know it.

The advantages of this factor are:

- Low cost
- Ease of deployment
- Low technical barrier

But there are disadvantages:

- Secrets can be stolen/snooped, especially if stored or written somewhere
- Secrets can be guessed; some are more likely than others, as there are some more used by everyone or related to personal info.
- Secrets can be cracked with enumeration or dictionary attacks.

Top 25 most common passwords by year according to SplashData

Rank	2011[6]	2012[7]	2013[8]	2014[9]	2015[10]	2016[5]	2017[11]	2018[12]	2019[13]
1	password	password	123456	123456	123456	123456	123456	123456	123456
2	123456	123456	password	password	password	password	password	password	123456789
3	12345678	12345678	12345678	12345	12345678	12345	12345678	123456789	qwerty
4	qwerty	abc123	qwerty	12345678	qwerty	12345678	qwerty	12345678	password
5	abc123	qwerty	abc123	qwerty	12345	football	12345	12345	1234567
6	monkey	monkey	123456789	123456789	123456789	qwerty	123456789	111111	12345678
7	1234567	letmein	111111	1234	football	1234567890	letmein	1234567	12345
8	letmein	dragon	1234567	baseball	1234	1234567	1234567	sunshine	iloveyou
9	trustno1	111111	iloveyou	dragon	1234567	princess	football	qwerty	111111
10	dragon	baseball	adobe123[a]	football	baseball	1234	iloveyou	iloveyou	123123
11	baseball	iloveyou	123123	1234567	welcome	login	admin	princess	abc123
12	111111	trustno1	admin	monkey	1234567890	welcome	welcome	admin	qwerty123
13	iloveyou	1234567	1234567890	letmein	abc123	solo	monkey	welcome	1q2w3e4r
14	master	sunshine	letmein	abc123	111111	abc123	login	666666	admin
15	sunshine	master	photoshop[a]	111111	1qaz2wsx	admin	abc123	abc123	qwertyuiop

Countermeasures do exist, but it is usually not possible to deploy a lot or all of them, because they are a cost in various terms (included usability). Putting too many countermeasures in place could also be counter-productive, always for usability and cost reasons.

The correct thing to do it to estimate the most likely attack scenarios in order to choose the countermeasures that is worth asking the user to adhere to. We need to find a balance between the assets to protect and the possible threat agents.

Countermeasures include:

- Enforce passwords that change/expire frequently
- Enforce passwords that are long and have a rich character set

- Enforce passwords that are not related to the user.
- Fix a limit to the number of attempts
- Set alerts after each login is received

These countermeasures are a cost because humans are not machines, so they are inherently unable to keep secrets and have difficulties in remembering complex passwords. We can't pick unlimited countermeasures, but we need to choose the right combination of them depending on the context.

To choose countermeasures we start from a risk and threat assessment, and based on it select the best countermeasures that are worth to deploy in terms of direct and indirect costs and risks.

The three main attacks against the "to know" factor are snooping, cracking and guessing. Snooping is the most common as it is the easiest attack to perform.

Against *snooping*, changing the password frequently is helpful because the attacker would find himself often at the starting point.

Against *cracking*, enforcing more complex passwords means requiring more resources to crack it. Also changing the password can help, because when a password is changed, the cracking process must start from the beginning. Usually, a cracking attack is not related to a specific targeted user, but different combinations are tried until a victim is found.

Against *guessing*, complexity of the password and frequent change can help, but the most important thing is that the password is hard to guess and thus is not related to the user.

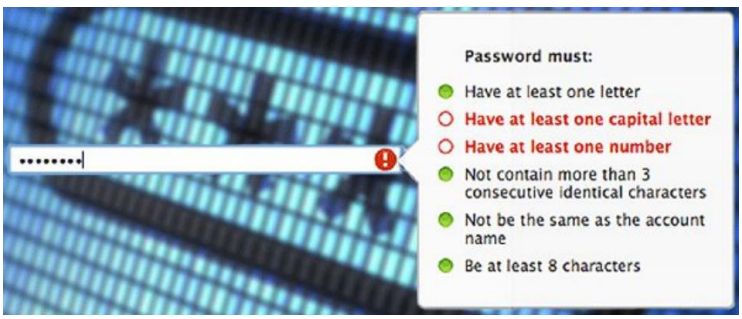
Countermeasure guideline: important, may help, unimportant

Against **snooping**	Against **cracking**	Against **guessing**
complexity	*complexity*	*complexity*
change	*change*	*change*
being related to users	being related to users	*not being related to users*

Each countermeasure is a cost for the user, because complexity means that a password is more difficult to remember, changing It frequently means involving the user often, not being related means that it would be a little more difficult to remember.

Changing the password is the countermeasure that is generally more effective, because it is an obstacle against more kinds of attacks. After changing, going in order of importance, there is complexity and then not being related to the user.

To enforce these countermeasures, humans must be "educated" to put them in place effectively. Ways to do so are enforcing strong passwords in the processes, enforcing password expiration/change policies, and showing password meters to users to balance usability.



With password meters, the software puts constraints and suggestions in order to educate the users to create strong passwords. Gamification is used, so the password is evaluated with a score (e.g. red to green bar). High score given to passwords pushes users to choose better combinations.

Current password meters are based on criteria that do not resemble the status of things. A password with a lot of special character is easier to crack than a password with no special character but way longer.

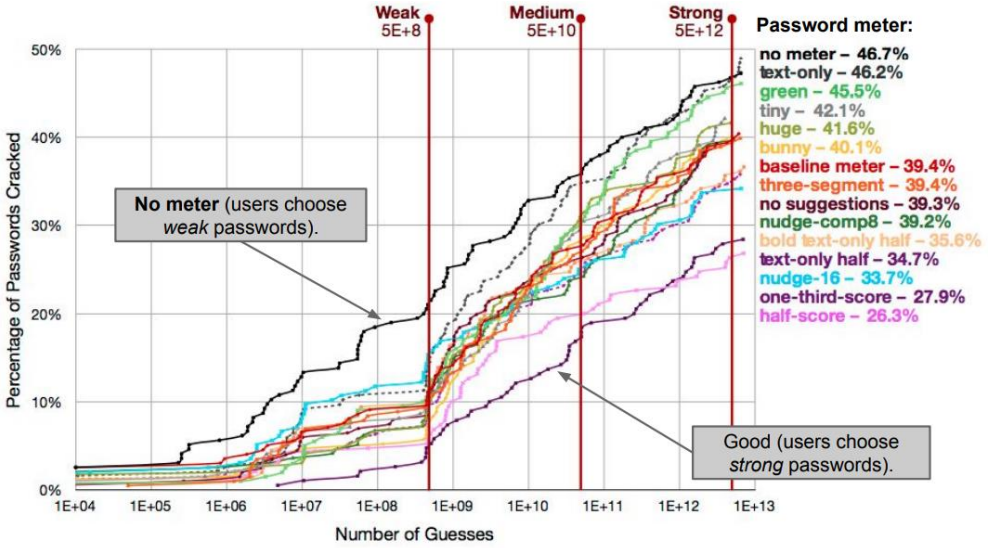
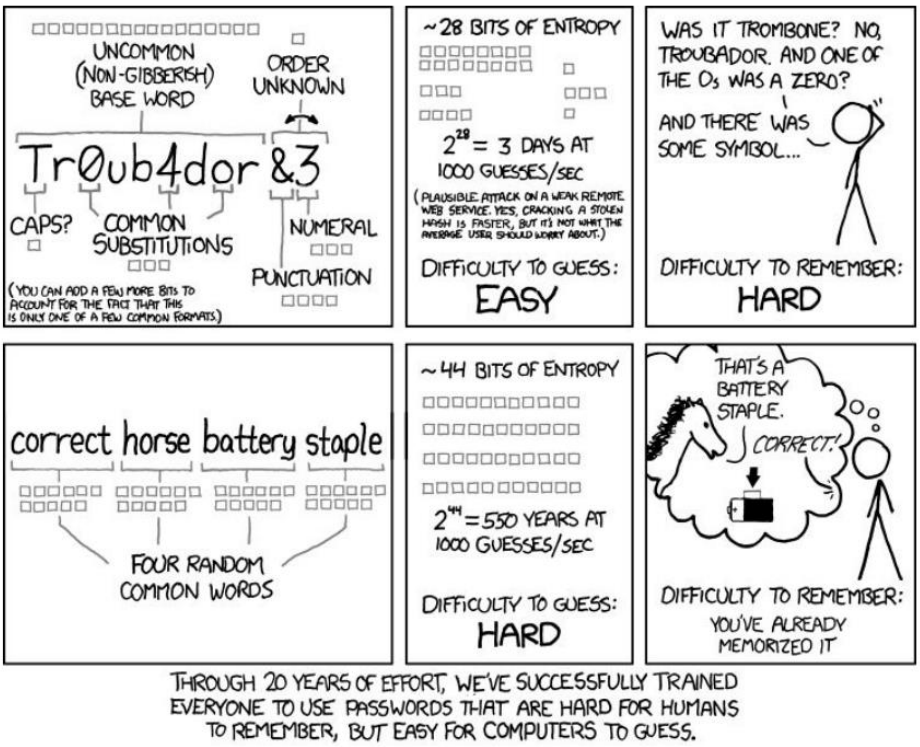


Figure 3: This graph contrasts the percentage of passwords that were cracked in each condition. The x-axis, which is logarithmically scaled, indicates the number of guesses made by an adversary, as described in Section 2.4. The y-axis indicates the percentage of passwords in that condition cracked by that particular guess number.

There are different ways in which a password may be complex.



A password must be complex but in a way in which usability is not compromised too much. A password composed of more uncorrelated words one after another has more entropy (is harder to guess) than a password shorter but with a richer character set.

Secure password exchange

Authentication is about sharing a secret, so at some point the secret must be shared and stored.

We need to minimize the risk that secrets get stolen during the sharing phase. Diffie-Hellman can for example be used to share a secret.

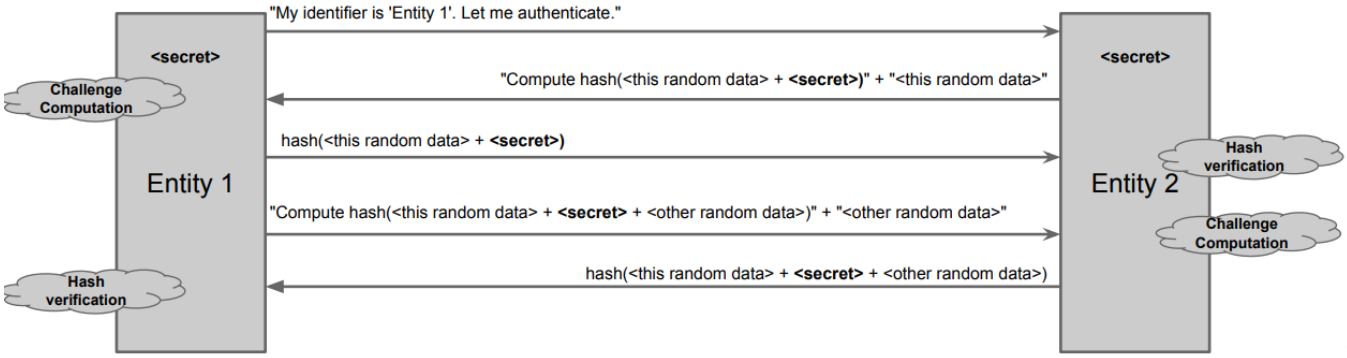
First, we need to use mutual authentication every time it is possible, to make sure an attacker is not impersonating the system we are authenticating to.

Then, we can use a challenge and response scheme to avoid sharing the secret in order to prove to know it. The method has the aim to verify if two entities know a secret without sharing that secret.

The protocol works like this:

- Entity 1 identifies and asks to authenticate
- Entity 2 sends some random data and asks to compute the hash of the secret concatenated with these random data
- Entity 1 sends the result of the required computation and entity 2 checks the result. If correct, then entity 1 is authenticated to entity 2.
- Now entity 1 sends some different random data and asks to compute the hash of the secret concatenated with the random data of before and the new random data.
- Entity 2 sends the result of the required computation and entity 1 checks the result. If correct, then entity 2 is authenticated to entity 1.

Being hash functions not revertible, it is not possible to retrieve a secret from its hash, but if we know the secret then we know what its hash is and can compare it with other hash to see if they come from the same secret.



Dictionary attack: try different commonly used passwords, hashed with different hash algorithms, and see if there is some correspondence.

Apart from using mutual authentication and a challenge and response scheme, it is important to use random data to avoid replay attacks. When carrying out a replay attack the attacker spoofs the data sent (for example the hash of a secret, that is deterministic) and uses it to impersonate one of the entities. If random data is always different, the sent hash from the secret concatenated with random data is always different.

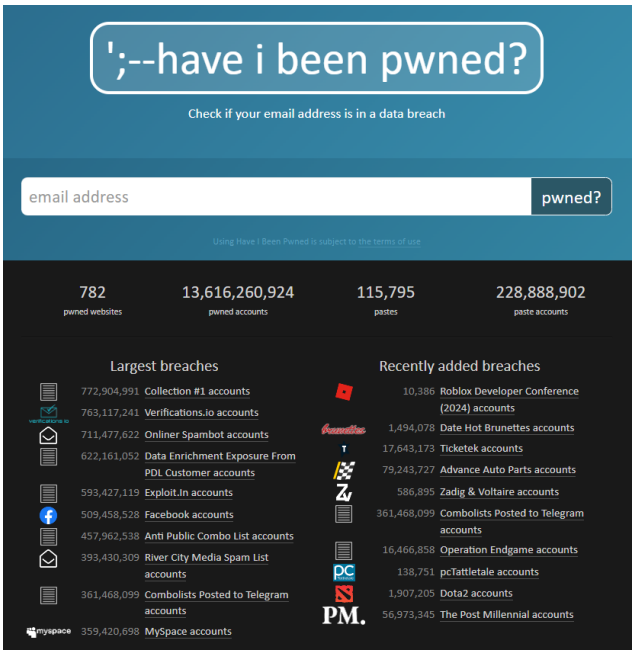
Another attack if no random data is used can be a DoS (Denial of Service) using spoofed data to generate a lot of requests.

Secure password storage

OS stores a file with usernames and passwords. An attacker could try to compromise the confidentiality and integrity of this password file.

There are ways to minimize the risk that secrets get stolen in this way.

The first thing to do is to use cryptographic protection. One can encrypt the passwords or can store their hash instead of the passwords. However, a lot of people use the same passwords, so we would have a lot of passwords with the same hash and information leakage. Hash alone is not sufficient, and a solution is to add **salting**: a sequence of bits different from user to user is added to the password before hashing, in order to obtain different hash for the same password of different users. This way, we also obtain resistance against dictionary attacks.



Put in place effective access control policies (system of privileges to read or write). The attacker needs to have high privileges to operate on a machine, even after having compromised it in some way.

Lastly, it's important to never disclose secrets in password-recovery schemes. In fact, the weakest element of the system may be the password recovery mechanism. Never send password in clear or generated password (can be figured how it is generated), a link to reset the password is better (attacker could access the account but not figure out the secret).

We need to take into account that information is often held in intermediate storage locations when operations are performed. We have a caching related problem: there is a limited amount of time in which the password is in clear in the cache and this time lapse is often vulnerable and must be minimized.

The “to have” factor

With the “to have” factor, we are authenticating who has the factor in his hands (not necessarily the entity). The user must prove that it possesses something, and if that something is stolen, he can report it and make it ineffective (but may require some time).

The advantages of the “to have” factor are:

- Human factor, meaning that a human is less likely to hand out a key than for example to share or disclose a secret.
- Relatively low cost.
- Good level of security.

The disadvantages are that these systems:

- Systems are hard to deploy
- Devices can be lost or stolen

These disadvantages make clear why this factor is often used only as a second factor, so it is used in combination with another factor. We do not want, for example, the attacker to be able to steal the factor and authenticate itself just because he has it in his hands.

There are no effective countermeasures, apart from deciding not to use the “to have” factor as the only authentication factor.

One-time password generators consist in devices that, after a button is pressed, can generate a secret key to insert somewhere, and have a counter synchronized with the host. The client uses the counter and the key to call MAC_COMPUTE(counter, key) and the host calls MAC_VERIFY(counter, key). The key is stored in the device but changes every 30-60 seconds, so an attacker needs to put in place the attack in real time, because the key is time dependent. The attacker needs to retrieve both the key and the counter. The counter is needed because these devices are isolated and not connected to the internet.

Application examples include online banking and admin console.

Smart cards are composed of a CPU and a non-volatile RAM with a private key (the secret is embedded in the card). The smart card authenticates itself to host via a challenge-response protocol, using the private key to sign the challenge.

The private key should not leave the device, and the card should be tamper proof to some extent. These devices are made in a way that if an attacker tries to retrieve the key from them then they will break.

These cards can be cloned, so it is important to pair them with a PIN, so the attacker would also need to steal a secret to use them.

Another attack could be through a side channel, by studying the power usage to gain information on the private key.

Application examples of smart cards are the credit cards.

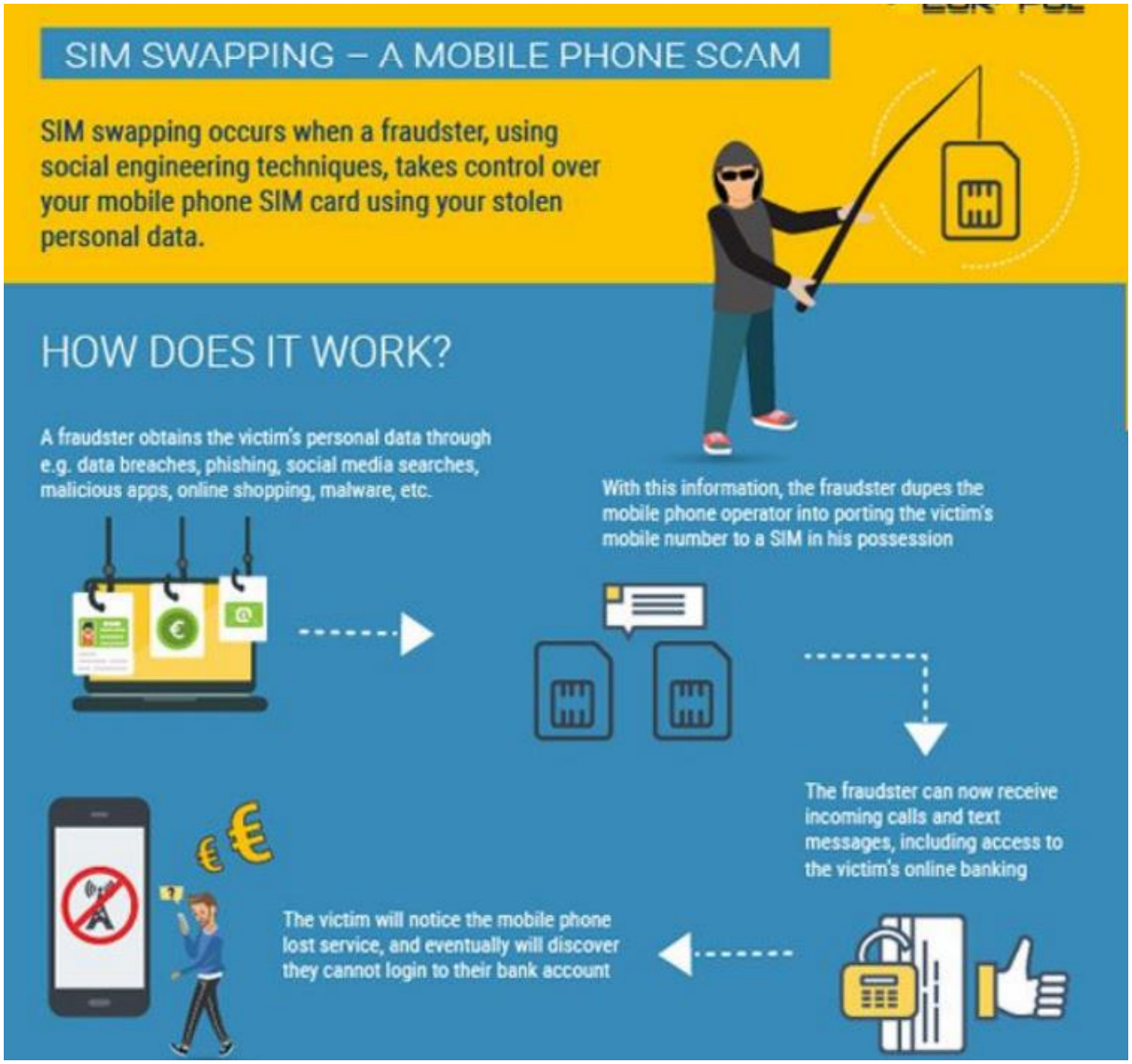
A cheaper alternative are the **static OTP** (one time pad) **lists**, consisting in cards with a list of secret keys on them, known by both the client and the host. The host chooses a challenge, so a random number (e.g., "second digit of the 14th cell"), and the client transmits the response (hopefully, over an encrypted channel).

The host should not keep the list in clear (for example, by using hashing). And the client must be careful to phishing and not to put the data in the wrong website or place. These lists are static and their effectiveness relies on their secrecy.

The current standard it **TOTP** (Time-based one-time password), consisting of applications for smartphones or other devices implementing the one-time pad code. TOTP is a computer algorithm that generates a one-time password (OTP) using the current time as a source of uniqueness. The generated codes are time dependent, so attackers need to put in place real time attacks.

The first advantage is in the fact that deployment is not needed, but the host can ask the user to install the application. These are not closed embedded systems, but they work on general-purpose software and hardware platforms, making them more vulnerable than other embedded devices, because a vulnerability of the whole smartphone can affect the application. However, they are a good trade-off, easy to deploy and time sensitive, and used because smartphone security is still relatively high and for usability reasons.

We need to be careful at when smartphones are substituted, and the password generation application needs to be transferred on the new device. SIM cards are used to identify, but **sim-swapping** targeted attacks can happen.



The “to be” factor

With the “to be” factor, an entity authenticates by showing a biometric factor or some specific characteristics as proof of its identity.

The advantages od the “to be” factor include:

- High level of security, as it cannot be forgotten or lost.
- Requires no extra hardware to carry around.

But the disadvantages are:

- Systems are hard to deploy, costly and need data to be trained.
- Systems use probabilistic matching: there is a threshold in accuracy to be set correctly (usability vs errors).
- Invasive measurement.
- Biometric factors can be cloned, for example fingerprints can be collected from where their owner touches.
- Bio-characteristics change, and changing the associated fingerprints Is not as easy as changing a password, the system would need to be retrained.

- Privacy sensitivity, as we need to store sensitive information, that must be protected
- Users with disabilities may have difficulties.
- The safety of people is in danger, because motivated criminals could want to obtain the biometric factor.

Countermeasures are not effective against all these disadvantages and include re-measuring the traits often with a usability problem to secure the stored data and to implement alternatives (but adding another attack surface).

Technologies available include, for example: fingerprints, face geometry, hand geometry (palm print), retina scan, iris scan, voice analysis, DNA, typing dynamics, peculiar movements, etc.

Voice can be recorded and easily cloned, or it can change easily according to situation, sickness or aging.

Using **fingerprints** require to collect a reference sample of the user's fingerprint at a fingerprint reader. More reference sample from different finger and different positions are needed to obtain higher accuracy.

So, features of the fingerprints are derived from the sample, and these values are used to identify the fingerprint. Features are stored in a secure database. When the user logs on, a new reading of the fingerprint is taken; features are compared against (similarity) the reference features. User is accepted if match is above a predefined threshold. Setting the threshold in the right way is extremely important, as doing it wrong can cause both false positives and false negatives.

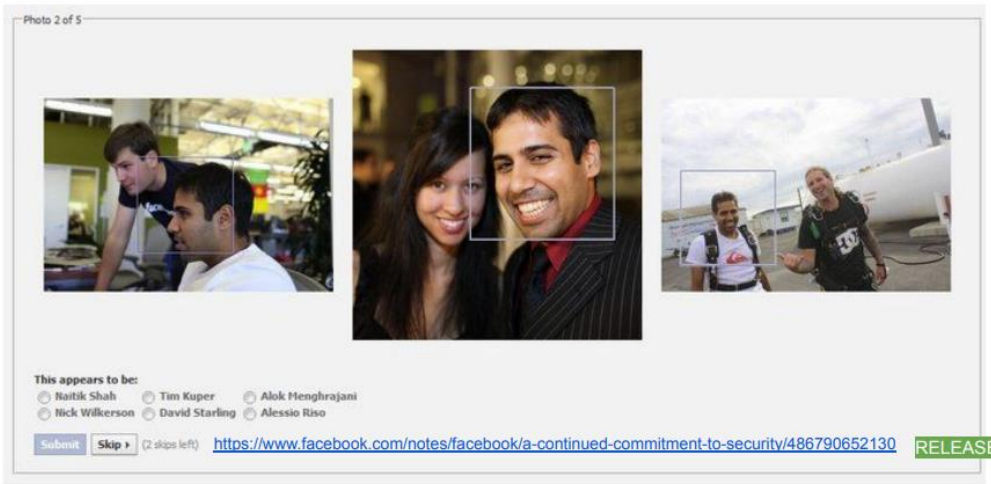
An example of beaten fingerprint authentication factors are touch ID by Apple that was cracked after 1 day of release in 2013 just by taking the fingerprint from the screen and printing it on paper with a laser printer.

Manufacturer	Model	Technology	Date	Difficulty
Identix	TS-520	Optical	Nov. 1990	First attempt
Fingermatrix	Chekone	Optical	Mar. 1994	Second attempt
Dermalog	DemalogKey	Optical	Feb.1996	First attempt
STMicroelectronics	TouchChip	Solid state	Mar. 1999	First attempt
Veridicon	FPS110	Solid state	Sept.1999	First attempt
Identicator	DFR200	Optical	Oct. 1999	First attempt

There are a lot of exploits to fingerprints, face and eye recognition in which the biometrics are replicated and recognized as correct match by the machines.

Facebook tried to use the so-called “social” factor, as an additional factor of authentication other than basic ones. The assumption behind was that the friends were an information that only the owner of a profile would know, but this assumption is false, as friends may have the same friends. Moreover, people leave enough information on the internet to be recognized, especially starting from names.

Alice must prove that she *knows someone.*



Papers

- H. Kim, J. Tang, and R. Anderson. *Social authentication: harder than it looks.* In Proceedings of the 2012 Financial Cryptography and Data Security conference.

CRACKED J. Polakis, M. Lancini, G. Kontaxis, F. Maggi, S. Ioannidis, A. Keromytis, S. Zanero, *All Your Face Are Belong to Us: Breaking Facebook's Social Authentication.* In Proceedings of 2012 Annual Computer Security Applications Conference.

Single Sign On

Managing, protecting and remembering multiple passwords is complex. Users tend to re-use the same passwords over multiple websites, password policies are replicated (cost).

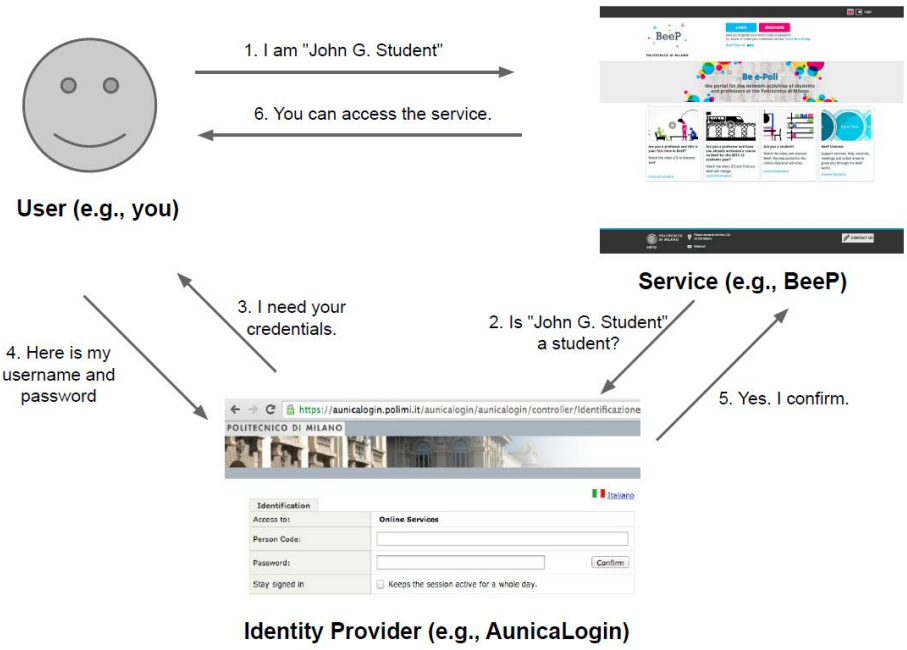
Single Sign-On (**SSO**) is an authentication process that permits a user to access multiple applications with one set of login credentials. This eliminates the need for multiple usernames and passwords, reducing the risk of password fatigue and improving security.

SSO enhances user experience by streamlining the login process and simplifies management for administrators by centralizing authentication. Common in both enterprise and consumer environments, SSO integrates with various systems, such as cloud services and on-premises applications, providing seamless access while maintaining stringent security protocols.

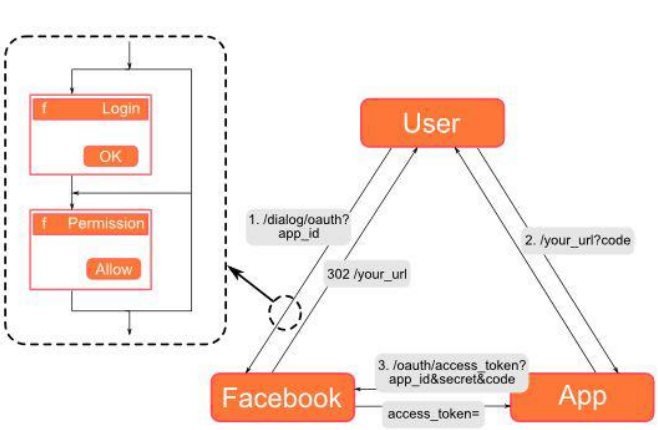
With SSO, users have just one identity, use one or two authentication factors to authenticate to a single selected trusted host (e.g. google).

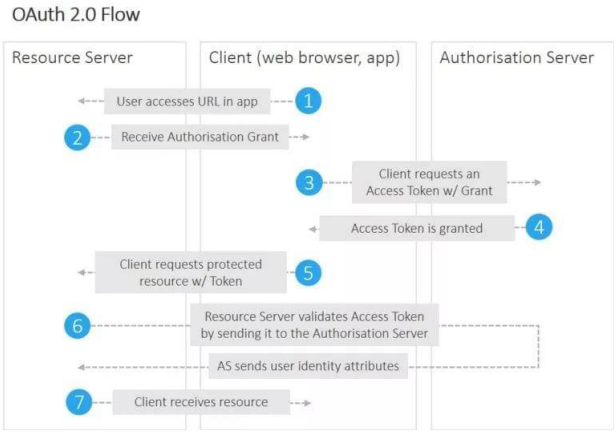
The user selects a trusted host providing SSO and authenticates on it, then other hosts ask the trusted host if that user is authenticated. The user authenticates to a third-party actor that authenticates him to other parties, guaranteeing the credentials.

An example is Shibboleth (AunicaLogin), that is the mechanism used by Politecnico di Milano to guarantee authentication.



OAuth2 Flow (Facebook) is a service that has the same aim but works by guaranteeing authorization, not authentication. User connects to an external app, asks for a resource and the server grants a token to access that particular resource.





Of course, there are challenges associated with Single Sign On. SSO is a single point of trust (the trusted server), and if that point is compromised, then all sites are compromised.

Moreover, if the trusted host is down, an alternative way to access the other sites is needed, so the attack surface is still wide. The alternative sign-in is often weaker.

Especially for SSO, password reset schemes must be bulletproof, and present no or limited vulnerabilities. The email used for password reset is a trusted element of the scheme.

Another problem is that SSO is difficult and complex to implement; libraries exist but they can be bugged and add more vulnerabilities.

Password managers

Password managers are an alternative solution to SSO to the same problem SSO try to solve.

A trusted host is elected, and this host has the aim to remember and manage the passwords for the user. The user authenticates (sign on) on the trusted host, usually with a master password, and has access to all his passwords.

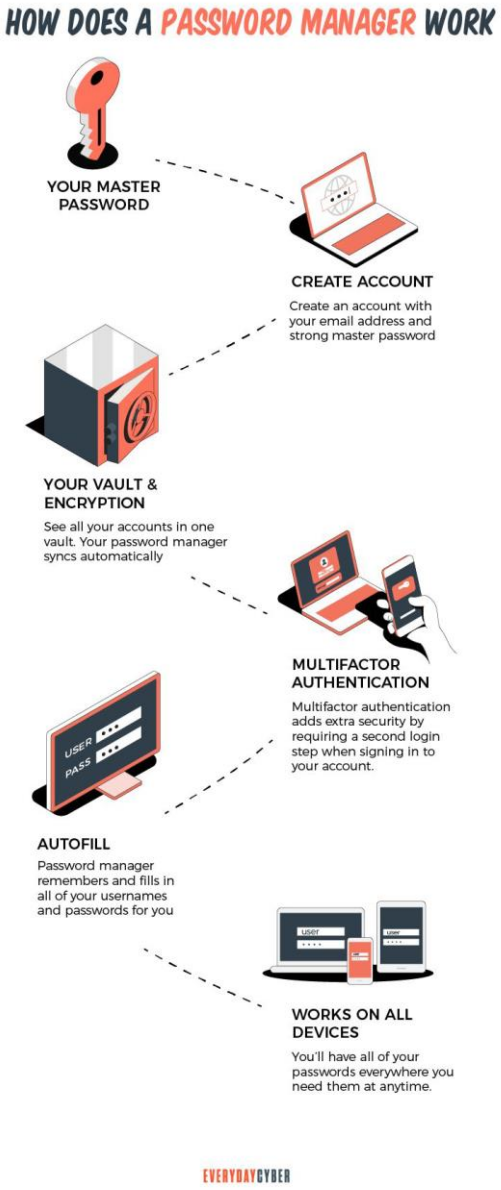
Of course, the password manager is the trusted element of the system and must not be compromised.

The advantages of password manager are:

- No need to remember all passwords, but just the master password (that must be a strong password)
- The password manager allows to generate and remember robust and strong passwords
- Good usability: auto fill, auto synch, multi devices.

The disadvantages:

- We are trusting a single sign-on
- There is a single point of failure
- If the user loses his master password, he loses access to all his accounts
- The attack surface is larger, because password managers are software, and every software must be very well implemented in order not to expose vulnerabilities

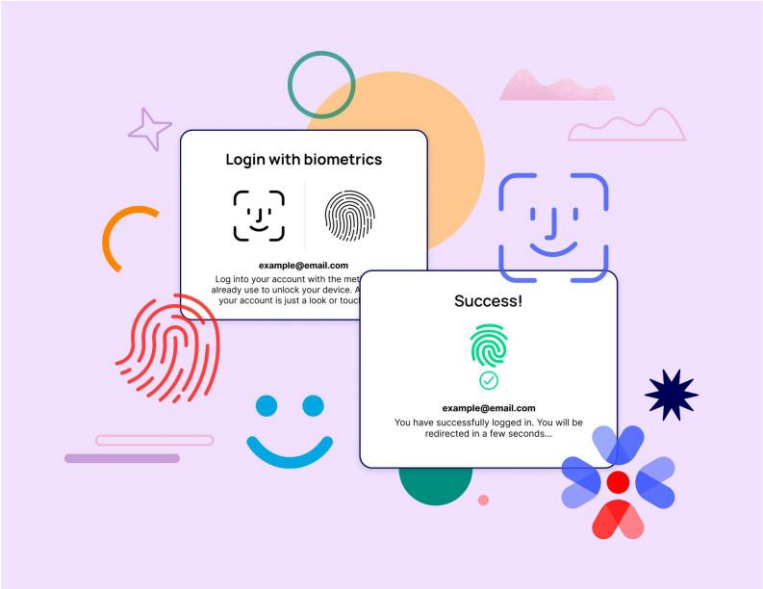


Passkeys

Password-less authentication with **Passkeys** involves using alternative methods to authenticate users without requiring traditional passwords. Passkeys are often based on public key cryptography and challenge and response protocols. Here's how it works:

1. During setup, a public-private key pair is created on the user's device. The public key is stored on the server, while the private key remains on the device.
2. When logging in, the user is prompted to confirm their identity using a biometric method (e.g., fingerprint or facial recognition) or a device PIN. The device then uses the private key to sign an authentication request, initiating a challenge and response scheme.
3. The server uses the stored public key to verify the signed request, confirming the user's identity.

This method has the aim to enhance security by eliminating passwords, reducing the risk of phishing and password breaches.



Conclusions

Identification, authentication and authorization are three distinct, yet inter-dependent, concepts.

There are three types of authentication factors, which should be used in combination.

Passwords are increasingly showing their limits.

New authentication schemes are promising but should be used with care.

4. Access control

Access control is the procedure that regulates who or what can view or use resources or perform some actions in a computing environment or system. It ensures that only authorized users can access specific data or systems, preventing unauthorized access and potential breaches. Access control mechanisms include authentication (verifying user identity) and **authorization** (granting permissions based on user roles).

At the base of access control, there is a binary decision: access to a resource or part of a system is either allowed or denied.

The problem is complex: the scale at which the problem extends requires effort. We usually cannot explicitly list all the answers and need to condense them in rules, from which (or from their combination) vulnerabilities can be exposed.

The job of access control is to answer the following questions:

- How do we design the access rules?
- How do we express the access rules in practice?
- How do we appropriately apply them?

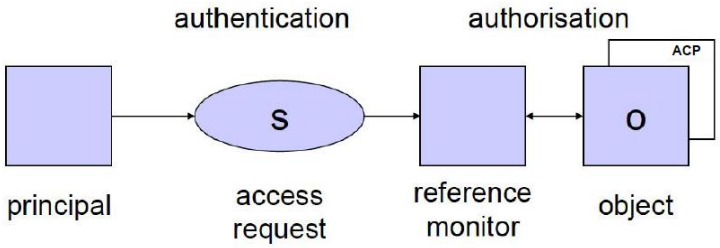
The **reference monitor** (RM) is implemented in all modern machines kernel and has the task to enforce access control policies. It is the trusted element of the system and has some requirements.

It must be tamper proof, so if an attacker tries to tamper with the system, it is evident, and if compromised should stop working.

It must not be possible to bypass the reference monitor in an way.

The reference monitor must be small enough to be verified and tested. The more complex the reference monitor is, the more difficult it is to analyze it and check its integrity, so it is better to have a small RM.

The access control mechanism has the following steps:



The user sends an access request (generally related to an object) to the reference monitor, which checks if the access to the object can be given to the user.

The RM manages both the authentication and authorization part.

Authentication: reference monitor verifies the identity of the principal (user) making the request:
- User enters username and password.
- If the values entered are correct, the user is “authenticated”.
  - If authentication is successful, then a process is created, with the rights of the user. We could say: “The machine now runs on behalf of the user”.
- This might be intuitive, but it is imprecise, in fact:
  - Log on creates a process that runs with access rights assigned to the user.
  - Typically, the process runs under the user identity of the user who has logged on.

Authorization: reference monitor decides whether access is granted or denied.
- Reference monitor has to find and evaluate the security policy relevant for the given request.

The problem is relatively “easy” in centralized systems but in distributed systems, we need to define how to find all relevant policies and how to make decisions if policies may be missing.

Requests to the reference monitor usually do not come directly from a user or a user identity, but from a process. The active entity making a request within the system is called the subject.

Concepts to distinguish between:
- **User**: person.
- **Principal (User identity)**: name used in the system, possibly associated with a user; Active entity.
- **Subject**: **Process** running under a given user identity.
- **Object**: Passive entity – file or resource.
- **Access operations**: Vary from basic memory access (read, write) to method calls in object-oriented systems.

There are different access control models, that can be roughly divided in three categories, based on who assigns the privileges.

- Discretionary Access Control (DAC), in which it is the owner of a resource to arbitrarily determine the access policies.
- Mandatory Access Control (MAC), in which the system administrator assigns privileges on resources.
- Role-Based Access Control (RBAC), that is a hybrid between DAC and MAC often implemented in companies where each role has different privileges.

The difference between DAC and MAC is in who assigns the privileges, while RBAC abstracts the roles from the identities of users.

Discretionary Access Control (DAC)

With the **Discretionary Access Control** (**DAC**) model, it is the resource owner to discretionarily decide its access privileges. So, who creates a file then decides who can access it and how.

All off-the-shelf operating systems implement DAC: Windows, Linux and other UNIX systems, Mac OS X, but also single applications or social networks are mostly DAC.

DAC models the following 3 entities:

- **Subjects**: users owning the resource or asking for it; who can exercise privileges (or rights).
- **Objects**: the resources, the files, on which the privileges are exercised.
- **Actions**: what can be done with the resources and can be exercised.

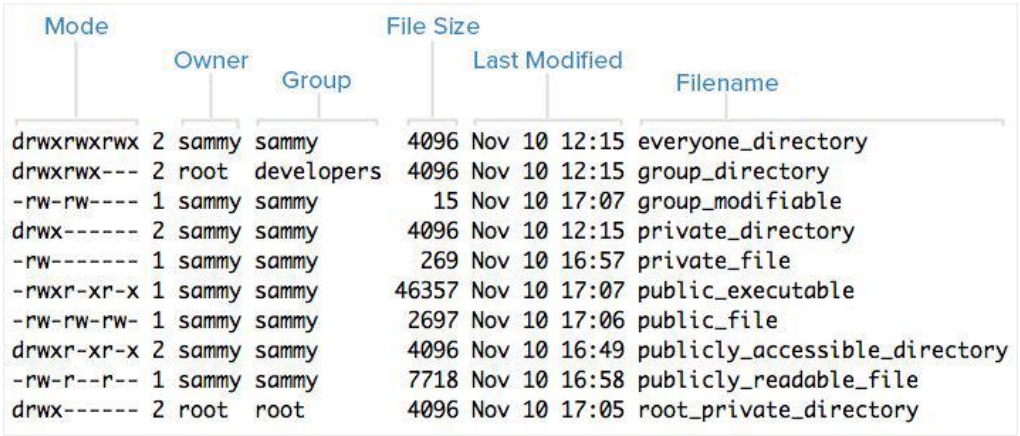
UNIX
- **Subjects**: users, groups
- **Objects**: files (everything, really)
- **Actions**: read, write, execute

Windows (not the 95/98/ME branches)
- **Subjects:** with *roles* instead of *groups*, multiple ownership of users and roles over files
- **Objects**: files and "other" resources
- **Actions**: delete, change permissions, change ownership.

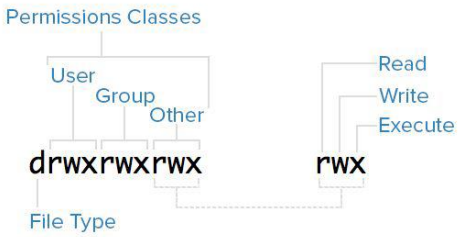
UNIX works by keeping a list of resources in each line of which it is specified the permission, the subject and the object.

$$permissions(r, w, x) + Subjects(user, group) + Objects\ (files, dirs, \dots)$$

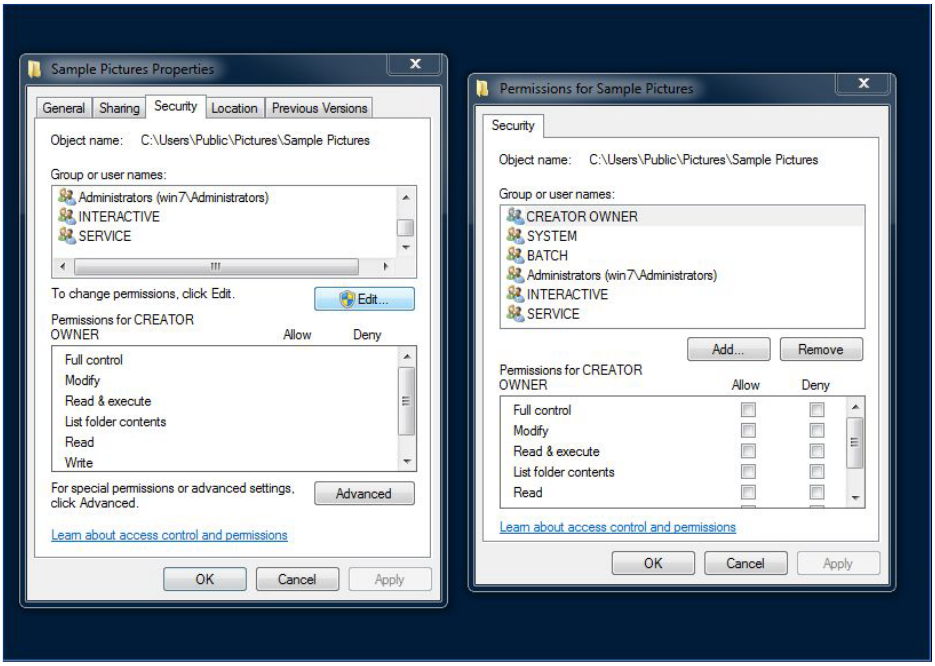
```
drwxr-xr-x    3 phretor  staff      102 Nov 19  2012 .idapro
drwxrwxr-x   14 phretor  staff      476 Aug 27  2011 .ipython
-rw-rw-r--    1 phretor  staff        1 Jan 30  2010 .irb_history
lrwxr-xr-x    1 phretor  staff       30 Jul  3  2011 .irssi -> /Users/phretor/dotfiles/.irssi
-rw-r--r--    1 phretor  staff      140 Dec  1  2012 .jackdrc
drwxr-xr-x    3 phretor  staff      102 Feb 21  2008 .jmf
-rwxrwxrwx@   1 phretor  staff       41 Dec  9  2012 .khlbshcrc
-rw-------    1 root     staff       51 Oct 27  2008 .lesshst
drwxr-xr-x    4 phretor  staff      136 Jan  4  2008 .lftp
drwx------    4 phretor  staff      136 Jul 22  2009 .links
drwxr-xr-x    3 phretor  staff      102 Apr  1  2013 .m2
lrwxr-xr-x    1 phretor  staff       32 Jul  3  2011 .mailcap -> /Users/phretor/dotfiles/.mailcap
drwxr-xr-x    3 phretor  staff      102 Jan  7  2008 .mldonkey
drwxr-xr-x    4 phretor  staff      136 Jan 25 16:29 .mono
lrwxr-xr-x    1 phretor  staff       31 Jul  3  2011 .mutt.d -> /Users/phretor/dotfiles/.mutt.d
lrwxr-xr-x    1 phretor  staff       36 Jul  3  2011 .muttprintrc -> /Users/phretor/dotfiles/.muttprintrc
lrwxr-xr-x    1 phretor  staff       31 Jul  3  2011 .muttrc -> /Users/phretor/dotfiles/.muttrc
drwxr-xr-x   11 phretor  staff      374 Jan 31  2008 .ncftp
drwxr-xr-x    8 phretor  staff      272 Dec  7 19:59 .neocomplcache
drwxr-xr-x    8 phretor  staff      272 Oct 21  2012 .neocon
drwxr-xr-x   11 phretor  staff      374 Feb  9  2013 .npm
lrwxr-xr-x    1 phretor  staff       38 Jul  3  2011 .offlineimaprc -> /Users/phretor/dotfiles/.offlineimaprc
drwxr-xr-x   15 phretor  staff      510 Feb  4 22:23 .oh-my-zsh
drwxr-xr-x    6 phretor  staff      204 Apr 20  2013 .parentseye
drwxrwxr-x    3 phretor  staff      102 Dec 24  2010 .pip
drwx------    6 phretor  staff      204 Apr  5  2008 .psi
-rw-------    1 phretor  staff       90 Mar 17  2010 .psql_history
```



Permission triads in UNIX are implemented with strings of 10 char containing information about access to some resources.



Windows works similarly.



A **protection state** is a triple (S, O, A) where:

- A is a matrix (*Access control matrix*) with S rows and O columns
- An entry $A[s, o]$ defines the privileges of subject s over object o

	file1	**file2**	**directoryX**
Alice	Read	Read, Write, Own	
Bob	Read, Write, Own	Read	Read, Write, Own
Charlie	Read, Write		Read

To update the access control matrix, transitions should respect the **HRU model** (Harrison-Ruzzo-Ullman), that is a formal model used to describe and analyze the protection mechanisms in computer systems and extends the access control matrix model to include operations that can change the access rights of users over resources, thereby addressing the dynamic aspects of access control.

Basic operations to update the access control matrix include:

- Create (or destroy) a subject $< s >$
- Create (or destroy) a object $< o >$
- Add (or remove) $< permission >$ into $[s, o]$ matrix

Transitions are sequences of basic operations.

For example, one to create a file could create and object f, add "own" into $[u, f]$ and add "read" into $[u, f]$. But this transition is not always valid and neither it is safe. If the file f already existed, this transition would allow the user u to access a file that he does not own, stealing it.

We need a way to check if a transition is safe or not, so to allow it or not. So, we need to be able to understand if a malicious action is possible given the list of specific actions that are forbidden and allowed.

More formally:

> Given an initial protection state and set of transitions, is there any sequence of transitions that leaks a certain right r (for which the owner is removed) into the access matrix?
> If not, then the system is safe with respect to right r.

Unfortunately, this is a version of the halting problem, and (thus) it is an undecidable problem in a generic HRU model.

It is decidable in mono-operational systems, but these systems are pretty useless (you cannot, for example, create a file and own it), or in systems in which subjects and objects have a finite number (so cannot grow).

Being it an undecidable problem, we cannot be sure that a transition done by a user would not compromise the security of a system.

Then, instead of forbidding specific actions, we have to prevent the users from doing some things.

DAC implementations

DAC implementations are a reproduction of HRU models.

Implementing the access matrix would be inefficient, because it is a sparse matrix. So, alternative implementations exist:

- **Authorizations table**: it records non-null triples S-O-A, typically used in DBMS. It is less efficient than the access matrix, but a safer implementation.
- **Access Control Lists**: records by column (i.e., for each object, the list of subjects and authorizations)
- **Capability Lists**: records by row (i.e., for each subject, the list of objects and authorizations)

Access control lists correspond to the columns of the access control matrix and focus on the objects. They are used when a lot of operations on the objects are often performed, as they are efficient with per object operations, in systems with few users and a lot of objects.

The downside is that, in principle, with ACLs there cannot be multiple owners of the same resource, even if the problem is partially solved with the use of groups.

Because in the majority of OS there are few subjects and a lot of objects, the standard for OS is ACLs. Some systems (e.g. POSIX) use abbreviated ACLs.

fun.com	Alice: {exec}	Bill: {exec,read,write}

Capability lists correspond to the rows of the access control matrix and focus on the subject. They are used when a lot of operations on the subjects are often performed, as they are efficient with per subject operations, in systems with many users.

Capability lists are inefficient in classical OS where objects change, and subjects stay. They are usually implemented in distributed system with the aim to make it easy to add or remove subjects in systems in which a lot of users perform operations on the same fewer objects.

Alice	edit.exe: {exec}	fun.com: {exec,read}

The choice between one or the other is made depending on the number of users and resources in the system.

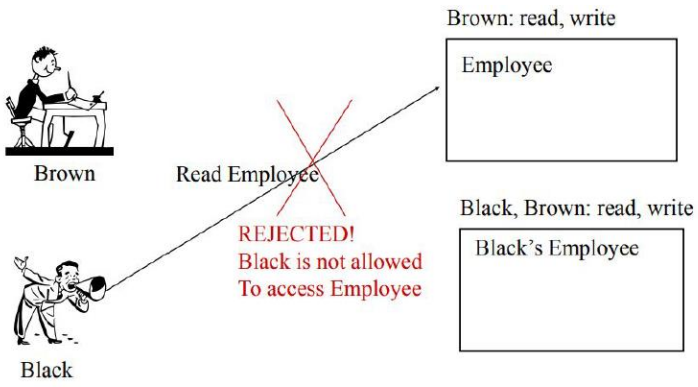
DAC shortcomings

General DAC shortcomings include:

- Cannot prove safety, as it is a non-decidable problem
- Control access to objects but not to the data inside objects (granularity)
  - Susceptible to malicious user problem, as if a user creates a transition to compromise the system, we cannot prove it, and every user may do it.
  - Susceptible to trojan horse problem: malicious program running with privileges of the user. A user with privileges could use a program with a trojan to access a file, so the trojan can access it too.
- Problems of scalability and management
  - each user-owner can potentially compromise security of the system with their own decisions

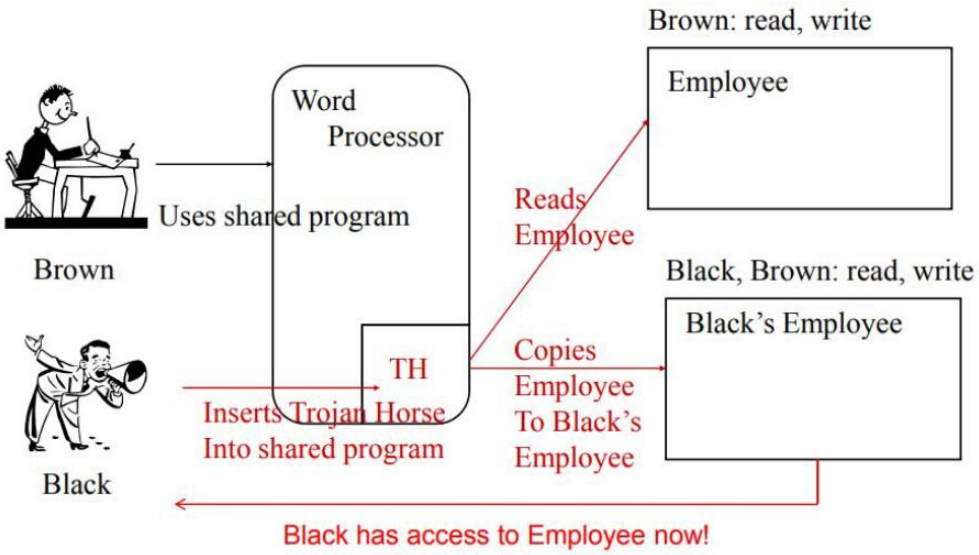
In a protected system, normally a user cannot access objects over which he does not have rights.

If user Black tries to read a file without having the permissions to do so, the action will be rejected.



But Black can use a trojan horse in the following way. Say there is a program, like a word processor, that is used by both Black and Brown, who has the right to access the file Black is interested to read but cannot.

When the word processor is used by Brown, it will run with Brown privileges. So, if black injects a malicious script (trojan horse) into the word processor, when brown uses it this script will be executed with brown privileges and open the file. This way, Black can gain access to the content of the desired file.



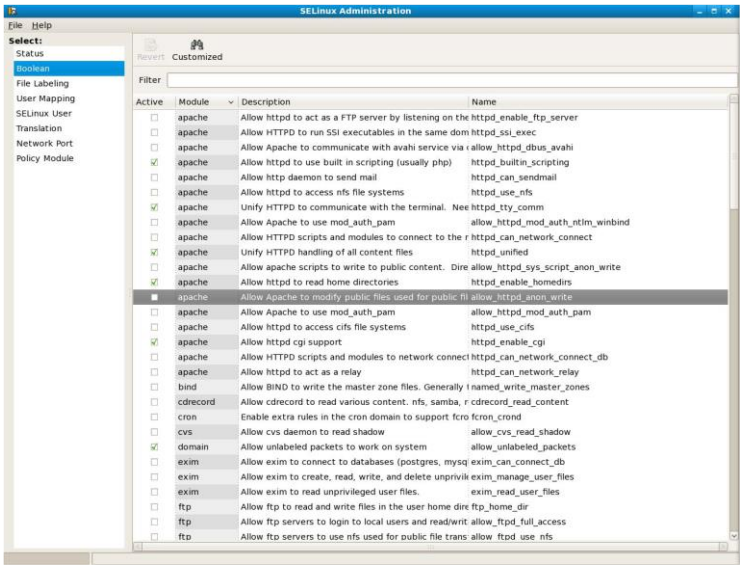
Mandatory Access Control (MAC)

The basic idea of **Mandatory Access Control** (**MAC**) is to not let owners of the objects assign privileges. Privileges are instead set and assigned by the security administrator of the system.

One way to do so is to define a classification of *subjects* (or "clearance") and *objects* (or "sensitivity").

The classification is composed of a strictly ordered set of secrecy levels and set of labels.

For example, MAC rules in SELinux look like this:



Secrecy levels assigned to resources in the system can look like this (US example):

$$Top\ Secret > Secret > For\ Official\ Use\ Only\ (FUOU) > Unclassified$$

Or like this (NATO example):

$$COSMIC\ Top\ Secret > NATO\ Secret > NATO\ Confidential > Unclassified$$

A **label** is also assigned to resources, based on a set of possible labels, for example:

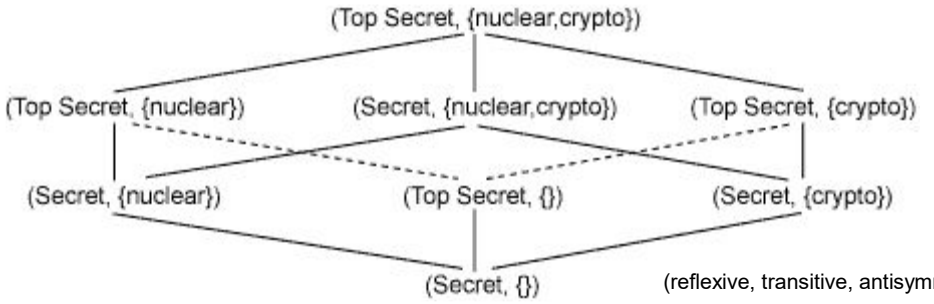
$$Policy, Energy, Finance, ATOMAL, NOFORN, \ldots$$

A possible implementation is the **lattice**-based access control.

Lattice-based access control (LBAC) is a security model that uses a lattice structure to define levels of access and authority. In LBAC, each security tuple:

$$(Security\ Level, Label)$$

is associated with a point in the lattice, representing different levels of sensitivity and clearance. Users and data objects are assigned these tuples, and access decisions are made based on the hierarchical relationship in the lattice.



(reflexive, transitive, antisymmetric property)

A user can access data if their clearance level dominates (is equal to or higher than) the data's classification level, ensuring a structured and hierarchical approach to managing permissions and maintaining security. In other words, a user's lattice must dominate the one of a resource in order for the user to be able to access the resource.

A lattice dominates another if the secrecy level of the first is higher than the one of the second and the second label is a sub-set of the first label.

$$\{C_1, L_1\} \geq \{C_2, L_2\} \quad iff \quad C_1 \geq C_2 \; and \; L_2 \subseteq L_1$$

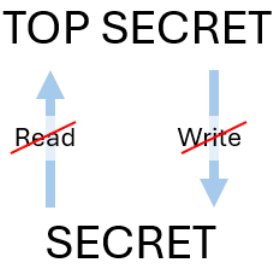
Lattice-based access control is typically used in environments requiring strict, hierarchical data classification and clearance, such as military, government, and certain corporate settings, to ensure that users access only the information appropriate to their security clearance level.

Bell-LaPadula Model (BLM)

The **Bell-LaPadula model** (**BLM**) is a security model combining aspects of MAC and DAC.

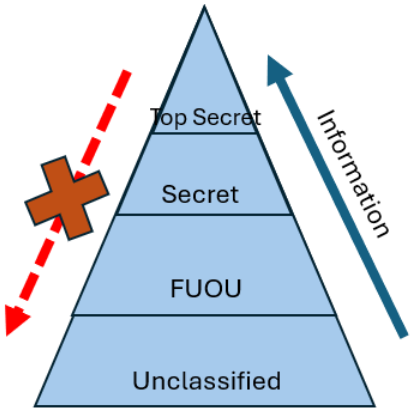
It enforces two main MAC rules and one DAC rule:

- Rule 1 (**no read up**, "*simple security property*"): a subject s at a given secrecy level cannot read an object o at a higher secrecy level.
- Rule 2 (**no write down**, "*star property*"): a subject s at a given secrecy level cannot write an object o at a lower secrecy level. This is to avoid data leakage from a higher level to a lower one.
- Rule 3 (*Discretionary Security Property*) states the use of an access matrix to specify the discretionary access control.

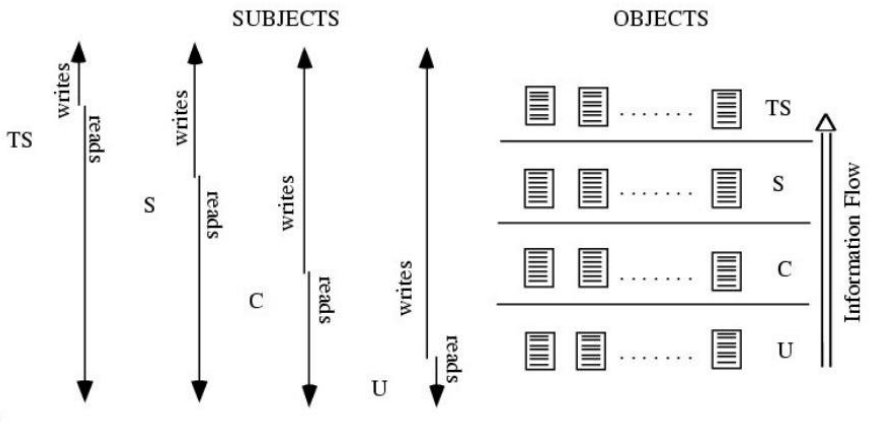


Tranquility property: secrecy levels of objects cannot change dynamically.

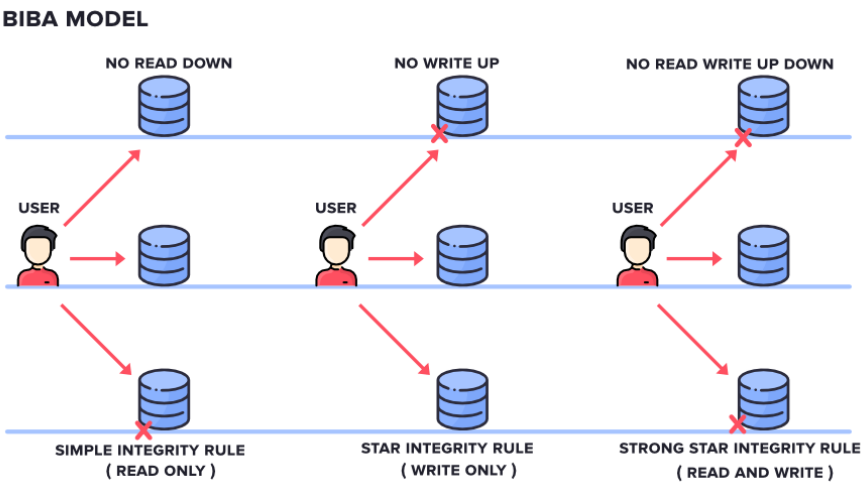
The result of enforcing this combination of rules is to obtain a monotonic flow of information toward higher secrecy levels, so information cannot go from higher to lower levels but tend to go up.



This flow calls for the need of trusted subjects who can declassify or sanitize documents.



A limitation of BLP is that it does not address integrity of information problems. There are other models for integrity, like the **Biba Integrity Model**. In this last model, data and subjects are grouped into ordered levels of integrity and the model is designed so that subjects may not corrupt data in a level ranked higher than the subject or be corrupted by data from a lower level than the subject.



Conclusions

Access control, or authorization, defines subjects, objects, and actions in a system.

Access control models define how actions are (un)assigned to subjects and objects.

DAC are more common and “natural” than MAC but can coexist.

5. Introduction to software security

A good software is one that meets the requirements, so behaves like it is supposed to.

Functional requirements	**Non-functional requirements**
Software must do what it is designed for. For example: • Data Input • Reporting • Notification • Etc.	Software must respect constraints and quality standards. • Usability • Safety • Security • …

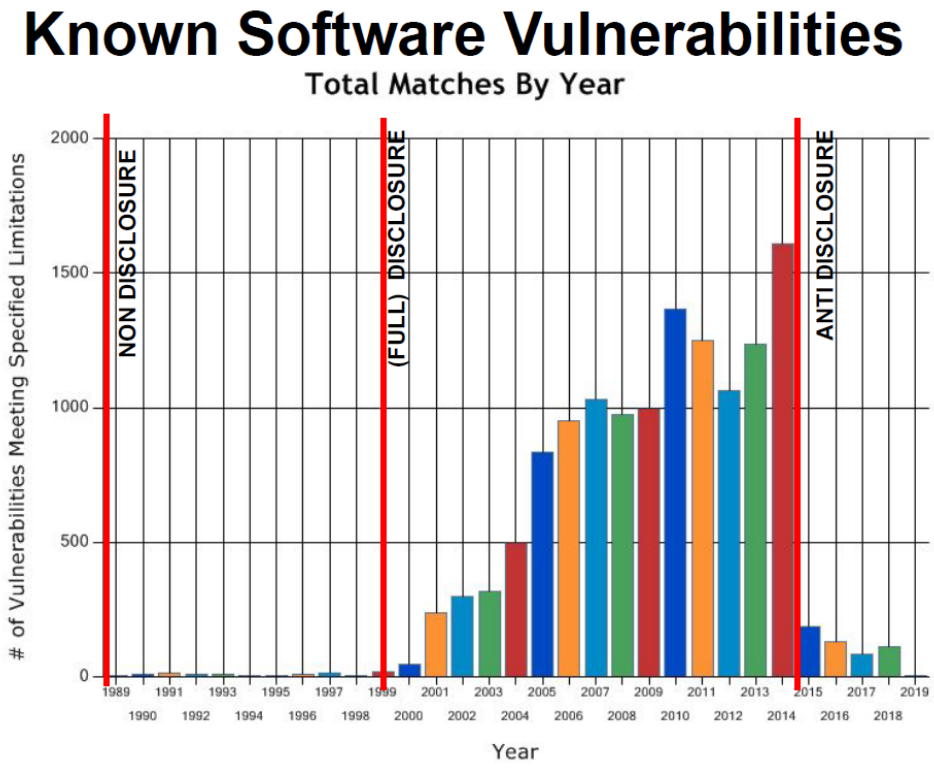
Creating inherently secure applications is a fundamental, yet often unknown, skill.

Software has vulnerabilities. Software should implement the specifications; an unmet specification means there is a software bug, while an unmet security specification means there is a vulnerability. A security bug can be exploited to make the software behave differently.

A way to leverage a vulnerability to violate the CIA is called exploit (knowing the vulnerability does not necessarily mean also knowing one related exploit). An exploit for a software is generally a program (piece of code) that interacts with the input of a non-secure program in order to exploit a vulnerability.

Life of a software vulnerability

The number of disclosed vulnerabilities disclosed each year varies with the period.



We can distinguish three main periods.

During the **non-disclosure** period, technology was still not so advanced and hacking groups kept vulnerabilities for themselves as a demonstration of the power of the hacking group. Exploiting vulnerabilities was often a game or a way to show off.

During the **full-disclosure** period, internet becomes popular and all machines started to be exposed to internet, so no physical access to them was needed to exploit some vulnerability. Hackers made the found vulnerabilities public with the objective to put pressure on vendors to patch these vulnerabilities,so to protect their own systems.

The reaction of vendors and companies was more than often to negate the vulnerabilities and denigrate the hackers reporting them.

Subject: Comments on the dvwssr.dll vulnerability threads
From: Iván Arce
Date: **2000-04-18** 1:25:52

I do not intend to go further down the full disclosure vs. mediated release of information discussion here, however [Microsoft's handler's] post on NTBugtraq regarding CORE's work requires some clarifications on our side.

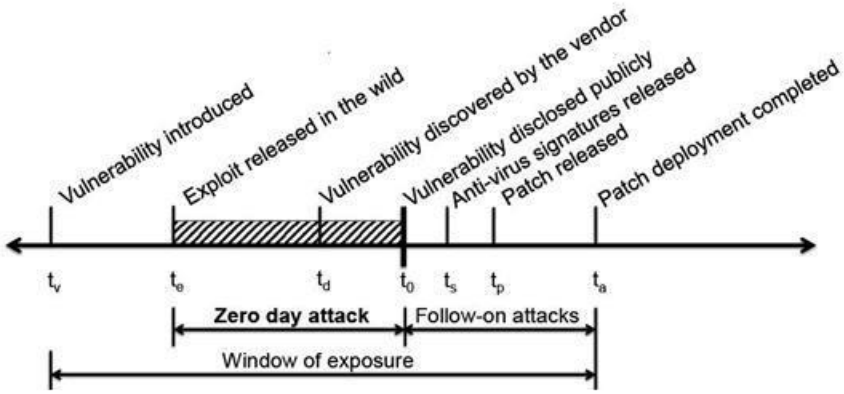
[...]

If someone yells 'FIRE' and that appears to be reasonable, I'd would be very careful in my methodology and editorial policies before yelling "NOT TRUE! NOT TRUE! EVERYTHING IS FINE!".

[...]

Excuse me if I'm being rude, but **I'm shocked by the fact that our company is being questioned because we found a bug.**

During the full-disclosure period, vulnerabilities lyfecycle could be schematized in some fixed phases (with possibily varying order of events).

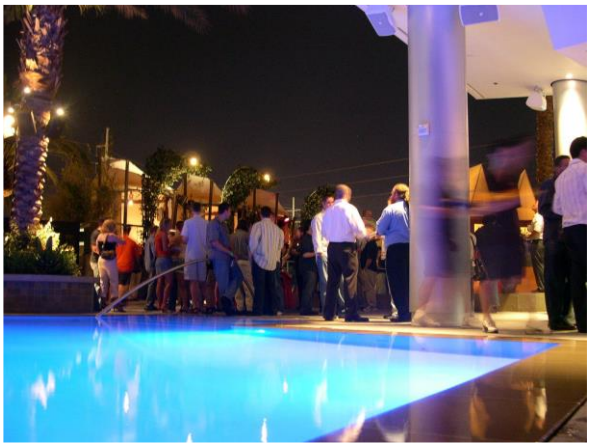


A vulnerability is discovered and at some point the exploit found, so the first attacks are performed. The vendor finds the vulnerability and starts to develop the patch, hoping to fix the vulnerability before the exploit becomes public. But after a while, the vulnerability is disclosed publicly, and follow-on attacks start coming from potentially everyone. The window of exposure of a software is closed when the patch is not only released, but also its deployment is completed.

The time to develop and distribute a patch can be considered approximately fixed.

A possible solution to reduce the window of exposure is to introduce automatic updates with patches. This way, the user doesn't need to be infromed, find and install the patch but this happens automatically and as soon as possible.

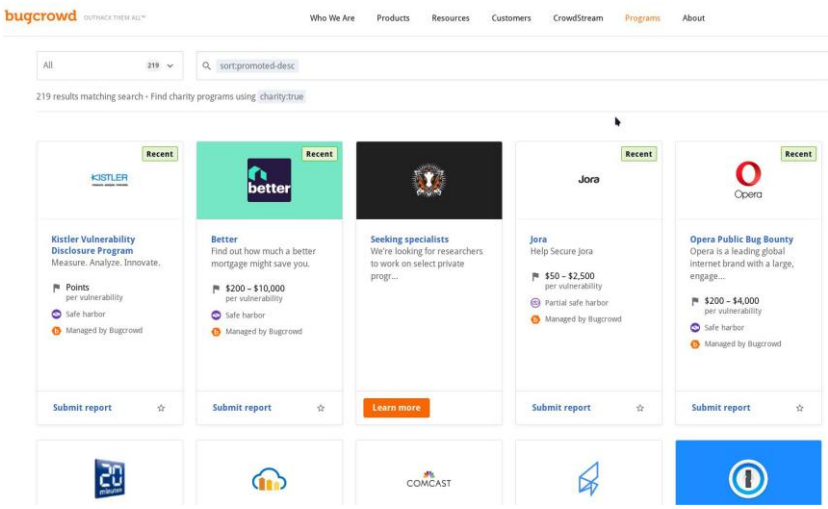
Another solution was to throw *black hat parties* where to meet hackers and convince them to disclose and release the vulnerabilities they have found to the companies before making them public, or propose to hire them or collaborate.



The last period is the **anti-disclosure** period, in which discovered vulnerabilities are kept secret and released only in exchange of good money.

Exploits are sold by hackers to various actors: companies, rival companies and governments.

In order to find vulnerabilities before someone buys them and uses them against a company, that company can organize *bug bounties*, contests authorized by the software vendors in which they pay hackers to find vulnerabilities in their software.

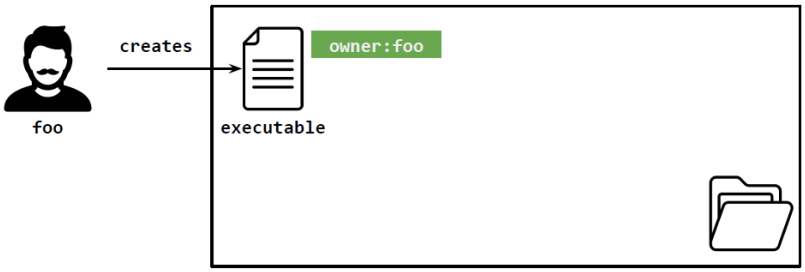


The important thing for white hat hackers it to search vulnerabilities in agreement with companies and in the legal way.

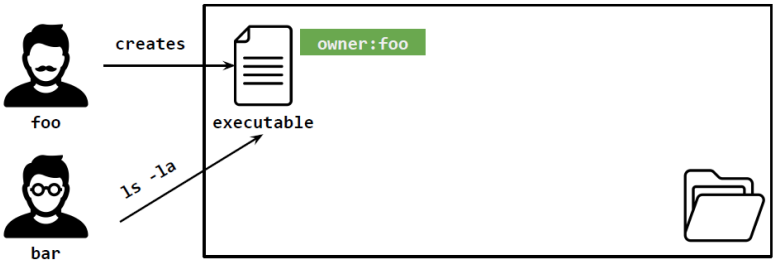
Vulnerability and exploit in UNIX-like system (example)

We will see a vulnerability related to permissions In UNIX systems.

Every file has an owner, that usually is the creator of that file.



Any other user of the system can usually check the permissions related to a file or executable.



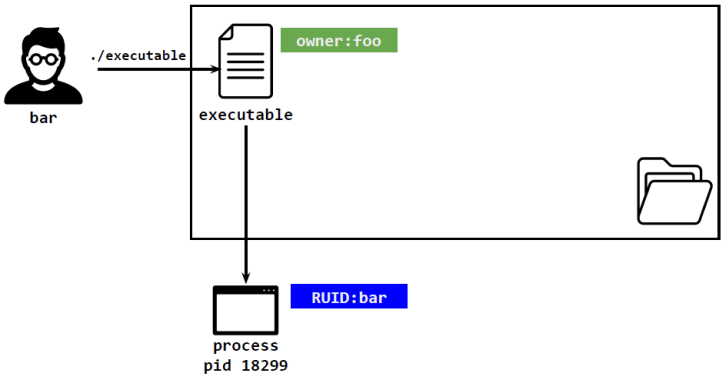

```
[bar@localhost]$ ls -la executable
 -rwxr-xr-x 1 foo group 41836 2012-10-14 19:19 executable
```

In this case, owner if foo and has $read, write, execute$ provileges, while other users, like bar can $read, execute$.

The **Real UID** (RUID, Real user ID) is the real owner of a process. The RUID can differ from the owner of a file or executable.

```
[bar@localhost]$ ./executable   #Darwin Kernel 13.1.0
[bar@localhost]$ ps -a -x -o user,pid,cmd
 USER  PID    COMMAND
 bar   18299  ./executable
```

When an executable is run, a process that is run under the RUID of the user will be spawned. This process has the same permissions of the user pointed by the RUID.

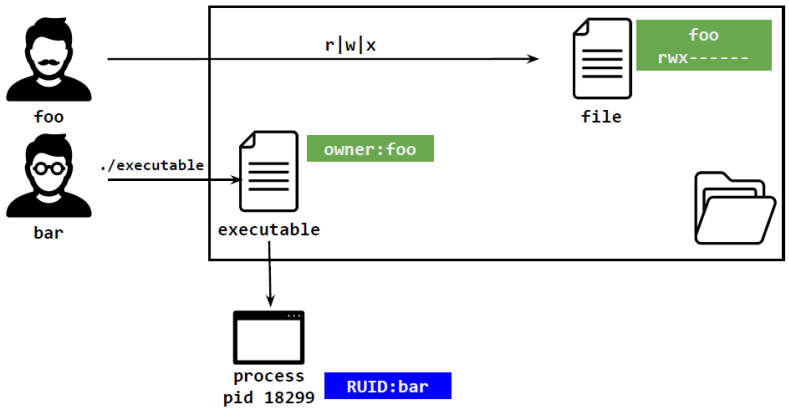


In our case, the process with RUID bar has the same permissions of user bar.

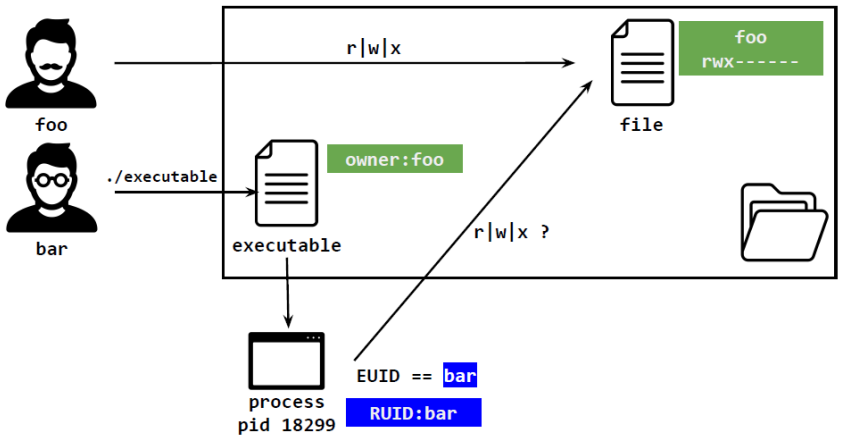
The executable will run under the privileges of your user, so also the process has the same permissions of the effective user. To check the permissions of a process, the **Effective UID** (EUID; Effective user ID) is used.

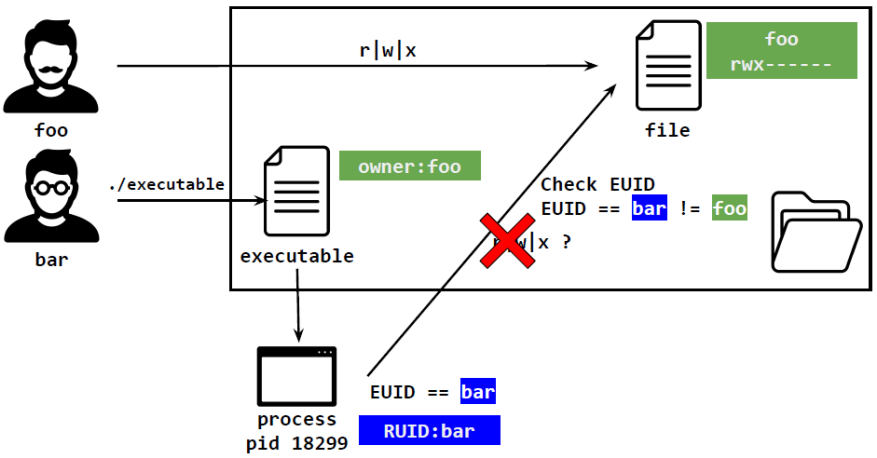
Normally the RUID coincides with the Effective UID.

Normally: $RUID == EUID$



A process spawned by bar runs with the EUID of bar. If this process tries to access a file for which bar has no permissions, the access will be denied.





In our case, the file can be $r|w|x$ only by foo, so access to bar is denied.

Programs may need to run privileged instructions, that the user is not allowed to perform normally (e.g. to send a network packet).

The **Saved Set-user-ID** (SUID) can be used to change the EUID at runtime, in order to perform operations that need special permissions, and is meant to be used for example for syscalls.

First thing to do for a process is to become root, setting the EUID to the one of the owner. To do so, of course, root privileges are needed.

At this point, commands like $chmod$, that allows to change the permissions associated to an executable, can be executed. With $u + s$ argument, the command is used to set the SUID of the file to the one of owner of a file.

After this command, the executable will be executed under the EUID of its owner.

```
[root@localhost]# chmod u+s executable
[root@localhost]# ls -la executable
 -rwsr-xr-x 1 foo group 41836 2012-10-14 19:19 executable
```

In our case, now the executable's SUID becomes foo.

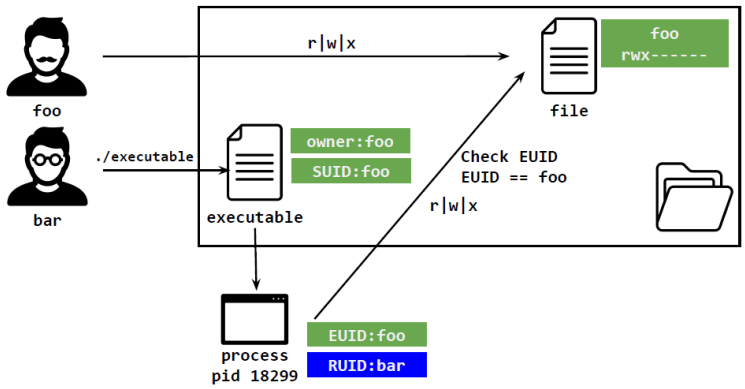
```
[bar@localhost]$ ./executable
[bar@localhost]$ ps -a -x -o user,pid,cmd
 USER  PID    COMMAND
 foo    18299  ./executable
```

bar is now running with the EUID of foo.

$$bar == RUID \neq EUID == foo$$

The objective could be to exploit a program that has higher privileges than yours to perform an operation.

For example, if an attacker (a malicious user) manages to make a program, that has the permission of roots, spawn a shell, then the commands in that shell will run with the permissions of root.



Leaving an unmanaged permission like the one to become root and change the EUID will live an entrance for the attacker.

A border case is the one in which the owner of a file SUID is $root$.

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, const char *argv[])
{
    printf("RUID %d EUID %d", getuid(), geteuid());
    return 0;
}
```

```
[foo@localhost]$ gcc -o executable -c executable.c
[foo@localhost]$ sudo su -                # become root
[root@localhost]# chown root                 # change the owner
[root@localhost]# chmod +s executable     # set the SUID root bit
[root@localhost]# exit                       # get back to foo
[foo@localhost]$ ls -la executable           # check the flags
 -rwsr-xr-x 1 root group 41836 2012-10-14 19:19 executable
```

An attacker shouldn't be able to become $root$, and later change the owner of a file to $root$ and the SUID. When a file Is executed by the attacker after this procedure, it will run with $EUID == root$.

Take now an executable with RUID if foo and EUID of $root$.

If executed as root, so with root privileges, the RUID becomes the one of $root$.

```
[foo@localhost]$ ./executable
RUID 501 Effective 0                        # 501 is foo's UID - 0 is root's UID

[foo@localhost]$ sudo -u root ./executable
RUID 0 Effective 0
```

Now, let's consider a program that opens a file, scans it and puts its content in an array and prints it.

```c
[foo@localhost]$ vim executable.c          // let's add a privileged instruction
#include <unistd.h>
#include <stdio.h>

int main(int argc, const char *argv[])
{
    FILE * fp;
    char line[1024];

    printf("Real %d Effective %d", getuid(), geteuid());

    fp = fopen("/etc/secret", "r");
    while (!feof(fp)) {
        fgets(line, 1024, fp);
        puts(line);
    }
    fclose(fp);
    return 0;
}
```

We are interested in accessing a file named $secret$, for which only $root$ has permission to access. A normal user cannot open this file.

```
[foo@localhost]$ ls -la /etc/secret
-rwx------  1 root  wheel  12 Mar 10 16:07 /etc/secret
```

If we execute the program on this file to read it without a SUID of $root$, it is going to fail to open the file.

But, if the EUID is changed to the one of $root$ before executing the program, the file will be accessed.

```
[foo@localhost]$ ls -la /etc/secret
-rwx------  1 root  wheel  12 Mar 10 16:07 /etc/secret

[foo@localhost]$ ./executable
Real 501 Effective 0
s3cr3t inf0
```

Programs are "SUID root" to allow them to execute privileged instructions.

What if the program licitly needs to gain root permissions to do something? After gaining permissions, the entire program is run with the SUID of the root, even if this is necessary only for some operations.

The aim of a defensive strategy should be to reduce the attack surface, while preserving functionality. So, the EUID should be changed back as soon as possible once the privileged instructions are done.

```
#include <unistd.h>
#include <stdio.h>

int main(int argc, const char *argv[])
{
                    // execute as EUID ------------------------
    char line[1024];
    FILE * fp;

    printf("Real %d Effective %d\n", getuid(), geteuid());

    fp = fopen("/etc/secret", "r");
    fgets(line, 1024, fp);
    fclose(fp);

    setuid(501);    //execute as unprivileged user
    printf("Real %d Effective %d\n", getuid(), geteuid());

    puts(line);

    return 0;
}
```

Once we read the file, we need to release the privileges, so that the subsequent instructions are executed with foo's privileges.

> The problem may not be solved yet, as the program might print the content of a file or needed information inside error messages

General idea

Let's analyze the pseudocode of a vulnerable program.

Vulnerable program

```
EUID: RUID -> SUID
read(config)
r = parse(config)
IF r = OK do_things() ELSE
error("...")

[user@host]$ ./ex /etc/shadow

ERROR in file, line 1:
root:<password hash>: ...
```

The program is supposed to read a file, parse it and, if the parsing is successful, do something, otherwise throw an error.

The program has root privileges to read files, so if a user cannot read something, this program can do it for him.

The $read(config)$ function prints the content of the file in the error message. This allows an unprivileged user to print the content of, e.g., the $/etc/shadow$ file, which can be normally read only by privileged users.

To fix this problem, we modify the program so to acquire the higher privileges as late as possible.

Fixed program

```
EUID: SUID -> RUID
read(config)  //low privs
EUID: RUID -> SUID
r = parse(config)
IF r = OK do_things() ELSE
error("...")
[user@host]$ ./ex /etc/shadow

Permission denied.
```

If a user tries to exploit the program to read something, he will fail because permission will not be granted as the $read()$ function is executed with the root privileges that will be granted later.

By acquiring higher privileges only after the file is read, the developer decreases the attack surface and effectively eliminates this specific vulnerability (there may be other vulnerabilities).

Another vulnerability

There is still a vulnerability in the vulnerable part of the program (the part with higher privileges).

(still) vulnerable program

```
EUID: SUID -> RUID
read(config)  //low privs
EUID: RUID -> SUID
r = parse(config)
IF r = OK do_things() ELSE
error("...")


[user@host]$ ./ex
carefully-crafted-file
```

Any bug in the $parse(config)$ function would happen in a privileged portion of the code, therefore potentially allowing the attacker to perform actions. To exploit this vulnerability, a malicious user may want to pass to this program an input file cracked such in a way to cause some behavior of the $parse()$ function, for example making it crash.

A way to fix the program is to follow the principle of acquiring the privileges as late as possible, and releasing them as soon as possible, so to decrease further the attack surface.

Fixed program

```
EUID: SUID -> RUID
read(config)  //low privs
r = parse(config)
IF r = OK
    EUID: RUID -> SUID
    do_things()
    EUID: SUID -> RUID
ELSE error("...")
```

However, there may still be other vulnerabilities in the $do\_things()$ part of code.

Vulnerability	Exploit
The developer acquired the privileges before $read(config)$	Invocation of the program with $/etc/shadow$ as the first argument.
The developer acquired the privileges before $parse(config)$	Invocation of the program on a specifically crafted file to exploit a vulnerability inside the configuration file

The problem often lies in wrongly managed permissions.

Key issues and principles of secure software design

To design a software with attention on security issues, we should:

- Reduce privileged parts to a minimum, so to reduce the attack surface.
- Follow the KISS (Keep It Simple, Stupid) principle, so that it is easy to verify requirements and find the vulnerabilities.
- Discard privileges definitively (i.e. SUID->RUID) as soon as possible.
- Open design: just as with Kerckhoffs principle, the program must not rely on obscurity for security, but relying on the community to find the vulnerabilities can be more effective.
- Pay attention to concurrency and race conditions that are tricky and may cause logic vulnerabilities.
- Fail-safe and default deny (allowlists are better than blacklists).
- Avoid the use of shared resources (e.g. mktemp) or unknown, untrusted libraries.
- Filter the input and the output.
- Do not write any own crypto, password, and secret management code: use trusted code that has been audited already.
- Use trusted entropy sources (RNGs) such as $/dev/urandom$

Conclusion

Bug-free software does not exist.

Not all bugs lead to vulnerabilities.

Vulnerability-free software is difficult to achieve.

Vulnerabilities without a working exploit exist.

Be careful with the SUID permission bit.

6. Buffer Overflows

A **buffer** is a limited and continuously allocated set of memory.

A **buffer overflow** is a memory exploitation technique consisting in an attack that exploits the capability to go over the boundaries of a buffer in memory, to operate on other data.

Buffer overflows vulnerabilities are present when there is no check on the input size in programs, and also the CPU does not distinguish between code and data. The attacker, in fact, usually wants to overflow the memory in order to put his code in the memory and make it be executed.

Let's start from the beginning.

We will analyze buffer overflows for 32-bit x86 machines for which we assume ELFs running on Linux ≥ 2.6 processes running on top. The following concepts apply, with proper modifications, to any machine architecture (e.g., ARM, x86), operating system (e.g., Windows, Linux, Darwin), and executable (e.g., Portable Executable (PE), Executable and Linkable Format (ELF)).

Dear reader, I'm sorry but this document ends here, because while studying for my exam I ran out of time and decided to stop writing this summary.
I hope that you found the incomplete text useful so far.

Dear Luca of the future, if you have spare time, finish this. You found the subject really interesting and going over it again (maybe updating to the state of the art some info) would be nice.