

Computer & Network Security

Giuliano Abruzzo

February 2, 2020

Contents

1	Introduction	5
2	Secret Key: Stream Ciphers & Block Ciphers	7
2.1	Stream Cipher	8
2.1.1	A5	8
2.1.2	RC-4	8
2.2	Block Ciphers	9
2.2.1	AES	10
2.2.2	ECB	14
2.2.3	CBC	15
2.2.4	PCBC	16
2.2.5	CFB	17
2.2.6	OFB	18
2.2.7	CTR	19
2.2.8	Initialization Vector	19
2.2.9	Strengthening a cipher	19
3	Data Integrity & Authentication	20
3.1	MAC	20
3.2	MAC based on CBC	21
3.3	MAC based on hash functions	21
3.3.1	SHA-1	23
3.3.2	HMAC	24
3.3.3	AE	24
4	Public Key Cryptography	26
4.1	Public Exchange of Keys	26
5	RSA	28
5.1	Math considerations	28
5.2	RSA Protocol	28
5.3	Attacks against RSA	31
5.3.1	Factorization	31
5.3.2	Weak Messages	31
5.3.3	Chinese Remainder Attack	32
5.3.4	Same N	32
5.3.5	Multiplicative Property of RSA	32
5.3.6	Chosen Ciphertext Attack	32
5.3.7	Chosen Plaintext Attack	32
5.4	RSA Standards	33
5.4.1	Public-Key Cryptography Standard	33
5.4.2	Optimal Asymmetric Encryption Padding	33
5.4.3	El Gamal Encryption	34

6 Digital Signature - DSA	34
6.1 Standards for Digital Signatures	35
6.1.1 RSA and PKCS#1	35
6.1.2 El-Gamal Signature Scheme	36
6.2 DSS, Digital Signature Standard	37
7 Authentication	39
7.1 Authentication by Symmetric Key	40
7.1.1 Challenge/Response Authentication	40
7.1.2 Timestamp Authentication	41
7.1.3 Mutual Authentication	41
7.2 Authentication through Third-Party	42
7.2.1 First schema	42
7.2.2 Needham-Schroeder Protocol	43
7.2.3 Needham-Schroeder Protocol variant	43
7.2.4 Needham-Schroeder Protocol Expanded	44
8 Kerberos	44
8.1 Kerberos preliminary implementation	45
8.2 Kerberos simplified version	45
8.3 Ticket-granting Ticket	46
8.4 Kerberos Realms	48
9 Authentication based on public keys & X.509 & PKI	49
9.1 Needham-Schroeder public key	49
9.1.1 Needham-Schroeder public key Attack	50
9.1.2 Needham-Schroeder public key fixed variant	50
9.2 X.509 Standard	51
9.3 PKI: Public Key Infrastructure	52
9.3.1 X.509 Certificate's Fields	52
9.3.2 Hierarchy of CAs	53
9.3.3 Certificate Revocation	53
9.3.4 OCSP	53
9.3.5 PGP: Pretty Good Privacy	54
10 Passwords	54
10.1 Unix Password Hash	54
10.2 Lamport's Hash	55
10.3 Encrypted Key Exchange, EKE	56
10.3.1 SPEKE & PDM	57
11 IPSEC	58
11.1 Security Associations (SA)	60
11.2 Transport & Tunnel Modes	61
11.2.1 Transport Mode	61
11.2.2 Tunnel Mode	62
11.3 Authentication Header AH	63

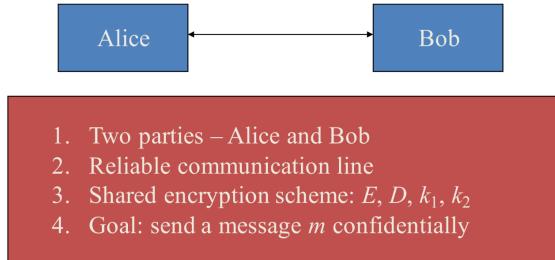
11.4	Encapsulating Security Payload ESP	64
11.5	Combining Security Associations: SAs	65
11.5.1	Transport Adjacency	65
11.5.2	Transport-Tunnel bundle	65
12	SSL/TLS	66
12.1	SSL Architecture	67
12.2	Handshake Protocol	69
12.2.1	Fixed Diffie-Hellman	69
12.2.2	Ephemeral Diffie-Hellman	70
12.2.3	Anonymous Diffie-Hellman	70
12.3	Attacks against TLS	70
12.3.1	Downgrade Attack	70
12.3.2	Heart Bleed	70
13	Firewalls	70
13.1	Packet Filtering	71
13.2	IPtables	72
13.2.1	Examples of IPtables	72
13.3	Bastion Host	73

1 Introduction

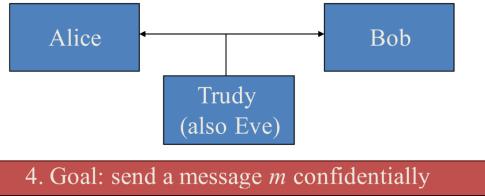
Cryptography and Security differ, in fact *Cryptography* deals with **secrecy** of information, *Security* deals with problems of *fraud*, like *message modifications* or *user authentication*. *Security* might use *Cryptography*, and **encryption** doesn't live alone without some form of *authentication*. We will call:

- **Encryption function:** E ;
- **Decryption function:** D ;
- **Encryption key:** k_1 ;
- **Decryption key:** k_2 ;
- For every message m : $D_{k_2}(E_{k_1}(m)) = m$;
- **Secret Key (symmetric):** $k_1 = k_2$;
- **Public Key (asymmetric):** $k_1 \neq k_2$;

A **Threat** is a *menace*, a source of danger, instead an **Exploit** is a *software* or a *chunk of data*, or a *sequence of commands* that take advantage of a *vulnerability* to cause *unintended* or *unanticipated* behavior to occur on computer. The **communication model** is:



Instead a **Threat (attack) model** is:



The **Adversary** can be **Passive**, that means that will reads the exchanged *messages* (without changes), or **Active**, that means that can modify the *message* between *Alice and Bob*, or can send *fake messages* claiming that they have been sent by someone else (*Alice or Bob*). The *passive adversary model* is also called **Packet Sniffing**, instead an *active adversary model* is also called **IP Spoofing**, in which T is able to forge *messages* that look like *messages* sent by A or B by

modification of the *IP header*. A *security threat* is **Denial of Service** or **DoS**, in which *attackers* send a lot of *packets* to the *attacked host*, a **DDoS** is a *distributed attack* through infection of *unaware computers*, often with the use of *SYN packets*.

If the *keys* are unknown then:

- It's **hard** to obtain even partial information on the *message*;
- It's **hard** to find the *key* even if we know *clear text*;

For *hard* we intend **Computationally hard**, that means that it takes a long time even with the most *powerful computers*. It's important to note that the *goal* is to:

- No *adversary* can determine *message M (not enough)*;
- No *adversary* can determine some *information about message M (not enough)*;
- No *adversary* can determine any *meaningful information about message M (good)*;

The *adversarial* attempts to discover information about the *message M*, and knows the *algorithms E, D*, the *message space*, and has at least *partial information* about $E_{k_1}(M)$, but doesn't know about k_1, k_2 . The **Plaintext** is the *message prior to encryption*, instead the **Ciphertext** is the *message after encryption*.

The **Perfect Cipher** is a *cipher* in which given a *Plaintext space* $= \{0, 1\}^n$, where D is known, given a *ciphertext C* the probability that exists k_2 such that $D_{k_2} = P$ for any *plaintext P* is equal to the apriori probability that P is the *plaintext*. In other worlds. the *ciphertext* doesn't reveal any information about the *plaintext*:

$$Pr(P|C) = Pr(P)$$

And for the *conditional probability* we have that in a *perfect cipher*:

$$Pr(C) = Pr(C|P)$$

An example of a *perfect cipher* is the **One Time Pad** in which we have:

- Plaintext space and Key space: $\{0, 1\}^n$;
- Symmetric scheme and k is chosen random;
- $E_k(P) : C = P \oplus K$;
- $D_k(P) : P = C \oplus K = P \oplus K \oplus K$;

Where \oplus is an *exclusive OR*, bit by bit. The problem is the size of *key space* since for the **Theorem of Shannon**: a *cipher* cannot be *perfect* if the size of its *key space* is less than the size of its *message size*.

The **proof** is done by *contradiction*:

- We assume that the number of the keys l is less than the number of messages n so $l < n$ and we consider ciphertext $C_0 : Pr(C_0) > 0$;
- For some key k , consider $P = D_k(C_0)$ So there exists at most l keys such messages, one for each key;
- Choose message P_0 such that it is not of the form $D_k(C_0)$ since there exists $n-l$ such messages;
- Hence $Pr(C_0|P_0) = 0$ but since in a perfect cipher $Pr(C_0|P_0) = Pr(C_0) > 0$ we have a contradiction;

We have several different **attack models** like:

- **Eavesdropping**: in which the *attacker* secretly listening private conversation of others;
- **Known Plaintext**: *attacker* has samples of both *plaintext* and its *encrypted version* so will use them to reveal information like *secret keys*;
- **Chosen Plaintext**: *attacker* has the capability to choose arbitrary *plaintexts* to be *encrypted* and obtain the corresponding *ciphertexts*. The goal is to gain information which will reduce the security of the *encryption scheme*.
- **Adaptive Chosen Plaintext**: the *crypt-analyst* makes a series of interactive *queries*, choosing *subsequent plaintexts* based on the information from the previous *encryptions*;
- **Chosen Ciphertext**: the *crypt-analyst* gathers information, at least in part, by choosing a *ciphertext* and obtaining its *decryption* under an unknown key;
- **Physical Access**;
- **Physical modification of messages**;

2 Secret Key: Stream Ciphers & Block Ciphers

The idea is that *Alice and Bob* share:

- a **crypto protocol** E ;
- a **secret key** k ;
- they *communicate* using E with *key* k ;
- *adversary* knows E and some *messages* but ignores k ;

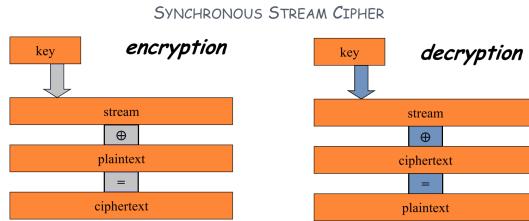
There are two different approaches **Stream Cipher** and **Block Cipher**:

2.1 Stream Cipher

The idea is try to simulate the *one-time-pad*, so we define a *secret key* called **Seed**, and we use the *seed* to generate a *byte stream* called **Keystream**:

- The i^{th} byte is function of:
 - Only key, **synchronous stream cipher**;
 - Both key and first $i - 1$ bytes of the *ciphertext*, **asynchronous stream cipher**;

In a **synchronous stream cipher** the output of the *encryption function* is generated independently from the *plaintext* and the *ciphertext*. So we have for **synchronous stream cipher**:



2.1.1 A5

A5 is an *encryption algorithm* (1987) used to provide *over-the-air communication privacy* in the *GSM cellular telephone standard*. Initially it was kept secret, but the general design was leaked in 1994 and the *algorithms* were entirely *reverse engineered* in 1999.

2.1.2 RC-4

RC or *Ron's Code* was an *algorithm* easy to program, fast and very popular considered *safe* between 1987 and 1994, in which we have:

- Variable key length (byte);
- Synchronous stream cipher;
- Starting from the key it generates a random permutation;
- Eventually the sequence will repeat but for long period 2^{100} so its simulate *one-time-page*;
- It's very fast in fact 1 byte of *output* requires 8-16 *instructions*;
- The goal is to generate random permutation of the first 256 natural numbers;

Algorithm RC-4 Init

```

 $j = 0;$ 
 $S_0 = 0, S_1 = 1, \dots, S_{255} = 255;$ 
assume a key of 256 bytes:  $k_0, \dots, k_{255}$  (if the key is shorter repeat);
for  $i = 0$  to 255 do:
   $j = (j + S_i + k_i) \bmod 256;$ 
  exchange  $S_i$  and  $S_j$ ;
  
```

So in this way we obtain a **permutation** of $0, 1, \dots, 255$, an the *resulting permutation* is a function of the *key*.

Algorithm RC-4 Key-Stream Generation

```

Input: permutation  $S$  of  $0, 1, \dots, 255$ ;
while (true):
     $i = (i + 1) \text{ mod } 256;$ 
     $j = (j + S_i) \text{ mod } 256;$ 
    exchange  $S_i$  and  $S_j$ ;
     $t = (S_i + S_k) \text{ mod } 256;$ 
     $k = S_t;$ 

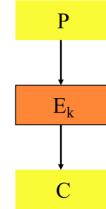
```

So at every *iteration* we compute the **XOR** between k and next *byte* of *plaintext* (or *ciphertext*). This *algorithm* is used in order to generate the *key* used for *encrypt* the next *byte* of the *plaintext*.

2.2 Block Ciphers

In the **Block Ciphers** instead we have:

- A **block** P of text of h bits (where h is fixed);
- A **key** k of fixed number of *bits*;
- A **cryptographic protocol** E_k produces a *block* C of h bits, function of P and k ;



There are several examples of **Block Ciphers** like:

- **DES**: (1976) with 64 *bit block* and 56 *bit key*;
- **RC-2**: (1987) designed by IBM, with 64 *bit block* and variable *key size*, vulnerable by an attack of 2^{34} chosen *plaintexts*;
- **IDEA**: (1991) with 64 *bit block* and 128 *bit key*, strong, only weaker variants have been broken;
- **Blowfish**: (1993) with 64 *bit block* and variable *key size* from 32 up to 448 bits, still strong;
- **RC-5**: (1994) with variable *block size* (32, 64, 128 bits) and variable *key size* (0 to 2040 bits) and *rounds* (0 to 255).
- **AES**: (2001) with 128 *bit block* and 128-256 *bit key*, very strong;

2.2.1 AES

AES or **Advanced Encryption Standard** is an *algorithm* based on *symmetric block cipher* in which we have a *block size* of 128 bits and a *key lengths* of 128, 192 or 256 bit that use **finite fields algebra**, and now have fun with *settordici* mathematical formulas and almost useless concepts:

- A set G with *binary operation* $+$ (*addition*) is called a **commutative group** if:
 - $\forall a, b \in G, a + b \in G;$
 - $\forall a, b, c \in G, (a + b) + c = a + (b + c);$
 - $\forall a, b \in G, a + b = b + a;$
 - $\exists 0 \in G, \forall a \in G, a + 0 = a;$
 - $\forall a \in G, \exists -a \in G, a + (-a) = 0;$
- Let $(G, +)$ be a group, then $(H, +)$ is a **sub-group** of $(G, +)$ if it is a group and $H \subseteq G$;
 - The *Theorem of Lagrange* says that: if G is finite and $(H, +)$ is a *subgroup* of $(G, +)$ then $|H|$ divides $|G|$;
- Two natural a and b are said to be **congruent modulo n** (with n a positive integer): $a \equiv b \pmod{n}$ if:
 - If $|a - b|$ is multiple of n or the integer division of a and n and of b and n yield the *same remainder*;
 - The *congruence relation* is *reflexive*, *symmetric* and *transitive*, hence it is an *equivalence relation*;
 - The *quotient set* Z_n is the set of n classes of *equivalence congruent* to $0, 1, \dots, n - 1$: $-1 \equiv n - 1 \pmod{n}, -2 \equiv n - 2 \pmod{n}$, etc.;
 - The properties of **congruence** are:
 - * **Invariance over addition:** $a \equiv b \pmod{n} \Leftrightarrow (a + c) \equiv (b + c) \pmod{n} \quad \forall a, b, c \in \mathbb{N}, \forall n \in \mathbb{N}_0$
 - * **Invariance over multiplication:** $a \equiv b \pmod{n} \Leftrightarrow (a \times c) \equiv (b \times c) \pmod{n} \quad \forall a, b, c \in \mathbb{N}, \forall n \in \mathbb{N}_0$
 - * **Invariance over exponentiation:** $a \equiv b \pmod{n} \Leftrightarrow a^k \equiv b^k \pmod{n} \quad \forall a, b, k \in \mathbb{N}, \forall n \in \mathbb{N}_0$
- Let's a^n denote $a + a + \dots + a$ for n times (*Why a^n and not $\sum_n a$? Only D'Amore knows, we spent two days wondering why the fuck he used this notation*), we say that a is **of order n** if $a^n = 0$:
 - For any $m < n, a^m \neq 0$ then all elements of finite groups have **finite order**, $a^n = 1$ for *multiplicative operator* (where a^n denotes $a \cdot a \times \dots$);
 - Z_m is the set of *natural numbers mod m* and the elements of Z_m are the classes of *equivalence* of **congruent integers**;
 - Z_m^* is the set of *natural numbers mod m* that are **coprime** to m , the **multiplicative group** of Z_m ;

- $\phi(m)$ is the **Euler's totient function** and it's equal to $= |Z_m^*|$;
- The **Euler Theorem** says that for all a in Z_m^* , $a^{\phi(m)} = 1 \cdot \text{mod } m$ so we have:
 $a^{k \cdot \phi(m)+1} = a \times \text{mod } m$, $k \geq 0$;
 - * We can also extend it to Z_m where $m = pq$ and p, q are **prime number** and we have:
 $a^{k \cdot \phi(m)+1} = a \times \text{mod } m$, $k \geq 0$;
- *Quindi in poche parole, Z_m è il set dei numeri naturali congruenti in modulo ai numeri che vanno da 0 a $m - 1$, Z_m^* è il subset di Z_m considerando solo i numeri coprimi con m (due numeri sono coprimi se non hanno nessun divisore comune apparte 1), la funzione toziente di eulero è la funzione che restituisce il numero di interi coprimi tra 1 e il numero m tipo $\phi(8) = 4$ perchè 1, 3, 5, 7 sono coprimi);*
- Let G be a **group** and a an element of order n , the set: $\langle a \rangle = \{1, a, \dots, a^{n-1}\}$ is a **sub-group** of G , a is called the **generator** of $\langle a \rangle$ (it's the number from which we can generate all the elements of the subgroup). If G is *generated* by a then G is called **Cyclic**, and a is the **primitive element** of G .
 - The *theorem* says that for any *prime number* p the **multiplicative group** of Z_p^* is cyclic;
- A set F with two *binary operations* $+$ *addition*, and \times (or \cdot) *multiplication* is called a commutative **ring** with identity if:

$1 \quad \forall a,b \in F, a+b \in F$ $2 \quad \forall a,b,c \in F, (a+b)+c=a+(b+c)$ $3 \quad \forall a,b \in F, a+b=b+a$ $4 \quad \exists 0 \in F, \forall a \in F, a+0=a$ $5 \quad \forall a \in F, \exists -a \in F, a+(-a)=0$	$6 \quad \forall a,b \in F, a \cdot b \in F$ $7 \quad \forall a,b,c \in F, (a \cdot b) \cdot c=a \cdot (b \cdot c)$ $8 \quad \forall a,b \in F, a \cdot b=b \cdot a$ $9 \quad \exists 1 \in F, \forall a \in F, a \cdot 1=a$ $10 \quad \forall a,b,c \in F, a \cdot (b+c)=a \cdot b+a \cdot c$
--	--

- A set F with two *binary operations* $+$ *addition*, and \times (or \cdot) *multiplication* is called a commutative **field** if:

$1 \quad \forall a,b \in F, a+b \in F$ $2 \quad \forall a,b,c \in F, (a+b)+c=a+(b+c)$ $3 \quad \forall a,b \in F, a+b=b+a$ $4 \quad \exists 0 \in F, \forall a \in F, a+0=a$ $5 \quad \forall a \in F, \exists -a \in F, a+(-a)=0$	$6 \quad \forall a,b \in F, a \cdot b \in F$ $7 \quad \forall a,b,c \in F, (a \cdot b) \cdot c=a \cdot (b \cdot c)$ $8 \quad \forall a,b \in F, a \cdot b=b \cdot a$ $9 \quad \exists 1 \in F, \forall a \in F, a \cdot 1=a$ $10 \quad \forall a,b,c \in F, a \cdot (b+c)=a \cdot b+a \cdot c$
$11 \quad \forall a \neq 0 \in F, \exists a^{-1} \in F, a \cdot a^{-1}=1$	

- A **field** is a *commutative ring* with *identity* where each non-zero element has a *multiplicative inverse*. $(F, +)$ is a **commutative (additive) group** and $(F \setminus \{0\}, \cdot)$ is a **commutative (multiplicative) group** (with \cdot distributive over $+$)

- Let $f(x) = a_0 + a_1 \cdot x + \dots + a_n \cdot x^n$ be a **polynomial** of degree n in one variable x over a field F , the **theorem** says that $f(x) = 0$ has almost n solution in F . It's important to note that this theorem doesn't hold over *rings with identity*.
 - Let $f(x) = a_0 + a_1 \cdot x + \dots + a_n \cdot x^n$ and $g(x) = b_0 + b_1 \cdot x + \dots + b_m \cdot x^m$ two *polynomials* over F such that $m \leq n$. There is a *unique polynomial* $r(x)$ of degree $< m$ such that: $f(x) = h(x) \cdot g(x) + r(x)$ where $r(x)$ is called the **remainder** of $f(x)$ **modulo** $g(x)$;
- A field $(F, +, \cdot)$ is called a **finite field** if the set F is *finite*.
 - The *theorem* says that for every **prime power** p^k with $k = 1, 2, \dots$ there is a **unique finite field** containing p^k elements. These fields are called **Galois Fields** and are denoted with $GF(p^k)$. There aren't *finite fields* with other *cardinalities*.
 - *Polynomial equations* and *factorizations* in *finite fields* can be different than over the *rationals*;
 - A *polynomial* is **irreducible** in $GF(p)$ if it doesn't *factor* over $GF(p)$, otherwise is *reducible*.
- For a *theorem*: let $f(x)$ be an **irreducible polynomial** of degree k over Z_p , the *finite field* $GF(p^k)$ can be realized as the set of **degree $k - 1$ polynomials** over Z_p , with *addition* and *multiplication* done *modulo* $f(x)$;
 - By the *theorem* the **finite field** $GF(2^5)$ can be realized as the set of *degree 4 polynomials* over Z_2 with *addition* and *multiplication* done *modulo* the *irreducible polynomial* $f(x) = x^5 + x^4 + x^3 + x + 1$;
 - The *coefficients* of *polynomials* over Z'_2 are 0 or 1, so a *degree k* can be written down by $k + 1$ *bits* so for example with $k = 4$:
 - * $x^3 + x + 1 \rightarrow (0, 1, 0, 1, 1)$;
 - * $x^4 + x^3 + x + 1 \rightarrow (1, 1, 0, 1, 1)$;
 - The *addition* can be done through **XOR** since $1 + 1 = 0$:

$$\begin{array}{rcc}
 x^4 + x^3 + x & (1,1,0,1,0) & + \\
 x^3 + x + 1 & (0,1,0,1,1) & = \\
 \hline
 x^4 & (1,0,0,0,1)
 \end{array}$$

- The **multiplication** is done by a *Polynomial multiplication*, and then remainder *modulo* the defining *polynomial*. For small size *finite field*, the *lookup table* is the most efficient method for implementing *multiplication*.

Now finally the **AES**, that is a *symmetric block cipher* with *key lengths* of 128, 192 or 256 bit, and it's *resistance* to all known *attacks*. It's *simple* and it's *fast*, so it's very good for devices with limited *computing power*. When we have *Input* and *Output block length* of 128 bits the *State* of 128 bits we can arrange it as a **4-by-4 matrix of bytes**. 128 since we have 16 elements in the *matrix* and each element is in byte that is formed by 8 bits so $(16 \cdot 8 = 128)$.

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

When we have a *key length* of 128, 196, 256 bits the *Cipher Key Layout* that is arranged in a 4-by- $\frac{n}{32}$ matrix of *bytes* (where $n = 128, 196, 256$ bit):

$K_{0,0}$	$K_{0,1}$	$K_{0,2}$	$K_{0,3}$	$K_{0,4}$	$K_{0,5}$
$K_{1,0}$	$K_{1,1}$	$K_{1,2}$	$K_{1,3}$	$K_{1,4}$	$K_{1,5}$
$K_{2,0}$	$K_{2,1}$	$K_{2,2}$	$K_{2,3}$	$K_{2,4}$	$K_{2,5}$
$K_{3,0}$	$K_{3,1}$	$K_{3,2}$	$K_{3,3}$	$K_{3,4}$	$K_{3,5}$

Initial layout: $K_{0,0}, K_{1,0}, K_{2,0}, K_{3,0}, K_{0,1}, \dots$

The *algorithm* at **high level** (*without explanations in details that we will see later*) is:

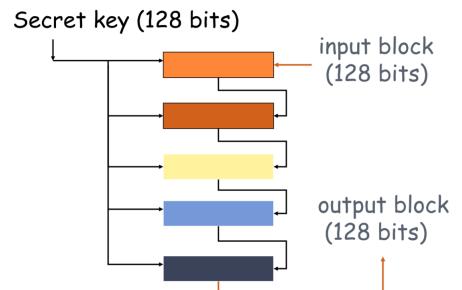
Algorithm High level AES

```

AES(State, Key);
KeyExpansion(Key, ExpandKey);
AddRoundKey(Key, ExpandKey [0]);
for (i = 1, i < R, i + +) do:
    Round(State, ExpandKey [i]);
FinalRound(State, ExpandKey [R]);

```

So, this *code* is repeated for each *block* of the *plain text* (each *block* is of 128 bits), so we start with a *key* of 128 bit, but before we start the *rounds* (that are 10) since we don't want to use the same *key* of 128 bits for all the different *blocks*, we need to expand the *key* in such a way for each *block* we have a different *key* of 128 bits. The **encryption** is done by following this scheme:



We can see that every *state* is done by the **encryption** of the *precedent round* with the *key extended* of the *current block*. 128 bits AES uses 10 *rounds*, in which the *secret key* of 128 bits is expanded to 10 *round keys* of 128 bits each. Each *round* changes the **state**, then we **XOR** the *round key*, if we have *longer keys* we add one *round* for every extra 32 bits. Now we will see in details what happen in each single *round*, we start from the assumption that the *plain text* that we are going to *encrypt* is divided in *state* of 128 bit and each *state* can be divided in a matrix 4×4 . We will **transform the state** by applying:

1. **Substitution;**
2. **Shift rows;**
3. **Mix columns;**
4. **XOR round key;**

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

The **Substitution** operates on every *byte* separately: $A_{i,j} \leftarrow A_{i,j}^{-1}$, so we apply a *transformation* in place in every single *block*, the operation is a *multiplicative inverse* which is highly *non-linear*. If $A_{i,j} = 0$ then we don't change $A_{i,j}$. It's important to note that the **substitution is invertible**.

We will **Shift** each element of each *row* to the **right**, 0 for the *first row*, 1 for the *second row*, 2 for the *third row*, 3 for the *fourth row*. This shift is invertible.

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	no shift
$A_{1,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	shift 1 position
$A_{2,2}$	$A_{2,3}$	$A_{2,0}$	$A_{2,1}$	shift 2 positions
$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,0}$	shift 3 positions

In the **Mix Columns** operation every state *column* is considered as a *Polynomial* over $GF(2^8)$, we will multiply with an *invertible polynomial* $03x^3 + 01x^2 + 01x + 02 \pmod{x^4 + 1}$ the *Inv* instead is $0Bx^3 + 0Dx^2 + 09x + 0E$.

The **Key Expansion** instead will generate a different *key* per *round*, and need a 4×4 matrix of values per *round*. It's based upon a *non-linear transformation* of the *original key*.

Breaking 1 or 2 *rounds* of the **AES** is easy, actually is not known how to break 5 *rounds*, and breaking the full 10 *rounds* efficiently is considered impossible.

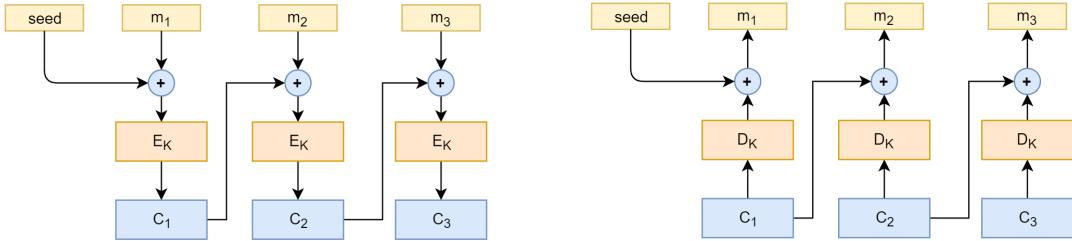
All *block ciphers* operate on *blocks* of *fixed length* cause *message* can be of any *length* and because **encrypting** the same *plaintext* under the same *key* always produces the same *output*. Several modes of *operation* have been invented which allow *block ciphers* to provide **confidentiality** for *messages of arbitrary length*.

2.2.2 ECB

ECB or also called **Electronic Code Book** is an stream cipher in which we *encrypt* each *plaintext block* separately, and it's very simple and efficient, and also it's possible to implement it in a *parallel way*. It doesn't conceal **plaintext patterns**, and it's possible to attack it with *active attacks*, in fact *plaintext* can be manipulated by removing, repeating or interchanging *blocks*.

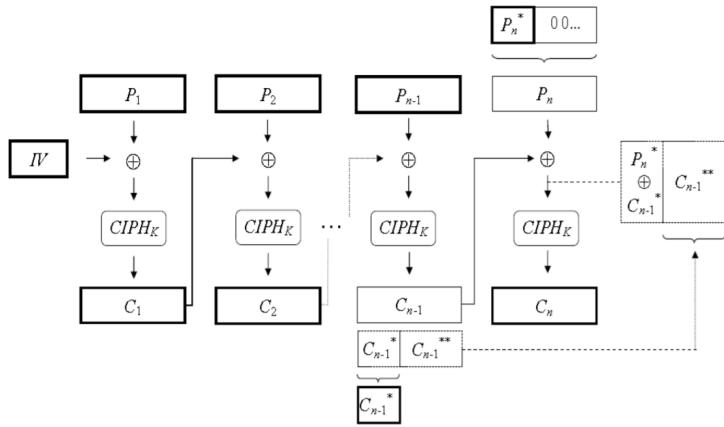
2.2.3 CBC

CBC is the **Cipher Block Chaining**, created by *IBM* in the 1976, is an *asynchronous stream cipher*, in which *errors* in one *ciphertext block* will propagate, that conceals *plaintext patterns* and there aren't *parallel implementation*. *Plaintext* cannot be easily manipulated and it's a standard in a lot of systems. In this cipher the previous *ciphertext* is **XORed** with current *plaintext* before *encrypting* current *block*. We use a *seed* to start the process, and this *seed* can be sent without *encryption*. If we set *seed* = 0, in most case is safe, but a *random seed* is better.



The *CBC* can lead to *problems*, in fact if there is a **transmission error** that changes just one bit of the C_{i-1} then block m_i changes in a *predictable way* (and this is be used by an *adversary*), but there are *unpredictable changes* in m_{i-1} , so the *solution* is to always use an **error detecting code**, that check the quality of the *transmission*. The *message* of the *CBC* must be **padded** to a multiple of the *cipher block size*, and in order to handle this issue we need to use the **Ciphertext Stealing**. A *plaintext* can be *recovered* from just two adjacent *blocks* of *ciphertext*, so as a consequence *decryption* can be **parallelized**, in fact usually a *message* is encrypted once, but *decryption* is done many times.

Ciphertext Stealing is a general method that allows for processing of *messages* that are not evenly divisible into *blocks*, so it doesn't result in an expansion of the *ciphertext*, it cost only in *complexity*. It consists of altering processing of the last two *blocks* of *plaintext*, so it re-order the *transmission* of the last two *blocks* of *ciphertext* and this method is suitable for *ECB* and *CBC*.



We can see that the last *plaintext block* is not divisible for 128 (*plaintext block size*, our *block* is made by n bits) so we fill the *block* with all 0, the **XOR** will be done only by the first n bits of the *previous ciphertext*. So during the *decryption* the *block* will be $[P_n^* \oplus C_{n-1}^* | C_{n-1}^{**}]$ so in order to get the original *plaintext* the result will be XORed with C_{n-1} in this way we obtain:

$$[P_n^* \oplus C_{n-1}^* | C_{n-1}^{**}] \oplus [C_{n-1}^* | C_{n-1}^{**}] = [P_n^* | 000000\dots]$$

- **Encryption:**

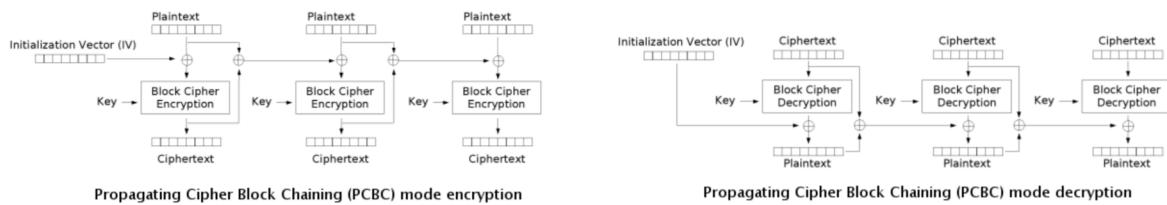
- If the *plaintext* length is not a multiple of the *block size*, we will pad it with enough zero until it is.
- We *encrypt* the *plaintext* using the *Cipher Block Chaining mode*;
- We swap the last two *ciphertext blocks* (? *swap?* where? in the figure there isn't any *swap*);
- We truncate the *ciphertext* to the length of the original *plaintext*;

- **Decryption:**

- If the *ciphertext* length is not a multiple of the *block size*, (like n bits short), then we pad the it with the last n bits of the *block cipher decryption* of the last full *ciphertext block*;
- Swap the last two *ciphertext blocks*;
- *Decrypt* the *ciphertext* using the *Cipher Block Chaining mode*;
- We truncate the *plaintext* to the length of the original *ciphertext*;

2.2.4 PCBC

PCBC is designed to extend or propagate a *single bit error* both in *encryption* and *decryption* (before if there was a *bit error* it will propagate to change all the *message* here only a *bit*). Here we have that the current *plaintext* (i) before *encryption* is XORed with: $\text{plaintext}_{i-1} \oplus \text{ciphertext}_{i-1}$. In the *decryption* will happen the same procedure, but the *XOR* will happen after the *decryption*, so since the *decryption* need the *plaintext* of the precedent *block* is not possible anymore to do it in *parallel* like in the *CBC*.



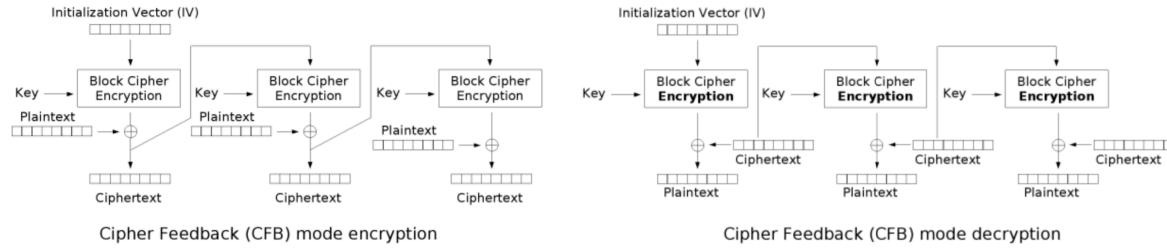
On a *message encrypted in PCBC mode*, if two adjacent *ciphertext blocks* are exchanged, this doesn't affect the *decryption of subsequent blocks* (this doesn't happen to *CBC*):

- $I_{i+2} = C_{i+1} \oplus (D_k(C_{i+1}) \oplus I_{i+1})$;

- $I_{i+1} = C_i \oplus (D_k(C_i) \oplus I_i)$;
- $I_{i+2} = C_{i+1} \oplus (D_k(C_{i+1}) \oplus C_i \oplus D_k(C_i) \oplus I_i)$

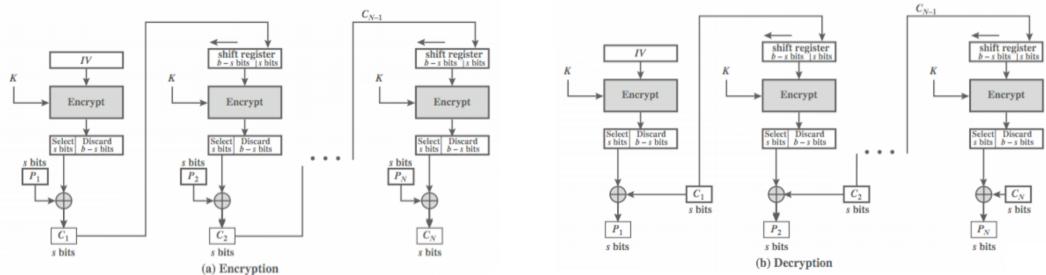
2.2.5 CFB

CFB is similar to **CBC** but makes a *block cipher* into an **asynchronous stream cipher**, so it supports some *re-synchronizing* after error if the input to the encryptor is given by through a **shift-register**. In the *encryptor* the *plaintext* will be *XORed* after applying the *encrypt* (in the *CBC* we do it before). Since the *XOR* is done after *encryption* we don't need anymore to pad last *block* with 0 when the *plaintext size* is not a multiple of the *block size*:



The *decryption* can be made in *parallel* since in order to obtain the *plaintext* I only need the *current block* and the *precedent ciphertext*. If there is an error in the output of the *encryption algorithm* this error will not be propagated in all the other *blocks*, since it will only infect the current *XOR* and the *decryption* of the next *block*. So **CFB** shares two advantages over *CBC*, the *block cipher* is only ever used in the *encryption direction* and the *message* doesn't need to be *padded* to a multiple of the *cipher block size*.

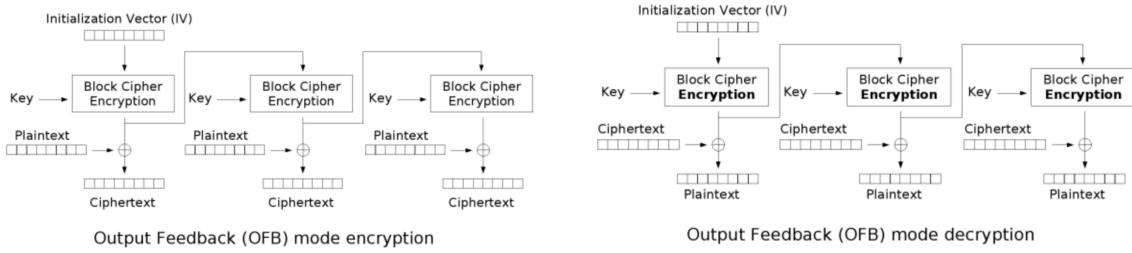
When we want to use a **CFB with shift register**, we start by initializing a *shift register* the size of the *block size* with the *initialization vector*, this will be *encrypted* with the *block cipher* and the highest s bits of the result are *XORed* with s bits of the *plaintext* to produce s bits of *ciphertext*, that then we be *shifted* in the *register* and this *process* will be repeated for the next s bits of *plaintext*. The *decryption* is similar, we start with the *initialization vector* we *encrypt* and *XOR* the high bits of the result with s bits of the *ciphertext* to produce s bits of *plaintext*, then we *shift* the s bits of the *ciphertext* into the *shift register* and *encrypt* again:



$$s = 1, 8, 64 \text{ or } 128$$

2.2.6 OFB

OFB or *Output FeedBack* makes a *block cipher* into a **synchronous stream cipher**, it generates *keystream blocks* which are then *XORed* with the *plaintext blocks* in order to get the *ciphertext*, similar to *CFB* the *plaintext* is *XORed* after the *encryption*. Flipping a *bit* in the *ciphertext* produces a *flipped bit* in the *plaintext* at the same location, and this property allows many error correcting codes to function normally even when applied before *encryption*. Contrarily to *CFB* in output in the second *block* I will not give the *XORed* with the *plaintext* but I will pass directly the *encrypted output* of the algorithm, in this way if in the *CFB* two *blocks* of *plaintext* are influenced by an *error*, in *OFB* only the *current block* is influenced by the *error*.



Since we have *symmetry of XOR operation* **encryption** and **decryption** are exactly the same:

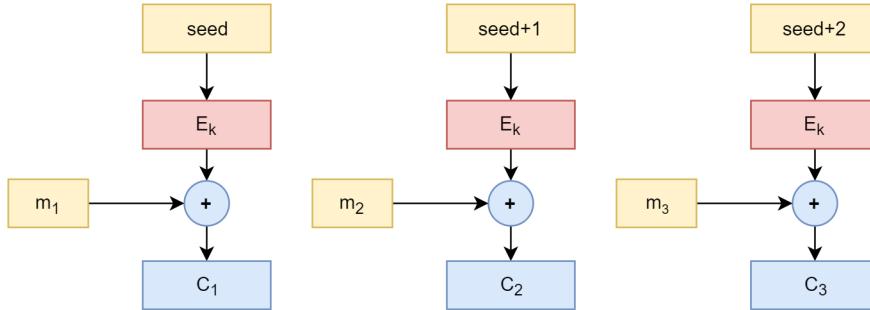
- $C_i = P_i \oplus O_i;$
- $P_i = C_i \oplus O_i;$
- $O_i = E_k(O_{i-1};$
- $O_0 = IV;$

If E_k is public, we need to **encrypt** the initial *seed*. If E is a *cryptography function* that depends on a *secret key* then the *seed* can be in clear. The *initial seed* must be modified for every new *message*, in fact if the *adversary* knows a *pair message, initial seed* then he can encode every *message*. So with **OFB** we have:

- *Synchronous stream cipher;*
- *Errors in ciphertext do not propagate;*
- *Pre-processing* is possible;
- *Conceals plaintext patterns;*
- *No parallel implementation known;*
- *Active attacks* by manipulating *plaintext* are possible;

2.2.7 CTR

CTR or *Integer Counter Mode* turns a *block cipher* into a *stream cipher*, in fact it generates the next *keystream block* by *encrypting* successive values of a *counter*. This *counter* can be any *function* which produces a *sequence* which is guaranteed not to repeat for a long time. It has similar characteristics to *OFB* but it also allows a *random access property* during *decryption*, so it's well suited to operation on a *multi-processor machine* where *blocks* can be *encrypted in parallel*. The generation of each *ciphered block* is independent from any other *blocks* since we use a different *seed* for each *block*.



Of course we have problems if we repeat the *seed* like *OFB*. When we use *CTR* we can *decrypt* the *message* starting from *block i* for any *i*, since we don't need to *decrypt* from the first *block*.

2.2.8 Initialization Vector

Most *modes* (except *ECB*) require an **Initialization vector**, or **IV**, that is a sort of *dummy block* used to kick off the *process* for the first real *block*. It doesn't need to be secret in most case but it's important that it is never reused with the same *key*, in fact, in *CBC* and *CFB*, reusing *IV* leaks some information about the first *block* of *plaintext*. In *CBC* mode the *IV* must be unpredictable in *encryption time*. For *OFB* and *CTR* reusing an *IV* completely destroys *security*.

2.2.9 Strengthening a cipher

There are two ways in order to **strengthening a cipher**:

- **Key Whitening:**
 - Consist of steps that combine the *data* with portions of the *key* (usually by using *XOR*) before the *first round* and after the *last round* of *encryption*;
- **Iterated Ciphers (Triple DES, 3-DES):**
 - In which the *plaintext* undergoes *encryption* repeatedly by *underlying cipher*, in which ideally we use a different *key* so we have for *triple cipher*:
 - * $C = E_{k_1}(E_{k_2}(E_{k_1}(P)))$ called *EEE mode*;
 - * $C = E_{k_1}(D_{k_2}(E_{k_1}(P)))$ called *EDE mode*;

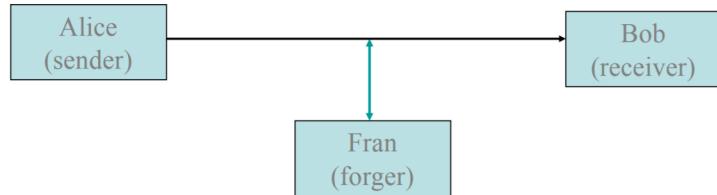
- So we use *three algorithms* of *encryption* or *two algorithm* of *encryption* and one of *decryption*. Sometimes only *two keys* are used in *3-DES*, and identical key must be at beginning and end.

Since the *goal* of the **adversary** it to find the *secret key*, the **double DES** cannot be used, in fact if we have a *double cipher* with two different *keys* the *adversary* can use the **meet-in-the-middle attack**. If fact if *plaintext/ciphertext* pairs are known, we have 2^n *encryption* and 2^n *decryption* so 2 keys of n bits. So it's possible to try all possible 2^n *encryptions* of the *plaintext* and all possible 2^n *decryptions* of the *ciphertext*. In other words, the presence of two different *keys* is represented as a *function* $h(x) = g(f(x))$, and the *middle attack* try to *map* the *co-domain* given the *domain* of the *function* f with the *counter-image* of the *co-domain* of the g function, so if initially you think to have a set of 2^{2n} keys in reality the attacker will make 2^n attempts, cause assuming that the *adversary* knows some *ciphertexts* C and some *plaintexts* P , the attacker can **in parallel** *encrypt* P and *decrypt* C trying to find some correspondences and if he find them he found the *key*.

It's even possible to use the *triple encrypting* with the **CBC**, and this can be done in two ways, by **external CBC**, in which we use a *triple encoding*, or **internal CBC**, in which we have an *encryption* after a *decryption* (with a different key) and another *encryption*.

3 Data Integrity & Authentication

The *goal* is to ensure the **integrity** of *messages* even in presence of an *active adversary* who sends own *messages*. It's important to note that *authentication* is orthogonal to *secrecy* (so they are independent between each other), so the *secrecy* of a *messages* doesn't not imply that the *sender* of the *message* is actually the one with who you think you are communicating.



3.1 MAC

The **authentication algorithm** is called A , the **verification algorithm** is called $V(\text{accept/reject})$, the **authentication key** is k , and the **message space** is usually a *binary strings*. Every *message* between Alice and Bob is a *pair*: $(m, A_k(m))$, where $A_k(m)$ is called the **authentication tag** of m . The *authentication algorithm* is called **MAC**, *Message Authentication Code*, in fact $A_k(m)$ is frequently denoted as $MAC_k(m)$ and the *verification* is done by executing *authentication* on m and comparing it with $MAC_k(m)$. The *security requirement* is that *adversary* can't construct a new **legal pair** $(m, MAC_k(m))$ even after seeing a $(m_i, MAC_k(m_i))$. The *output* should be as short as possible and the *MAC function* is not 1-to-1.

The **adversary** knows the *MAC algorithm*, knows *plaintext pairs* $(m, MAC_k(m))$, and knows *chosen plaintext* (unrealistic) so choose m get $MAC_k(m)$ and the **goal** is given n *legal pairs* find a **new**

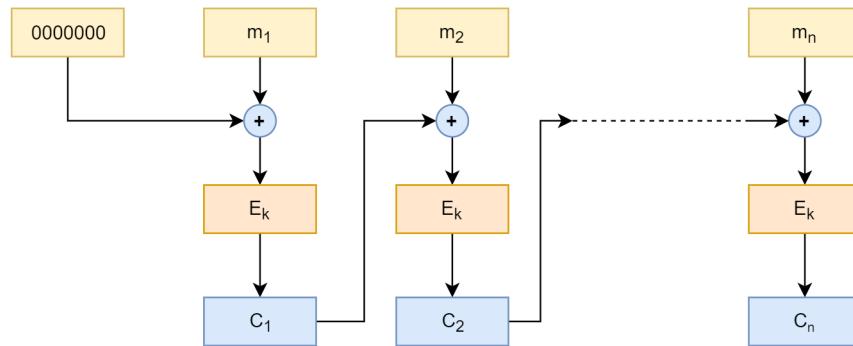
legal pair $(m, MAC_k(m))$ efficiently and with **non negligible probability** (so, *non sparato a cazzo*).

In practice **MAC** used are based on two types:

- **MAC based on CBC encryption**, so uses a block cipher (slow);
- **MAC based on cryptographic hash functions**, no restriction on export (fast);

3.2 MAC based on CBC

In CBC the previous ciphertext is XORed with current plaintext before encrypting the current block. With CBC with MAC's we start with the all zero seed, and given a message m consisting of n blocks M_1, M_2, \dots, M_n we apply CBC using a secret key k .



So this produces n *ciphertexts block* C_1, C_2, \dots, C_n and we decide that $MAC_k(m) = C_n$, so we take only the last *ciphertext*. This is a very *secure* way, but it's **very slow** since the *encryption* cannot be done in *parallel*. A **pseudo random function**, it's a function that looks random, and a good *encoding scheme* transforms the *message* in an apparently *random string*. If A_k is a *pseudo random function*, the **fixed length CBC MAC** is resilient to forgery. *CBC MAC* is secure for *fixed-length messages* but insecure for *variable-length messages*, in fact, if an *attacker* knows correct *message-tag pairs* (m, t) and (m', t') then can generate a *longer messages* m'' whose CBC-MAC will also be t :

- *XOR first block of m' with t and then concatenate m with the modified m' ;*
- $m'' = m | m'_1 \oplus t | m'_2 | \dots | m'_x$;

So in this way the *tuple* sent by the *attacker* will be accepted by the *receiver*. A solution in order to avoid this attack is to produce *ciphertext blocks* with a *different key*, not the same of the *key* of the *authentication tag*.

3.3 MAC based on hash functions

The **hash functions** are *functions* in which we map *large domains* to *smaller ranges*, used extensively for *searching* (*hashing tables*), we have the problem of *collisions* that are resolved with

chaining or *double hashing*, etc. The goal is to compute *MAC* of a *message* by using an *hash function* h for a *message* m with a *secret key* k , so *MAC* must be a *function* of the *key* and the *message*. For example $MAC_k(m) = h(k \parallel m)$.

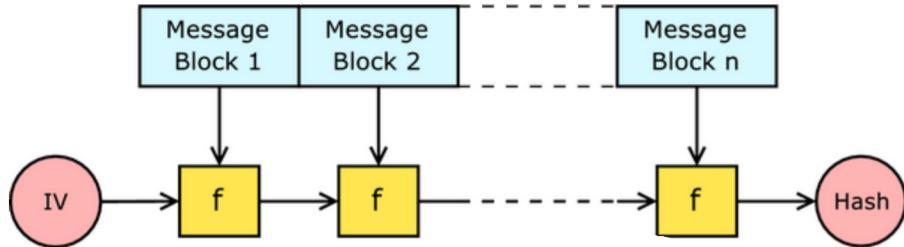
- An *hash function* $h : D \rightarrow R$ is called **weakly collision resistant** for $x \in D$ if it's hard to find $x' \neq x$ such that $h(x') = h(x)$;
- An *hash function* $h : D \rightarrow R$ is called **strongly collision resistant** if it's hard to find x, x' such that $x \neq x'$ but $h(x') = h(x)$;

The difference between the two definitions is that in the *weakly* the *element* is given, instead in the *strongly* we need to find two *elements*. It's important to note that *Strong \Rightarrow Weak* and the proof is:

- If I have an **non-weak** I can use an *algorithm* that takes x as *input* and returns x' that *collide* with x . I can also use an *algorithm* that finds both (x, x') that *collide*, so it's *not strong*;

The **birthday paradox** says: if 23 people are chosen at random the *probability* that two of them have the same birthday is > 0.5 . In other words, if $h : D \rightarrow R$ if we choose randomly $1.1774|R|^{1/2}$ elements of D the probability that two of them *collide* is > 0.5 . This means that if we have 2^n possible *message tags*, we have to try only $2^{n/2}$ *messages* to have a **collision probability** > 0.5 . The **Birthday Attack** uses this paradox, in fact given a *function* f find two different *input* x_1, x_2 such that $f(x_1) = f(x_2)$ with a 64-bit hash, would take at least 5.38×10^9 attempts to generate a *collision* by using *brute force*. This limit is called **birthday bound**.

Cryptographic hash function are *hash functions* that are *strongly collision resistant*, they don't use a *secret key*, usually they are defined by a *compression function* that maps n bits in m bits with $m < n$. We talk about **Merkle-Damgard construction**:



A mode of *operation* that returns a **digest** from a *document*. f is an *hashing function* that has a *domain* of n bits and a *codomain* of m bits. This means that the *message* has to be divided in *blocks*. In the figure we can see that the *block size* is $n - m$. To avoid *birthday attacks* we simply have to define a large **codomain size**, (more than 160 bits) in such a way *brute-force* is not feasible.

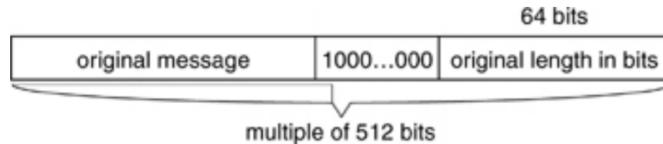
The **Keyed Hashing Functions** have the goals to combine *messages*, *keys* and *hash functions* in order to produce *MAC*. These provide both *data integrity* and *authentication*. We have to mix the *message* and the *key*, in a concatenation, and then use *hash-based algorithms*. We have different ways:

- $MAC_k(m) = h(k \parallel m)$, **insecure**, cause it suffers the *length-extension attack*, in which an *attacker* can add extra bits to the *message* and compute correct *MAC*;
- $MAC_k(m) = h(m \parallel k)$, **insecure**, cause if *attacker* finds a *collision* in the *hash function* (*birthday attack*) between two *messages*, then he will know two *messages* with the same *MAC* for all *keys*;
- $MAC_k(m) = h(k \parallel m \parallel k)$, **secure**;
- $MAC_k(m) = \text{first bits of } h(k \parallel m) \text{ or } h(m \parallel k)$, **secure**;

3.3.1 SHA-1

As any *hash function*, **SHA-1** produces a *digest* of fixed length (160 bit) starting from a *message* of variable length, that will be *padded*. There are 4 phases:

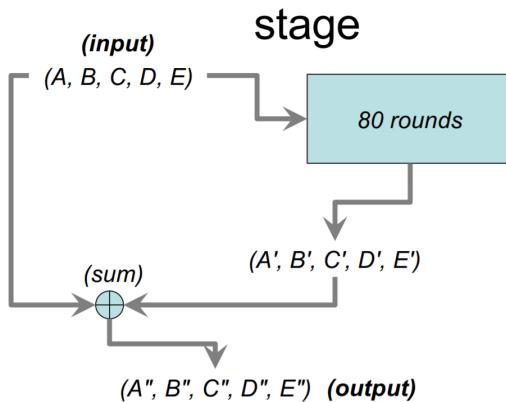
- **Padding:**
 - In this phase, *padding bytes* are added to the original *message* until it is congruent in bit to $448 \bmod 512$;
- **Adding length:**
 - To the *final message* (*original message* plus *padding*) we add an *unsigned integer* of 64 bits that denote the *full length* of the *message*, so we obtain a *message* that is multiple than 512 bit;



- **MD buffer initialization:**
 - A 160 bit *buffer* is subdivided in 5 *registers* of 32 bits each one that are:
 - * $A = 67452301$;
 - * $B = EFCDA89$;
 - * $C = 98BADC8E$;
 - * $D = 10325476$;
 - * $E = C3D2E1F0$;

- **Block Elaboration:**

– Now that we have the *5 registers* we can start computing *hashing*. The *complete message* is subdivided into several *blocks* of 512 bits. The core of the **SHA-1 algorithm** is the **compression function** which is composed by 4 *cycles* of 20 *steps* each one. Starting from the *first block* of 512 bits we give as *input* to the *compression function* the *block*, a *constant k*, and the values of the *five registers*. At the end of the *compression function* a new state of the *five registers* is observed, and we go on with the next *block*. When the *last block* of the *message* is compressed the *final values* of the *five register* will be summed to the initial values of the *five register* and the resulting values creates the **digest** that will be composed as $A|B|C|D|E$.



3.3.2 HMAC

It's a **Keyed Hashing MAC** which uses a pattern similar to the last one we listed. It's a *mode of operation* that we can use inside an *hash-algorithm*, so it's not an *hash-algorithm*. **HMAC** receives as input a *message m* a *key k* and an *hash function h* and then it computes:

$$HMAC_k(m, h) = h(k \oplus opad \parallel h(k \oplus ipad \parallel m))$$

Where *opad* is the **outer padding**: `0x5c5c...5c`, and *ipad*, **inner padding**: `0x3636...36` both long as a *block*.

We can note that if two *messages* collide in the *inner hash* $h(k \oplus ipad \parallel m)$ they will collide for the whole *hashing scheme*, cause the *outer hash* adds the same stuff to both *messages*. So, **HMAC** still suffers the **birthday paradox**, but the *adversary* cannot understand which are the *colliding pairs*, in fact, he is able to find two strings which *collide*, but since the *adversary* doesn't know the *key k* he cannot retrieve the *messages* of the strings.

3.3.3 AE

AE, or **Authenticated Encryption**, is a form of *encryption* which simultaneously assures the *confidentiality* and the *authenticity of data*, often is offered as single primitive in modern *API*, and it

makes *chosen-ciphertext attack* less dangerous, since the *attacker* is not able to choose a *ciphertext* and present it to the *decryption* in a proper way. There are three different approaches to *AE*:

- **Encrypt-then-MAC (EtM) :**

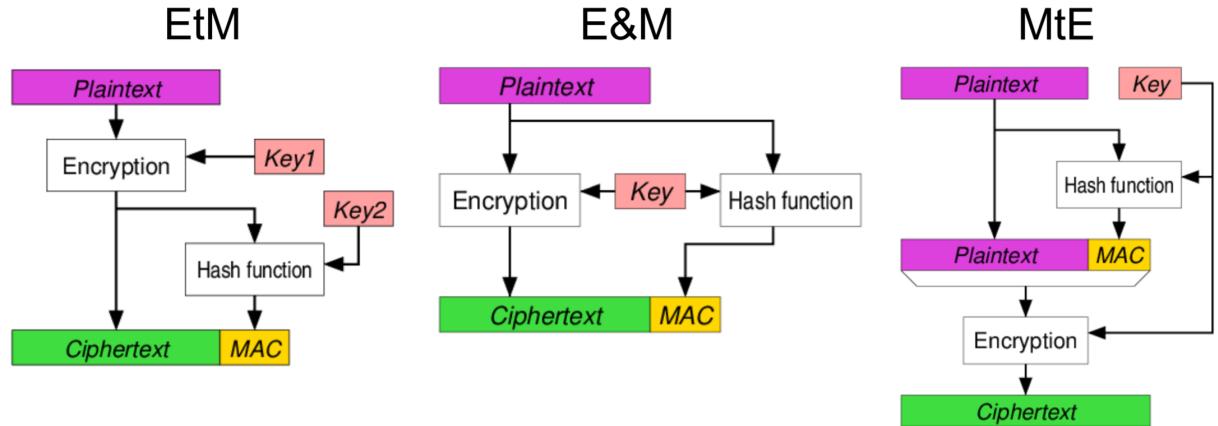
- The most secure today, in which we use two *different keys* k_1 and k_2 where the first one is used for the *encryption*, the second is used to find the *MAC* starting from the *encrypted message*;

- **Encrypt-and-MAC (E&M) :**

- *Safety* has not yet been proven, in which the *same key* is used for the *encryption* and in order to find the the *MAC*, starting from the *plaintext*;

- **MAC-then-Encrypt (MtE) :**

- Proven to be *secure* in specific setting, like in *E&M* we use only one *key*, but here first we generate the *MAC* starting from the *plaintext*, and after *plaintext* and *MAC* are *encrypted* together with the *key*;



4 Public Key Cryptography

We have seen a *model* in which Alice and Bob share the same *secret key* $k_{A,B}$, and it must be *secretly generated* and *exchanged* before using the *communication channel*. Then in 1976, *Diffie and Hellman* came up with a great idea: the **Public Key Infrastructure**. It's a model that uses **asymmetric encryption**, where each part has *two keys* K_E and K_D used for *encrypting* and for *decrypting*. They are *asymmetric*, this means that once we *encrypt* with a *key*, we must *decrypt* with the other *key*, and it's important to note that one of the *two keys* is *public* and the other is *private*. The implementation where *keys* are *asymmetric* it's hard and there are several ways:

- **One Way Function:**

- Are *functions* for which given the *input* it's **easy** to compute the *output* (in *polynomial time*), but given the *output* is **difficult** to retrieve the *input* that generated that *output*. An example of such a *function* is multiplication: given two numbers p, q compute $N = p \times q$ it's easy but having N it's hard to retrieve p and q , the mathematical definition is:
- A function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ is **one-way** if f can be computed in *polynomial time*, but for every *randomized polynomial time algorithm* A , and for every *polynomial* $p(n)$ with sufficiently large n :

$$\Pr[f(A(f(x))) = f(x)] < \frac{1}{p(n)};$$

- So the *function* must be **hard** to invert in the *average case*, rather than *worst-case* sens, and that the *probability* to find the correct x that returned $f(x)$ trough A , it's very small;
- A *very typical question in the exam is: SHA-1 is a one way function? The answer is yes, and we cannot use it to make encryption since it's not a one-to-one, since is not injective, if you don't know the key the ciphertext is hard to be inverted to find the plain text, you can brute force it but it's very hard;*

- **Discrete Log:**

- Let G be a *finite cyclic group* with n elements and g a generator of G . Let y be any element in G , then it can be written as $y = g^x$ for some integer x (for example Z_n of 14 is $\{1,3,5,9,11,13\}$ then the *generator* of this *group* is $g = 3$ cause from this value we get all the values of the *group*: $3 \bmod 14 = 3, 3^2 \bmod 14 = 9, \text{etc}$). Let $y = g^x$ and x the *minimal non negative integer* satisfying the equation, then x is called **discrete log** of y to base g ;
- Let $y = g^x \bmod p$ in Z_p^* (set of **positive integers** less than p and **coprime** to p):
 - * Given x calculating $y = g^x \bmod p$ it's **easy and computable** in $O(\log^3 P)$ steps;
 - * Instead given g, y and p it's believed to be **hard** to retrieve x , the **discrete log**;
- So retrieving the **Discrete Log** is believed to be a **One Way Function**.

4.1 Public Exchange of Keys

The **goal** is to have the possibility that two *parties*, *Alice and Bob*, who don't share any *secret information*, could perform a *protocol* in order to **derive** the same *shared key*, without exchanging

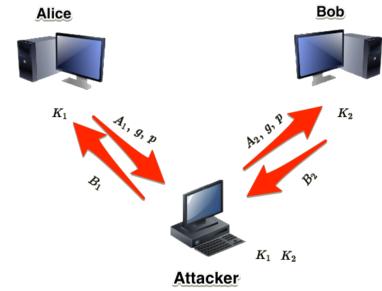
it in *clear*. Eve, the *attacker*, that is listening, cannot obtain the new *shared key* in *polynomial time* (unless $P = NP$ fuori di testa zi). This *protocol* was created by Diffie and Hellman, and the idea is to use a **session key**, that is a **shared secret key**, used for a *confidential conversation* for a certain period and then it's dropped. The protocol works as:

- The **public parameters** are: a *prime* p , a number g (possibly a *generator* of Z_p^*), and it works better if p is a **safe prime**, a *prime* number for which $p = 2q + 1$ where even q is *prime*;
- Alice chooses a value a at random $\in [1, p - 1]$, and sends $g^a \bmod p$ to Bob.
- Bob chooses a value b at random $\in [1, p - 1]$, and sends $g^b \bmod p$ to Alice.
- Both the parties computes the **shared key** $g^{ab} \bmod p$ in fact:
 - Alice that holds a computes $(g^b \bmod p)^a = (g^b)^a \bmod p = g^{ab} \bmod p$ (for the properties of *mod*);
 - Bob that holds b computes $(g^a \bmod p)^b = (g^a)^b \bmod p = g^{ab} \bmod p$ (for the properties of *mod*);
- So at the end they have the **same key**;

This can be applied even to *multiple parties*, of course the number of *exchanged message* will increase. Despite 25 years this is still **secure**. At the end of the session the *values* a and b are *discarded*, so the *Diffie-Hellman* achieves the **perfect forward secrecy**, that means that if an *attacker* discovers secret information about a *previous session* of *message exchanging*, he cannot be able to retrieve any information for the next session of *message exchanging*. The *computation time* is $O(\log^3 p)$.

Diffie-Hellman is effective only against a *passive adversary (sniffing)*, instead suffers the **Man-In-The-Middle attack**, which is *lethal* for this *protocol* and the attacker can control the whole conversation. The *attacker* makes *independent connections* with the victims, and relays *messages* between them, making them believe that they are talking directly to each other over a *private connection*.

Eve **intercepts** Alice's *public value* and sends her own *public value* to Bob. When Bob transmits his *public value*, Eve substitutes it with her own and sends it to Alice. Eve and Alice thus agree on one **shared key** and Eve and Bob agree on another **shared key**. After this *exchange*, Eve simply decrypts any *messages* sent out by Alice or Bob, and then *reads* and possibly *modifies* them before *re-encrypting* with the appropriate *key* and transmitting them to the other party. This *vulnerability* is present because **Diffie-Hellman key exchange** doesn't **authenticate** the participants.



In order to have a secure **DH protocol**, an **authentication system** must be added, in fact, most *cryptographic protocols* include some form of *endpoint authentication* to prevent this *attack*. *Key Exchange* is used by **VPN**, that are made by two *protocols*, an *Handshake Protocol*, in which we have a *key exchange* between parties sets *symmetric keys*, and a *Traffic Protocol*, in which *communication* is *encrypted* and *authentication* is made by *symmetric keys*.

5 RSA

RSA is the implementation of the idea of Diffie and Hellman on the Public Key Infrastructure, and these implementation need some mathematical properties:

5.1 Math considerations

- **Multiplicative Group Z_{pq}^* :**
 - Let p and q be two large *prime numbers*, and let's denote their *product* with $N = pq$, N it's not *prime*, but it's a **semiprime** (a product of two primes);
 - The **multiplicative group** $Z_{pq}^* = Z_N^*$, contains all *integers* in the range $[1, pq - 1]$ that are **coprime** to both p and q . It's **cardinality** it's not $pq - 1$ cause N it's not *prime*. There are $(p - 1)$ multiples of q and $(q - 1)$ multiples of p so the *cardinality* (the size of the group) is:

$$\phi(pq) = |Z_{pq}^*| = pq - 1 - (p - 1) - (q - 1) = pq - (p + q) + 1 = (p - 1)(q - 1);$$

- So for the **Euler theorem**, we have that for every $x \in Z_{pq}^*$, $x^{(p-1)(q-1)} = 1$;

- **Exponentiation in Z_{pq}^* :**

- We notice that not all the integers between $[1, pq - 1]$ are in Z_{pq}^* , this means that each element in that range doesn't have necessarily have a *unique inverse* in Z_{pq}^* . We want an e that is in the range $[1, (p - 1)(q - 1)]$, such that the power operation $x \rightarrow x^e$ is a **one-to-one operation** in Z_{pq}^* (*every element of the function's codomain is the image of at most one element of its domain*);
- Since an element in the range $[1, pq - 1]$ can have more than one inverse in Z_{pq}^* , then we cannot choose e at random;
- If we set e as a coprime to $(p - 1)(q - 1)$, then since the $\gcd(e, (p - 1)(q - 1)) = 1$ (*from the definition of coprime, gcd is mcd*), e has a **modular multiplicative inverse** called d of mod $(p - 1)(q - 1)$ (*a number for which $(e \cdot d) \bmod (p - 1)(q - 1) = 1$*);
- Then we have that $ed = 1 + C(p - 1)(q - 1)$ where C is a constant;
- Let $y = x^e$, then:

$$y^d = (x^e)^d = x^{1+C(p-1)(q-1)} = x^1 \cdot x^{C(p-1)(q-1)} = x(x^{(p-1)(q-1)})^C = x \cdot 1 = x;$$

- This means that the power $y \rightarrow y^d$ is the inverse of $x \rightarrow x^e$, so $x \rightarrow x^e$ is a **one-to-one operation**.

5.2 RSA Protocol

- Choose two *prime numbers* p and q and let $N = pq$ be their *product*;
- Choose an e such that $1 < e < \phi(N)$ and $\gcd(e, \phi(N)) = 1$ so e is *coprime* with $\phi(N)$;
- Let d such that $de \equiv 1 \bmod \phi(N)$, so d is the *modular multiplicative inverse*;

- Then the public key is (e, N) ;
- The private key is (d, N) ;
- We will encrypt $M \in Z_n$ by $C = E(M) = M^e \text{ mod } N$;
- We will decrypt $C \in Z_n$ by $M = D(C) = C^d \text{ mod } N$;
- Decryption is valid since:
 - If $ed = 1 \pmod{\phi}$ then there is an integer k that $ed = 1 + k\phi$, where $\phi = (p-1)(q-1)$;
 - If $\gcd(m, p) = 1$ then by **Fermat theorem** $m^{p-1} = 1 \pmod{p}$ so if we power both the sides for $k(q-1)$ we obtain: $m^{k(p-1)(q-1)} = 1^{k(q-1)} \pmod{p}$, ($1^{k(q-1)}$ does 1) and then we multiply both sides for m and we get: $m^{1+k(p-1)(q-1)} = m \pmod{p}$;
 - If instead $\gcd(m, p) = p$ then still equal cause both sides are congruent to 0 \pmod{p} cause $m = jp$ for some $j \geq 1$ so m is multiple of p ;
 - In both cases we have that $m^{ed} = m \pmod{p}$, similarly $m^{ed} = m \pmod{q}$;
 - This means that $m^{ed} = m \pmod{N}$
 - So this proof says, since m is *coprime* with p , and since m is *coprime* to q then it will be *coprime* even with N ;

RSA uses a *variable-length key*, usually 512 bits, and a *block size* also variable, but it's important that the **size of the plaintext is less than N** , $|plaintext| < N$, and that the $|ciphertext| = N$, it's slower than *DES*, and it's not used in practice for *encrypting long messages*, and it is used to *encrypt a secret key* (instead of a *message*);

An example of RSA: $p = 47, q = 59$ so $N = 2773$ and $\phi(N) = 46 \times 58 = 2668$, we pick a $e = 17$ and $d = 157$, cause $(17 \cdot 157 \pmod{2668} = 1)$, for $N = 2773$ we can encode two letters using a two digit number per letter: $A = 01, B = 02, \dots$

Since e and d are **symmetric**, since they are *bounded* by the *relation* $ed = 1 \pmod{\phi(N)}$, it doesn't matter which you choose first, but from a *security prospective* these values are not *symmetric*: in fact e is *public* (used for *encryption*), and d is *private* (used for *decryption*), so we need that d is a hard value to guess:

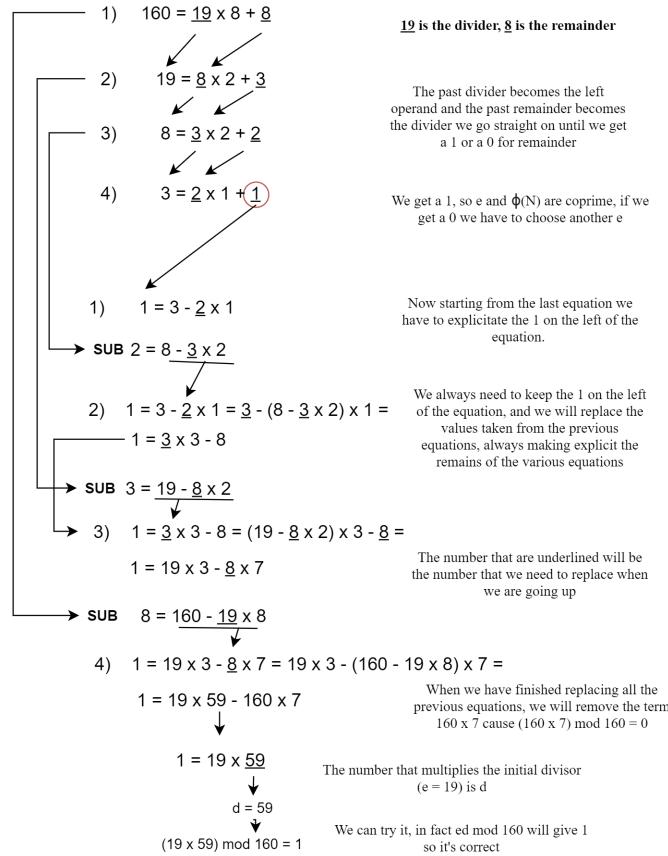
- If we choose e and then compute d , we can choose e to be a small value, and this is a real advantage, cause the *encryption* is much faster and without any loss of security;
- If we choose d and then compute e , we have to select a large value for d , and this slows down the operation;

In order to compute d we use the **Bezout's identity** and the **Extended Euclid's Algorithm**:

- $au + bv = d$ where $a > b$ and u, v are signed integers then there exists d that is the common greatest divisor of a and b ;
- If a and b are coprime, then $d = 1$, and u is the modular multiplicative inverse of a instead v is the modular multiplicative inverse of b ;

- The **Extended Euclid's Algorithm** uses this *identity* in order to compute the d starting by the e , and since the *math proof* is too incomprehensible I will just put an exercise to expose how it works:

- We have two *prime numbers* $p = 11$ and $q = 17$, so we have that $N = p \cdot q = 187$ and $\phi(N) = (p - 1)(q - 1) = 160$;
- Now have to choose an e that is **coprime** with $\phi(N)$ that is in range $[1, \phi(N)]$ (*usually we take a prime number, so we have sure that is coprime with $\phi(N)$*) in this case we choose $e = 19$;
- Now we know that $ed \equiv 1 \pmod{\phi(N)}$ so we have: $19 \cdot d \equiv 1 \pmod{160}$;
- We start using the **Extended Euclid's Algorithm**: in the equation at the left we put the $\phi(N)$ and at the right we will divide it for e plus the *remainder* so:



- If we obtain a d (that can be also negative) that is not in the range $[1, \phi(N)]$ we can sum the value of $\phi(N)$ (how many times you need). In fact it's easy to see, that also with this exercise if we sum $\phi(N) = 160$ to $d = 59$, we obtain $d_1 = 59 + 160 = 219$ and we compute $e \cdot d_1 \equiv 1 \pmod{160}$ it still gives 1 as result: $(19 \times 216) \pmod{160} = 1$, and this is also valid for $d_2 = 59 + 160 \times 2, d_3 = \dots$;

5.3 Attacks against RSA

RSA can be seen as a **one-way trapdoor function**, that is a *function* for which there is a *trapdoor* s such that:

- Without knowledge of s the *function* is a *one-way function*;
- Given s inverting the *function* is easy;

So in the **RSA** the **trapdoor** is of course d in fact given d going back from the *encryption* is very easy, instead without it is **very hard**.

5.3.1 Factorization

A first, trivial attack against *RSA* could be try to **factorize** N in order to get p and q from that $\phi(N)$ and then d . So it's recommended to take p, q large enough like 100 digits each, and to make sure that p and q are not too close together. (Since *prime number* are relatively frequent in $[N, 2N]$ and there are at least $\frac{N}{\ln N}$ *primes* in this range). It's important that $(p - 1)$ and $(q - 1)$ have large *prime factors* to foil **Pollard's rho algorithm**.

5.3.2 Weak Messages

There are weak message that are not good messages to be encrypted with RSA protocol and they are:

- $m = 0$ cause 0 to the power of anything is 0 and so the ciphertext will always be exactly equal to the plaintext;
- $m = 1$ for the same reason as above;
- $m = N - 1$ cause if the exponent is $e = 3$ then we have that $(N - 1)^3 \bmod N = (N - 1)^2(N - 1) \bmod N$, that can be written as $(N - 1)^2 \bmod N = 1 \bmod N$, but since $(n - 1)^2 \bmod n = 1$ then $(N - 1)^2(N - 1) \bmod N = (N - 1) \bmod N$. Therefore the *ciphertext* is equal to the *plaintext*, and this works for every odd $e \geq 3$. We need to use **salt** that is a sequence of *random bits*.
- If both m and e are *small* then we might have that $m^e < N$ and therefore we have $m^e \bmod N = m^e$ so the *adversary* can compute it and find m , the solution is to add *non-zero bytes* to avoid small *messages*.
- If we have a small e (suppose $e = 3$) and two *messages* are **similar**, let's suppose one is m and the other is $m + 1$ then the two ciphertexts are $C_1 = m^3 \bmod N$ and $C_2 = (m + 1)^3 \bmod N$ then it's easy to write a system of *equations* and we can get:

$$m = \frac{(C_2 + 2 \cdot C_1 - 1)}{C_2 - C_1 + 2}$$

So the *adversary* will be able to get the two *messages* easily, and this works for every *pair* m_1 and $m_2 = \alpha m_1 + \beta$. The solution is choose a large e .

5.3.3 Chinese Remainder Attack

The **Chinese Remainder Attack** is an *attack* in which an user send the same *message* to three different users with the same **public key** and we assume $e = 3$, let's say $(3, n_1), (3, n_2), (3, n_3)$. *Adversary* knows *public keys* and will know: $m^3 \bmod n_1$, $m^3 \bmod n_2$ and $m^3 \bmod n_3$. So he can compute $m^3 \bmod (n_1 \cdot n_2 \cdot n_3)$ by using the **Chinese Remainder Theorem**, so we have that $m < n_1, n_2, n_3$ and this means that $m^3 < n_1 \cdot n_2 \cdot n_3$ and therefore we have that: $m^3 \bmod (n_1 \cdot n_2 \cdot n_3) = m^3$. So the *adversary* will compute the *cubic root* and will get the result, the *solution* is to add *random bytes* to avoid equal *messages*.

5.3.4 Same N

Another *attack* is when two *users* choose the **same N**, this means that they have different p and q , so one of them can try to compute p and q starting from his e, d, N . Then the user could easily discover the *secret key* of the other user given his *public key*. The *solution* is that each person chooses his own N .

5.3.5 Multiplicative Property of RSA

Another *weak property* is that if a message M is the *product* of two other *messages* M_1 and M_2 then by the **invariance over multiplication** we have that:

$$M^e \bmod N = (M_1 \cdot M_2)^e \bmod N = (M_1^e \bmod N \cdot M_2^e \bmod N) \bmod N$$

So an *adversary* can proceed using **small messages**. The *adversary* wants to *decrypt* $C = M^e \bmod N$, so the *adversary* will compute $X = (C \cdot 2^e) \bmod N$. He will then use X as chosen *ciphertext* and will ask to the *oracle* (owner of the *private key*) to encrypt the new *message* $Y = X^d \bmod N$, but we have that:

$$X = (C \bmod N) \cdot (2^e \bmod N) = (M^e \bmod N) \cdot (2^e \bmod N) = (2M)^e \bmod N$$

But this means that the *adversary* will get $Y = 2M$

5.3.6 Chosen Ciphertext Attack

Similar to the precedent the *adversary* knows a **ciphertext** $C = M^e \bmod N$, and will **randomly choose** an X . Then will compute $C' = C \cdot X^e$ and will ask to the *oracle* to *decode* it. Then the *adversary* will compute $(C')^d = C^d \cdot (X^e)^d = M \cdot X \bmod N$. So the *adversary* since it does know X can compute the *plaintext* M . The *solution* is that the *oracle* should verify if the *message* incoming respect a given *structure*.

5.3.7 Chosen Plaintext Attack

Also called **CPA**, is an *attack model* which presumes that an *attacker* has the capability to choose an **arbitrary plaintexts** to be *encrypted* and obtain the corresponding *ciphertexts*. The goal is to gain further *information* which reduces *security* of *encryption scheme*. There are two forms of **CPA**, **Batch CPA** where all *plaintexts* are chosen before any of them are *encrypted*, **Adaptive CPA** where *subsequent plaintexts* are based on information from the previous *encryptions*.

5.4 RSA Standards

At the end of all this discussion, we got that **RSA** presents some *weaknesses*, and therefore a good *approach* that has to be followed when using RSA is the following:

- Choose good numbers p, q, e, d such that they satisfy the *properties* discussed so far;
- Before *encrypting message* M , perform some *preprocess* on M in order to obtain M' , (M' has not to change the meaning of M) and then perform the encryption of M' ;

For this reasons some **standards** have been invented in order to perform a good *RSA usage*.

5.4.1 Public-Key Cryptography Standard

Public-Key Cryptography Standard or **PKCS**, is a set of *standards* that specify how *RSA* should be used. We will be seeing only the **PKCS#1 standard** in which a message is pre-processed as:

$$m = 0 \parallel 2 \parallel \text{at least 8 non-zero bytes} \parallel 0 \parallel M$$

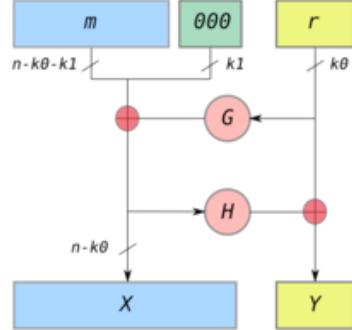
Where M is the *original message*. We set the *first byte* to 0 so we are sure that the *message* is less than N , the *second byte* equals to 2 denotes *encryption* of a *message* (if it is 1 it denotes *digital signature*). The *random bytes* imply that the same *message* sent to different people will result in different *ciphertext*.

5.4.2 Optimal Asymmetric Encryption Padding

OAEP is a *padding scheme* often used with RSA encryption that uses a pair of **random oracles** G and H to process the *plaintext* prior to *asymmetric encryption*. OAEP satisfy two goals:

- Add an element of *randomness* which can be used to convert a *deterministic encryption scheme* (the *traditional RSA*) in a *probabilistic scheme*;
- It prevent *partial decryption* of *ciphertexts* by ensuring that an *adversary* cannot recover any portion of the *plaintext* without being able to invert the *trapdoor*;

In the **encryption** we have that m is the *original message* a *string* of $(n - k_0 - k_1)$ bits, where k_0 and k_1 are two *integers* fixed by the protocol. G and H are two *cryptographic hash functions* and n it's the number of bits in *RSA modulus*. In order to **encode** the *original message* is *padded* with k_1 zeroes, and G will expand r (a random k_0 bit string) to $n - k_0$ bits, then the *XOR* is done between it and the *padded message* which results in X . Then H reduces the $n - k_0$ bits of X in k_0 bits that will be *XORed* with r in order to get Y . The *output* will be $X \parallel Y$ which will be then *encrypted* with *RSA*. In the **decryption** the *outcome* will be *decoded* in order to retrieve m in the following way: $r = Y \oplus H(X)$ and then $m|000\dots = X \oplus G(r)$ then we discard the k_1 zeroes and we obtain the *message*.



This scheme is also called **All-Or-Nothing security**, in fact, in order to retrieve m we must have X and Y , since we need X in order to retrieve r from Y and we need r in order to retrieve m from X .

5.4.3 El Gamal Encryption

El Gamal it's a mode of *encryption* strongly based on *Diffie-Hellman key exchange approach*, indeed we will see that the underlying math is the same. *El Gamal* is an alternative to *RSA* and so is able to generate a *pair of keys public/private*, in order to perform **encryption** and **decryption**. The protocol is the following: given a *prime* p and a *generator* of the Z_p *cyclic group* of order p , *Alice* chooses an x in the range $[1, p - 1]$ as the **private key**, and she chooses the number $g^x \bmod p$ as the **public key**. If *Bob* wants to send a *message* $m \in Z_p$ to *Alice*, he chooses a number y in the range $[1, p - 1]$ and he sends to *Alice* the pair: $(g^y \bmod p, m \cdot g^{xy} \bmod p)$. He can send the second *message* cause he knows m, y and $g^x \bmod p$ which is the *public key* of *Alice*. When *Alice* gets the *pair of messages*, she uses the first one in order to compute $g^{xy} \bmod p$ then she finds a number $(g^{xy})^{-1} \bmod p$ and so she can get the *message* as follows: $m = m \cdot g^{xy} \cdot (g^{xy})^{-1} \bmod p = m \bmod p = m$. This scheme requires two *exponentiations* per each *transmitted block*, so it require a little bit demanding in terms of *computational effort*.

6 Digital Signature - DSA

A **Digital signature** is a mathematical scheme for demonstrating the **authenticity** of *digital messages or documents*. A valid *digital signature* gives a recipient reason to believe that the *message* was created by a *known sender* (**authentication**), that the *sender* cannot deny having sent the *message* (**non-repudiation**), and that the *message* was not altered in transit (**integrity**). In few words, a *digital signature* provides *authentication, integrity* and *non-repudiation*. We already studied an approach for *data integrity*, it was based on *MACs*. Let's suppose that *Alice* and *Bob* share a secret key K , then $(M, MAC_K(M))$ will convince *Alice* that the *message* was created by *Bob*. But in case of dispute *A* cannot convince a judge that $(M, MAC_K(M))$ was sent by *Bob*, since *A* could generate it herself.

Then we discussed *RSA* and in general about the usage of **public and private keys**:

- **Encryption with other party's public key:** you get **confidentiality** (he *decrypts* with his *private key*);
- **Encryption with my private key: no confidentiality, but authentication** (everyone is enabled to *decrypt* my *ciphertext*, it is just sufficient to use my *public key* in order to do it, but in this way other *parties* can be sure about the fact that it was me who *encrypted* the *message*);

So, it's easy to **forge signatures** of *random message* even without holding D_A , in fact *Bob* can pick an R arbitrarily and can compute $S = E_A(R)$, this means that the pair (S, R) is a **valid signature** of *Alice* (even if it was created by *Bob*), so the scheme is subject to **Existential Forgery**, that is the ability of an *adversary* to create a *pair, message* and *signature*, where the *message* has not been signed in the past by the *legitimate signer* but the *adversary* doesn't have any control over the *message*. In the **Selective Forgery** instead, the *adversary* creates a pair *message signature*, where the *message* has been chosen by the *adversary* prior to the *attack*, if an *adversary* can do the *selective forgery* it can also do the *existential forgery*. In the **Universal Forgery** instead an *adversary* can create a *valid signature* for any given *message*, and this is the strongest ability in *forging* and it implies all the other types of *forgery*. So we need a method to prevent the *Existential*

Forgery and this method is the **hashing**.

An *adversary* can use the **forgery attack**, in fact he can generate a *random file* T and will *encrypt* it with the K_p (*public key of Bob*). The *ciphertext* generated will be $R = E_{K_p}(T)$ then the *attacker* sends R as *plaintext* and T as *digital signature*. Alice will take the *digital signature* T and so she *encrypts* T with K_p of *Bob* and she gets $E_{K_p}(T) = R$, but since the *attacker* sent R as *plaintext*, Alice sees that the *decryption* of the *digital signature* matches with the *plaintext* sent, so she accepts the *document*. That's why all this is not good.

Let E_A be *Alice public encryption key*, and D_A the *private decryption key*:

- In order to **sign** the *message* M , Alice will compute two *strings*: $y = H(M)$ and $z = D_A(y)$ and then will send to *Bob* (M, z) ;
- In order to **verify** the *Alice's signature*, *Bob* will compute the *string* $y = E_A(z)$ and will check if $y = H(M)$;

Alice create z with D_A (the *decrypt function*) cause it need to use its **private key** to *encrypt* it, instead *Bob* will read y using E_A (the *encrypt function*) to *decrypt* it with the **public key** of *Alice*. The function H should be **collision resistant** so that cannot find another M' with $H(M) = H(M')$. With **hashing** the process of *signing* is much faster, we are using *asymmetric encryption*, which is slow, but here it is performed on the *hash* of the document which is much smaller than the *plain document*.

6.1 Standards for Digital Signatures

The structure of the *general digital signature* is divided in three steps, the **generation of private and public keys** (in a *random way*), the **signing** (*deterministic or randomized*), and the **verification** (*accept or refuse*) usually *deterministic*. There are three different schemes used in practice: **RSA**, **El Gamal** and the **DSS**.

6.1.1 RSA and PKCS#1

We already saw the **PKCS#1 standard**, the *signature* is made by *hashing* using the *private key*, only the person who knows the *secret key* can *sign*, everybody can verify the *signature* by using the *public key*. We construct the *digital signature* by *encrypting* with our *private key* the following *message*:

$$0 \parallel 1 \parallel \text{at least 8 byte FF base 16} \parallel 0 \parallel \text{specs of hash function} \parallel \text{hash}(M)$$

Where:

- The first byte 0 ensures that the *message* is less than N ;
- The second byte 1 denotes *signature*, if it's 2 it means that the *message* is *encrypted* for *confidentiality* purpose;
- The bytes 111111..... implies that the *encoded message* is large;
- And M is the *original message*;

As a conclusion for **RSA** I want to stress the concept that when you are using *RSA* you want to *sign*, the other typical usage of *RSA* is send an **encrypted key**. You don't use *RSA* for *confidentiality* purpose, for *encrypt* a *document*, you *encrypt* just a *key*. Because of its *complexity, computational effort*, much bigger than Rijndael. PKCS#1 define a **Signature Scheme with Appending or SSA**, in which the *appendix* is the *signature* added to the *message*, and we don't consider *signature schemes with message recovery* (*message* is embedded in the *signature*). There are two approaches that differ in how the **encoded message** is obtained:

- **RSASSA-PSS**: a *probabilistic signature scheme* that uses **EMSA-PSS** an *encoding method* for *signature appendix*, inspired to *OAEP* that makes use of *random salt* added per *signature*, in which first the message is hashed, after the output is padded and we add a salt at the end so we generate a new message, and this message will be hashed and then XORed with another padding and salt that will be concatenated to $h(M)$;
- **RSASSA-PKCS1-v1.5**: that is *deterministic*;

6.1.2 El-Gamal Signature Scheme

El-Gamal scheme can be divided into three parts: *generation*, *signing* and *verification*. The **generation** of the *key* is made as:

- Choose a *prime* number p of 1024 bits such that the **Discrete Logarithm (DL)** in Z_p^* is *hard*, the only way to get the *private key* x is by computing the *discrete logarithm* of p in base g ;
- Find a **generator** g og Z_p^* , an integer for which the *powers* are congruent with the numbers *coprime* with p ;
- We pick a x in $[2, p - 2]$ at random;
- Compute $y = g^x \bmod p$;
- Then the public key $K_p = (p, g, y)$ and the private key $K_s = x$;

In order to **generate** the *signature* instead, if we want to sign a *message* M we define a new pair of *keys* (r, k) , where r is a *public key* instead k is a *private key* used in order to have a different *signature* every time:

- We will start by compute the *digest* $m = H(M)$;
- We will pick k in $[1, p - 2]$ coprime to $p - 1$ at random;
- After we will compute $r = g^k \bmod p$ and $s = (m - rx)k^{-1} \bmod (p - 1)$, if s is 0 then we need to restart;
- The *output signature* will be (r, s) ;

The **verification** follows this pattern:

- First we compute $m = H(M)$;
- We will accept only if $0 < r < p \wedge 0 < s < p - 1 \wedge (y^r \cdot r^s = g^m) \bmod p$;

- Since we have that $s = (m - rx)k^{-1} \text{ mod } (p - 1)$ we have that $sk + rx = m \text{ (mod } p\text{-1)}$ disappears cause mistero della fede). Now:
 - We know that $r = g^k \text{ mod } p$ so if we power left and right operands for s we obtain: $r^s = g^{ks} \text{ mod } p$;
 - We know that $y = g^x \text{ mod } p$ so if we power left and right operands for r we obtain: $y^r = g^{rx} \text{ mod } p$;
 - If we multiply we obtain: $y^r \cdot r^s = g^{ks+rx} \text{ mod } p = g^m \text{ mod } p$;

6.2 DSS, Digital Signature Standard

DSS is important because is the choice made by *NIST*, it is inspired on *El-Gamal approach* and according to the original formulation *DSS* is using *SHA*, just because the *hashing function* chosen by the *NIST* is *SHA*. There are two acronyms: **DSS, Digital Signature Standard** and **DSA , Digital Signature Algorithm**. The second is just an *algorithm* in order to get the *digital signature*.

The idea of **DSA** is to use two prime numbers p and q such that:

- p is a L bit prime number such that the Discrete Log problem $\text{mod } p$ is intractable;
- q is a 160 bit prime that divides $p - 1$, so we have that $p = j \cdot q + 1$, so j is a constant equal to $j = \frac{p-1}{q}$;

Then let α be a q^{th} root of $1 \text{ mod } p$, so we have that $\alpha^q = 1 \text{ mod } p$. In order to compute α we need to take a random number h such that $1 < h < p - 1$ and we compute $g = h^{\frac{p-1}{q}} \text{ mod } p = h^j \text{ mod } p$, since we need to apply the *little theorem of Fermat*, we need that h is *coprime* with p . If $g = 1$ we have to choose a different h for *security reason*. Then it holds that $g^q = h^{p-1} = 1 \text{ mod } p$ by the **Fermat Theorem**, since p is prime and every number less than p raised to $p - 1$ are equal to $1 \text{ mod } p$. Finally we choose $\alpha = g$.

Math proof: $g = h^j \text{ mod } p = h^{\frac{p-1}{q}} \text{ mod } p$ so if we power both the operators with q we have: $g^q = h^{p-1} \text{ mod } p$ and that from the Fermat Theorem $g^q = (1 \text{ mod } p) \text{ mod } p$ so we obtain that $g^q = 1 \text{ mod } p = \alpha^q$.

So now let's talk about the algorithm, we have to choose two prime numbers p, q such that $p - 1 = 0 \text{ mod } q$ (so $p - 1$ is a multiple of q) and we consider α as the q^{th} root of $1 \text{ mod } p$. Then the **private key** will be s a random integer $1 \leq s \leq q - 1$ and the **public key**: $(p, q, \alpha, y = \alpha^s \text{ mod } p)$. s will be hard to find since an *adversary* should compute the *discrete log*, and choosing a t that computing the discrete log of $\text{mod } p$ of the multiplicative group Z_p^* is hard. The *signature* is inspired to *El-Gamal*, in fact we choose a random integer *secret* in the range $[1, q - 1]$ and we have two parts:

- Part 1: $P_1 = (\alpha^k \text{ mod } p) \text{ mod } q$;
- Part 2: $P_2 = (SHA(M) + s \cdot P_1) \cdot k^{-1} \text{ mod } q$;

Also the **verification** is done in two parts:

- $e_1 = \text{SHA}(M)(P_2)^{-1} \bmod q$;

- $e_2 = P_1 \cdot (P_2)^{-1} \bmod q$;

And the *signature* will be accepted only if $(\alpha^{e_1} \cdot y^{e_2} \bmod p) \bmod q = P_1$. The *proof* of the *correctness* of this is:

Accept if $(\alpha^{e_1} y^{e_2} \bmod p) \bmod q = \text{PART I}$
 $e_1 = \text{SHA}(M) / (\text{PART II}) \bmod q$
 $e_2 = (\text{PART I}) / (\text{PART II}) \bmod q$

Proof : 1. definition of **PART I** and **PART II** implies
 $\text{SHA}(M) = (-s(\text{PART I}) + k(\text{PART II})) \bmod q$ hence
 $\text{SHA}(M)/(\text{PART II}) + s(\text{PART I})/(\text{PART II}) = k \bmod q$

2. Definit. of $y = \alpha^s \bmod p$ implies $\alpha^{e_1} y^{e_2} \bmod p = \alpha^{e_1} \alpha^{(s \cdot e_2)} \bmod p$
 $= \alpha^{\text{SHA}(M)/(\text{PART II}) + s(\text{PART I})/(\text{PART II}) \bmod q} \bmod p = \alpha^{(k+cq)} \bmod p$
 $= \alpha^k \bmod p$ (since $\alpha^q = 1$).

3. Execution of $\bmod q$ implies
 $(\alpha^{e_1} y^{e_2} \bmod p) \bmod q = (\alpha^k \bmod p) \bmod q = \text{PART I}$

So, starting from the *Part 2*, we isolate the $k \bmod q$, after from the definition of the y and from the *verify formula* we obtain $\alpha^{e_1}(\alpha^{s \cdot e_2} \bmod p) \bmod p$ that we can rewrite as $\alpha^{e_1+s \cdot e_2} \bmod p$. Now we will use the formula obtained before and we get $\alpha^k \bmod q \bmod p$ that is equal to $\alpha^k \bmod p$ cause k is a value less than q . Note that the external $\bmod q$ there was since the beginning of the proof but we omitted it. At the end: $(\alpha^{e_1} \cdot y^{e_2} \bmod p) \bmod q = (\alpha^k \bmod p) \bmod q = P_1$.

The **security of DSS** is ensured cause the *secret key* s is not revealed and it cannot be forged without knowing it, and we use a *random number* for *signing* (k) that is not revealed, and we don't have any *duplicates* of the same *signature* even with the same *message*. It's important to note that if k is known then you can compute $s \bmod q = s$, since s is less than q , and that we have two *signed messages* with the same k then k value can be revealed, so combined with the precedent equation it's possible to find s .

Finding two *primes* p and q such that $p - 1 = 0 \bmod q$ it's **not easy** and can take time, and p and q are *public* so they can be used by *many persons*. So, we have that *DSS* is faster than *RSA* for *signing* if we have some *preprocessing*, but the problem is to generate *random numbers* that may depend on the *message* so in this case we can't have *preprocessing*. *RSA* is usually used for *signature* and for *key management*.

When a *document* needs to be *signed* in many cases it's important for a formal point of view that also the the **time** is certified. This is called **Timestamping**, so we associate a *timestamp* to a *document*, in many cases we use a *third party* (an *authority*) also called **TSA** (**TimeStamping Authority**), who signs the *document* with the *timestamp*. If Alice want to *timestamp* a *document*:

- Alice compute *hash* of *document* and sends it to *TSA*;
- *TSA* will add the *timestamp*, will compute a new *hash* (of *timestamp* and *received hash*) and will *sign* the obtained *hash* and after will reply to Alice;
- Alice will keep the *TSA signature* as a proof, so everybody can check the *signature*, and *TSA* doesn't know *Alice's document*;

7 Authentication

Authentication is the procedure of providing the **identity** to a *party* in order to *access services* that this *party* provides to the *allowed ones*, in our example *Alice* needs to show her *identity* to *Bob*, cause *Bob* wants to show *information* only to *authorized people*. One of the most common *attacks* against *authentication* is **Spoofing**, that is the ability to cheat about the *identity* of the creator of the *message*, in fact with *authentication* we want to prevent *spoofing attacks*. We have two interesting cases of *authentication*:

- A *computer* is authenticating another *computer*;
- A *person* is using a *public workstation*, so we don't want to store *secrets* for all the possible *user*;

The **Closed World** assumption, it's an idea based on the sentence: **what is not currently known to be true, is false**. So it basically means that everything is *false* until we can prove that it's *true*. Many people don't like it because they prefers the concept of *black list* (forbidden or bad things) and *white list* (allowed things). Your *environment* is considered a **Closed Environment** when you are working in a *closed place*, like your enterprise. In many cases you are somewhat *dealing* with a **third-party** (a **trusted server**) that is protected by many methods, this *server* can support *authentication* of many *users*, we will see *Kerberos* that is a very *secure environment*. If the *attacker* belongs to the same company, it is called an **insider**, and it has a lot of power compared to an *external one*.

There are many differences between **authenticating a human and a computer**.

- **Computers** can store a *high-quality secret*, such as a *long random-looking number (key)*, and they can do *cryptographic operations*;
- **Humans** instead are not able to do this, they use some *passwords* that are very different from a *key*;

The first difference between a *password* and a *key*, is that *humans* are stupid, so they use *password* that contains *words*, and *words* belongs to a *dictionary*, so an *adversary* can run an attack called **Dictionary Attack**, a *probabilistic attack* that use most common *words* in order to find a *password*. Also *password* are usually *short* and *easy to be guessed* by an *attacker*. Instead *computers* stores **long keys**, usually *hidden* or *encrypted* or also *one-time password*. One of the most common *attack* is the **Login Trojan Horse**, in which an *attacker* prompts a *fake login window* that induces the innocent user to prompt the *password* that is collected by the *attacker*.

The **authentication of people** can be based on *password*, *smart card*, *biometric tools* or *network address*. *Biometrics* are usually used, like *retina examination*, *fingerprinting reader* or *voice*, but we can have *false positive* and *false negative* (like *fake fingerprints*, or *modified voices*).

There are three main approaches in order to achieve **authentication** and they are:

- **Shared secret key**: *Alice* and *Bob* share a *secret key* and use it to *authenticate* each other;
- **Authentication through a Third Party**: there is an *external server* and if two *users* want to authenticate each other they have to pass through the *server*;

- **Public Key Cryptography:** users use the *Public Key Infrastructure* in order to achieve *authentication*;

We can use different *techniques* in order to *authenticate* an *user* through a *key*, like **Timestamp**, or a **Nonce**, a random number chosen by the *user* that *authenticates* to verify the other knows the *key*, or **Sequence Numbers**, like in *TCP* that uses *sequence numbers* in order to do some things like tracking the order of *connections*. It's very important that we need to guarantee *authentication* and *integrity* that **Encryption** doesn't guarantee **Authenticity of a message**.

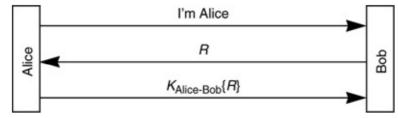
7.1 Authentication by Symmetric Key

We denote with K_{AB} the *secret key* shared between A and B , and with $K\{M\}$ we denote that M is *encrypted* with the *secret key* K , with R we denote the *Nonce*, but the challenge can be called R or K_{AB} . There are several ways to obtain **authentication** through **symmetric keys**:

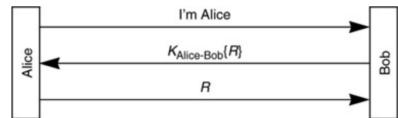
- One Way authentication using Nonce (Challenge);
- One Way authentication using Timestamps;
- Two Ways (mutual) authentication using Nonce;

7.1.1 Challenge/Response Authentication

Alice want to **authenticate** *Bob*, she will send to him a *plaintext*, presenting herself, *Bob* will respond with a **challenge** (that is R , the *nonce*), then *Alice* will reply with the *encrypted version* of R with the *key* $K_{AB}\{R\}$ (the correct answer of the *challenge*). If *Bob* receives the right *encryption* then he authenticates *Alice*. This scheme is **not mutual** (in fact *Bob* authenticates *Alice*, but *Alice* doesn't). This scheme can be *easily attacked*, cause *Alice* cannot be sure that she is *authenticating* actually to *Bob*, so maybe she talking with the *attacker* (*Trudy*) that is able to send a *challenge* R and will get K_{AB} that can be derived from a **password** that *Trudy* can try to obtaining from R and K_{AB} .



There is **variant** in which *Alice* again start presenting herself, but now the *challenge* sent by *Bob* is the encrypted R and then *Alice* is expected to find the *nonce* R , then she sends R to *Bob* in order to be *authenticated*. This **variant** provides a little bit more **security**, because when *Bob* sends the *challenge*, we understand that only *Bob* can generate that *challenge* (only him knows the **private shared key**), so *Alice* can be sure it is *Bob* that is sending the *challenge*. So *Alice* can trust the *party* with who she is *authenticating*. This doesn't change a lot for the adversary cause he still can obtain R and K_{AB} . If R has a **limited lifetime**, like *random numbers plus timestamp*, meaning that we can consider it valid for a small amount of time after which it is not valid anymore, then *Alice* trusts *Bob*, because the limited lifetime of R does not allow **replay attacks**.



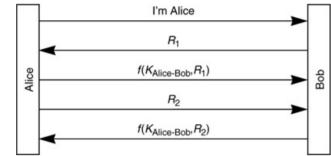
7.1.2 Timestamp Authentication

In which *Alice* and *Bob* have **reasonably synchronized clock** (this can be a weakness). *Alice* sends to *Bob* the *encrypted timestamp*, then if *Bob* can *decrypt* he can *authenticate Alice*, the *timestamp* has a *timeout* after which it expires, therefore if *Bob* saves that *timestamp* until expires, no *replay attack* is possible. It is very efficient in fact there are no *intermediate states* like before.

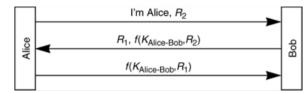


7.1.3 Mutual Authentication

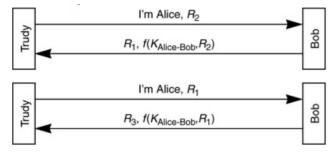
In the **Mutual Authentication** as the *one-way authentication*, *Alice* presents to *Bob*, and *Bob* reply with the *nonce* R_1 , and *Alice* responds with the *encrypted version* of R_1 . Now *Alice* is challenging *Bob* so she sends him the *nonce* R_2 and *Bob* sends back the *encrypted* R_2 . But we have too many *messages*.



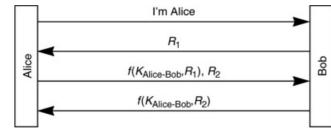
This **variant** is a compact way to do the *mutual authentication*, *Alice* presents to *Bob*, together with the *challenge* R_2 , *Bob* replies with his *challenge* R_1 and with the response to *Alice's challenge*. Finally *Alice* sends back the response to *Bob's challenge*. In this way we only exchange 3 *messages*, but this variant suffer the **Reflection Attack**.



The **Reflection Attack**, is an *attack* that aims to *construct* an extra *session* useful just for the *adversary*. The *second session* will be left *uncompleted*, but is used by the *attacker* to complete the first one. *Trudy* is pretending to be *Alice* (*messages* are **spoofed**), so she sends to *Bob* “*I'm Alice, this is my challenge R_2* ”, *Bob* is correcting responding with *encrypted* R_2 , but *Trudy* cannot send back the *encrypted* R_1 cause it doesn't know the *key*. So she start another *session* with *Bob* with the *challenge* R_1 that received from *Bob* in the *first session*, and in the *second session* he will get the *correct result* of the *encryption* of R_1 with another *challenge* R_3 , that he will *ignore*. *Bob* can obviously suspect because the *second session* will be uncompleted. We can have two different solutions, like using *different keys* for each *session* (K_{AB} and $K_{AB} + 1$) or *different challenges*.



This is another **variant less optimized**, in which the number of *messages* exchanges is 4. *Trudy* can set up an **offline-password attack**, so she plays as *Bob*, she sends the *challenge* and *Alice* sends her back the *correct response*, so *Trudy* by knowing the *pair* (*challenge, response*) can guess the **secret key** with brute force.

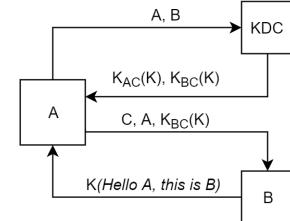


7.2 Authentication through Third-Party

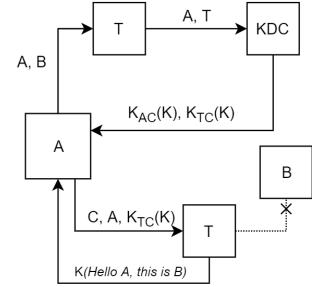
Since each *user* must have a *different key* for every other *user*, we have a *quadratic number of keys*, and this is a problem. For this reason we will use a **Trusted Party** that realizes the *authentication*. It is a very typical scenario used by companies and enterprises, and this *third party* is called **Key Distribution Center (KDC)**, or **Authentication Server**, usually we denote it with C . Every *participant* in the *network* is sharing a *secret key* only with this *server*, so with n parties there are n *secret keys* that the *server* maintains and this means that every *user*, every *party* has to be registered to the *server*. Now having this *server*, we want to implement *single or mutual authentication* as before. An optional would be to use a **session key** for *short time communications*. If the *attacker* is an **insider**, he is sharing a *private key* with the *server*, he can *sniff* and spoof *messages*, but he cannot guess *random numbers* generated by the *parties* and cannot be able to get the *private keys* of other *parties* and cannot *decrypt messages* in short time without knowing the *key* because the *attacker* is expected not to have high *computational power*.

7.2.1 First schema

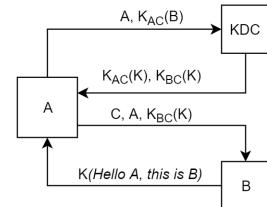
The **first schema** is presented just for educational purposes, in fact it's **very simple and weak**. *Alice* wants to talk with *Bob*, so she sends a message to $C : A, B$, saying *I am Alice, I want to talk with Bob*, C chooses the **session key** K and sends *Alice* two *messages* that are K *encrypted* with K_{AC} and K encrypted with K_{BC} . *Alice* will *decode* and will compute K , and will send to *Bob* a message constituted by three elements $(C, A, K_{BC}(K))$ like "*I'm Alice, I obtained this thing by C, this thing is K_{BC}(K)*". So *Bob* can *decrypt* $K_{BC}(K)$ and will find K and will reply to *Alice*, a *message* encrypted with the **session key**: K ("*Hello Alice this is Bob*").



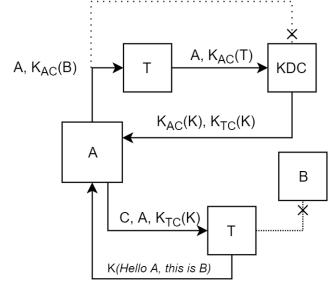
This *schema* can be **attacked** in fact, if we assume that *Trudy* can **sniff** and **spoof** then she can make the **Man In The Middle Attack**, with *Trudy* between them, so she can intercept *messages*, and create a **session key** between her and *Alice*, and another between her and *Bob*. *Alice* sends to *Trudy* (instead of C): A, B (*I am Alice, I want to talk with Bob*). Immediately *Trudy* sends to C : A, T (*I am Alice, i want to talk with Trudy*), so C will choose a K and will send to *Alice*: $(K_{AC}(C), K_{TC}(C))$. When *Alice* sends to *Bob* the *message* of three elements: $C, A, T_{TC}(K)$ *Trudy* will intercept it and will reply to *Alice* with the **session key**: K ("*Hello Alice this is Bob*").



In order to *solve this problem* we can modify the *protocol* in such a way that when *Alice* wants to talk with *Bob*, she send to $C : A, K_{AC}(B)$ so the *receiver* is **encrypted**, so an *attacker* *Trudy* cannot know that *Alice* wants to talk with *Bob*.



Even this scheme is *violable*, in fact if Alice want to talk with *Bob* like before $A, K_{AC}(B)$, but in precedence *Trudy* and *Alice* had already talk, *Trudy* can use these **previously exchanged messages**, and will intercept the *message* of *Alice*, and will send to $C : A, K_{AC}(T)$, even if *Trudy* doesn't know with who *Alice* wants to talk. C will choose a K and will send to *Alice*: $K_{AC}(K), K_{TC}(K)$. Now *Alice* will *decode* K and will send to *Bob* the message of three elements: $C, A, K_{TC}(B)$, *Trudy* will get it and finds that *Alice* want to talk with *Bob*, so she will now acts in place of *Bob*.



7.2.2 Needham-Schroeder Protocol

The **Needham-Schroeder Protocol** it's a very important *protocol* that poses the basis for **Kerberos** and it's used to create a *session key* between two *parties* by means of a *third party* and it's based on *Nonce* and *challenge response*. If *Alice* wants to talk with *Bob*, she chooses a *Nonce* N and will send to C : A, B, N . When C receives the *request*, will generate a *session key* K , and replies to *Alice* an *encrypted message* by means of the *secret key* of *Alice* $K_{AC}(N, K, B, K_{BC}(K, A))$, the *message* contains the *nonce*, the *session key*, the identity of *Bob*, and the *encryption* of K and A by means of the *key* of *Bob*. *Alice* will *decode* the *message*, will check N and B , and will send to *Bob*: $K_{BC}(K, A)$. *Bob* will *decode* this *message*, will choose another *Nonce* N' and sends back to *Alice*: $K(This \ is \ Bob, N')$. *Alice* can reply with a *message encrypted*: $K(This \ is \ Alice, N' - 1)$ so *Alice* replies with the *nonce* decremented by one. It's important to note that *Bob* is not communicating with C . Also this *protocol* can be *attacked*, and this is called **Denning & Sacco Attack**, it works as a *replay attack* and uses an old session key. Like before *Alice* chooses a N and sends to C : A, B, N , then C chooses K and sends to A : $K_{AC}(N, K, B, K_{BC}(K, A))$ but now *Trudy* will acts in place of *Alice*. *Alice* will *decode* and check N and B and will send to *Bob* $K_{BC}(K, A)$. *Trudy* as *Alice* replays to *Bob* $K_{BC}(K', A)$, with K' an **older session key**. *Bob* will *decode*, will choose an N' but will send to *Trudy* instead of *Alice* $K'(This \ is \ Bob, N')$. So *Trudy* will reply to *Bob*: $K'(This \ is \ Alice, N' - 1)$.

We can also have *Trudy* that uses the *Man in the Middle Attack*, if we have a *session key* that is compromised. There are many possible variants of this *attack*, and all these *variants* are important just to let us understanding why we need to introduce more ingredients like **timestamps**, **sequence numbers** and so on. Therefore a *variant* of the *protocol* has been invented:

7.2.3 Needham-Schroeder Protocol variant

Since the *original protocol* was *weak* and we have seen some attacks against it, the *protocol* has been **revisited** and here is the new version:

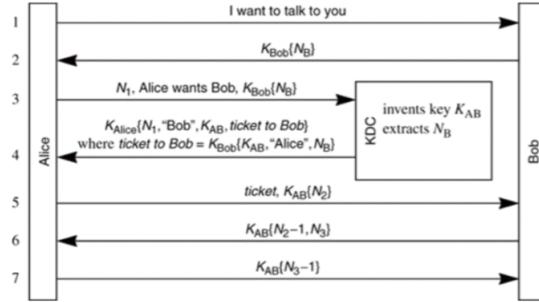
- C exchanges *messages* with both A and B ;
 - *Timestamp* t is used only between B and C ;
 - K *session secret key*;
1. *Alice* chooses a *nonce* N and sends to *Bob*: (A, N) ;

2. *Bob* chooses a *nonce* N' and sends to the *Server* $(B, N', K_{BC}(N, A, t))$;
3. The *Server* sends to *Alice*: $(K_{AC}(B, N, K, t), K_{BC}(A, K, t), N')$;
4. *Alice* sends to *Bob*: $(K_{BC}(A, K, t), K(N'))$;

So in few words, *Alice* wants to talk to *Bob*, so she sends him a *nonce* N . *Bob* sees this *message* and create a *nonce* N' and a *timestamp* t and sends to the *Server* the *encrypted version* by means of K_{AC} of the two elements, so only *Bob* and the *Server* knows the *timestamp*, and the *Server* will have the power to say if the *timestamp* is valid or not. Then the *Server* sends to *Alice* the *encrypted session key* and the *encrypted timestamp* by means of K_{AC} , and another *encrypted message* by means of K_{BC} that *Alice* has to forward to *Bob*, with the *nonce* of *Bob* *encrypted* with the new *session key*.

7.2.4 Needham-Schroeder Protocol Expanded

Alice want to talk with *Bob*, *Bob* generate a *nonce* and sends it to *Alice* encrypted by the *secret key* of *Bob*, now *Alice* sends a *message* to the *Server*. Then the *Server* generates the *session key* K_{AB} and extracts the *nonce* created by *Bob*. Then the *Server* replies to *Alice* with a *message* that contains the *ticket* to *Bob*. *Alice* *decrypts* in order to get the *ticket* and sends it to *Bob* together with a new *nonce encrypted* by means of the *session key*. *Bob* replies with a *message encrypted* by the *session key*. The last *message* is not really necessary. This last version has inspired **Kerberos**.



8 Kerberos

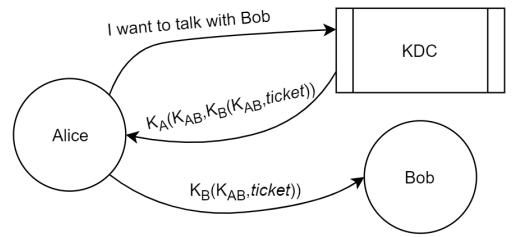
Kerberos is a *general framework* used in order to provide *authentication* in *distributed systems*, this *framework* is providing **safe access** to *resources* of a *network*. The main idea of this framework is that *users* that want to use the *service* of another party have to make a *request* to the *KDC* for a **ticket**, and then they have to present this *ticket* to that *party* who offers the *service*. Of course every *user* share a *secret key* with *KDC*. Not only human are *users*, many times also *applications* but the difference is that while humans have not strong capabilities and their *keys* are derived from *passwords*, *applications* can store *long keys* (not predictable) that are not stored in *clear*.

The **Scenario** is: *A* needs to access *service* provided by *B*, so *A* need to be *authenticated* in order to use these *services*. In some cases we could also need *B*'s *authentication*. Then there is a *trusted authority* *C*, that is a **reference Server trusted** by every party also called **KDC**, that

provides **session keys**. *KDC* is storing a **Master Key** for every *participant*, so every *user* has a *master key* that share with the *KDC*. The approach is to generate **tickets** allowing *users* to access *services* and also there can be a **timeout** for *expiring tickets*. Even the *KDC* has a own *master key*, which *uses* to *encrypt* all the other *keys*, so the *KDC* stores all the *master keys* of the *users* encrypted by *KDC's master key*. It's also important that all *machines* must have their **clocks synchronized** cause **timestamps** are used in the *protocol*.

8.1 Kerberos preliminary implementation

The *idea* is **simple**, *Alice* wants to use a *service* of *Bob*, she is sending a *request* to *KDC* for the **ticket** and the *KDC* is providing it obviously *encrypted*, after that *Alice* will use this *information* with *Bob*. All the *information* provided by *KDC* is *encrypted* by the *master key* of *Alice* K_A and such *information* contains the *session key* K_{AB} and the **ticket** for *Bob*. This *ticket* contains further *information*, and is *encrypted* by means of the **master key** of *Bob* K_B , therefore only *Bob* can read such a *ticket*. Such *ticket* contains the *identity* of *Alice* of course, the *session key* and some other stuff like the *lifetime*. So basically when *Alice* asks to *KDC* to want to talk with *Bob*, *KDC* replies with $K_A(K_{AB}, K_B(K_{AB}, \text{ticket}))$. So *Alice* will *decrypt* the *message* obtaining the *session key* and the *ticket encrypted*, and will send the *encrypted ticket* $K_B(\text{ticket})$ to *Bob* in order to show she is allowed to use *Bob's service*.



8.2 Kerberos simplified version

In the **simplified version of Kerberos** we have:

1. *A* sends to *C* : A, B, N ;
 - So *Alice* asks to *KDC*, "I am *Alice*, i want to talk with *Bob*, here my *nonce N*" (this *nonce* will be used in order to verify the *authenticity* of *C*);
2. *C* sends to *A* : $[\text{Ticket}B, K_A(K_{AB}, N, L, B)]$;
 - So the *KDC* reply to *Alice*, with a *ticket encrypted* in K_B (the *master key* of *Bob*), and with the *encryption* by K_A (the *master key* of *Alice*) of the *session key* K_{AB} , the *nonce N*, the **lifetime L** and the *identity* of *Bob*;
3. *A* will check *N* and known the lifetime, sends to *B* : $[\text{Ticket}B, K_{AB}(A, t_A)]$;
 - So *Alice* will check the *nonce*, and learns about *ticket lifetime*, then *Alice* sends to *Bob* the *TicketB* and the **authenticator** which is $K_{AB}(A, t_A)$ and t_A is the *timestamp* of *Alice*;

4. B will check that A 's *identity* in *TicketB* and in *authenticator* are the same, *time validity* of *ticket*;
- So *Bob* will *decrypt* the *ticket* with his *master key*, obtaining the *identity* of *Alice* and the *session key*, then with the *session key* will check if the *authenticator* and the *identity* of *Alice* corresponds to the *identity* in the *ticket*, and will also check if the *ticket* is still valid with the *timestamp*;
5. B sends to $A : K_{AB}(t_A)$;
- So *Bob* will reply to *Alice* with the *encryption* of the *timestamp* with the *session key* K_{AB} ;

But we have some **practical problems**, because for every *session* between *user* and *KDC*, the *messages* should be protected with the **master key** (derived from the *user's password*), so every time *Alice* requires a *ticket*, this *message* has to be *encrypted* but this cause too much overhead. We have 2 possible solutions:

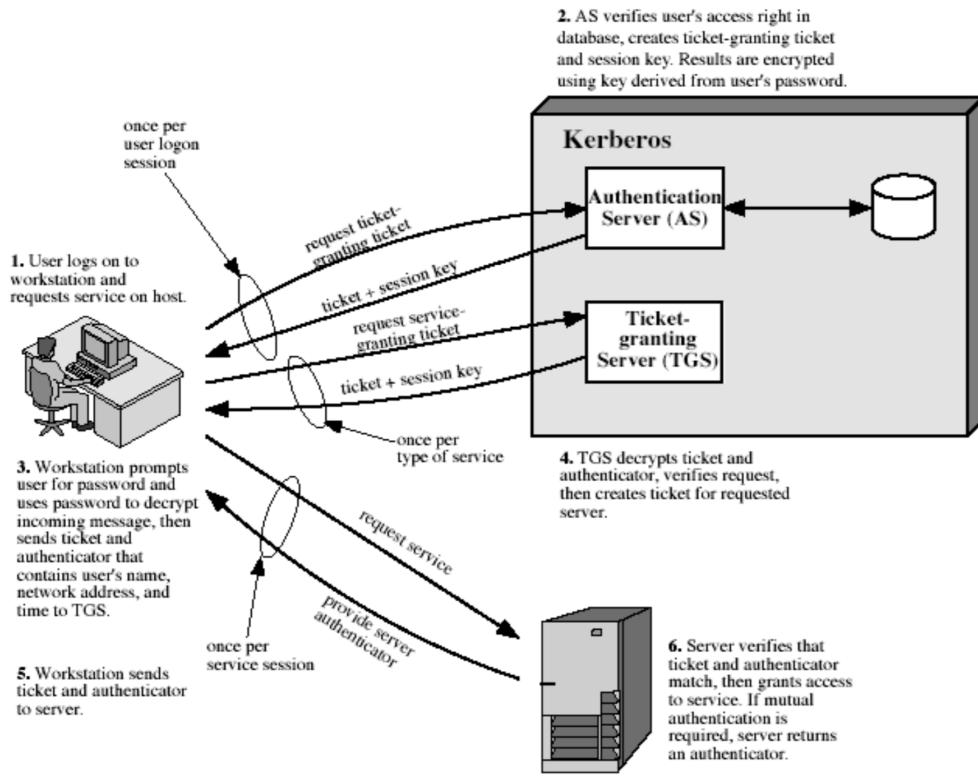
- Every time a *user* needs the *Server* he has to retype the *password*;
- The *user's password* is **temporarily stored locally**, in such a way that an *user* doesn't need to retype it;

These two solutions are *inadequate*, so we need to introduce another type of **Kerberos**:

8.3 Ticket-granting Ticket

TGT, or *ticket-granting ticket*, uses a *meta-ticket* used to ask the real *ticket*, the idea is that A is having a *ticket* for asking *tickets*, so every time he needs a *ticket* he shows a certain *ticket* instead of retying the *password*. We have different steps:

1. $Alice$ login for the first time and requests the *ticket-granting ticket* to the *authentication server AS*, that will check its local database;
2. AS will send to $Alice$ the *ticket-granting ticket* and the *session key* S_A , with a fixed *lifetime* and other infos;
3. $Alice$ now wants to use a particular *service*, so she will request it by showing the *TGT* to the *TGS* (*ticket-granting server*, the *server* that provides *tickets for services*);
4. *TGS* will answer with the *requested ticket* and a *session key* valid between *Alice* and the *service*;
5. $Alice$ now can request the *service* to the *server* cause he has the *service-granting ticket*;
6. The server will answer with *server authenticator*;



In details can summarize all these *operations* in three *phases*:

- **Login phase:**

- Alice does the *login* typing her *password* and the *KDC* derives from this *password* a *session key* S_A (with a *lifetime*, cause *KDC* will not keep it), then *KDC* will answer to Alice with a *TGT* and the *session key* S_A encrypted in K_A . This *TGT* is also *encrypted* with the *master key* of the *server* K_{KDC} , so Alice will have a message like: $K_A(S_A, TGT) = K_A(S_A, K_{KDC}("Alice", S_A))$;

- **Ticket request phase:**

- In which Alice request to the *KDC* a *session key* in order to talk with Bob, so she need to be *authenticated* by sending to the server the *TGT* and an *authenticator* (given by the *timestamp* and the *session key* S_A). Once received the *server* will: decrypt *TGT* to get S_A , will decrypt the *authenticator* in order to verify the *timestamp*, will find *Bob's master key* K_B , and will create a *session key* K_{AB} , so with all these information will create a *ticket* to Bob: $K_B("Alice", K_{AB})$. So will send back to Alice: $S_A("Bob", K_{AB}, \text{ticket to Bob})$ encrypted in S_A ;

- **Ticket usage phase:**

- Now *Alice* has the *ticket*, so in order to use it she sends it to *Bob*, together with the *timestamp* encrypted by K_{AB} (like in *simplified version of Kerberos*). So *Bob* will decrypt the *ticket* which contains the *session key* K_{AB} and so he will be able to decrypt the *timestamp* in order to trust *Alice*, and at the end will reply with an incremented *timestamp* in order to get authenticated by *Alice*.

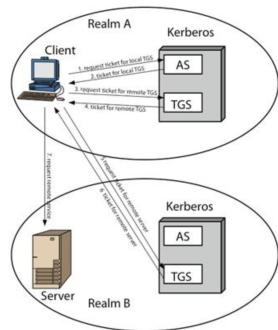
Authenticator is often the *encryption* of a *timestamp*, so we need a *Global Synchronous Clock*, and it is used to avoid the *replay* of *messages* sent to the same *server* by the *adversary* (old *message* are eliminated) and to avoid *replay* to a *server* (when there are many *servers*) and it's important to note that the *authenticator* doesn't guarantee *data integrity* so we need a *MAC*.

KDC and **TGS** are not the same, in fact they are separate *entities* for historical reasons, in fact one *KDC* can serve different systems of *TGS*. Commonly the *server* who provides the *TGT* is the **Authentication Server (AS)** and the *users* interact with it once a day (or once until the *TGT* is valid), while the *server* who provides the *service tickets* is the **Ticket-Granting Server (TGS)**. The important thing is that we understand that the *Server* who provides the *TGTs* and the *Server* that provides the *Service Tickets* are **different entities**.

8.4 Kerberos Realms

In several cases an *user* want to use a *service* of another *network*, **Realms** is a way to implement that, and is like a *network* where there are *users*, *KDCs* and the *Master KDC*. Each *Realm* has a different **master KDC** (we have seen that in addition of a *KDC* there can be other *KDCs*), if you have let's say 2 realms, your *master KDC* should be a principle of the *master KDC* of the other *realm* and vice-versa. This means that there must be some *key agreed* between different *master KDCs*. You can setup so that you have just one *master key* shared between all possible *KDCs*, but it is actually unrealistic. A very critical part of using a service in *Kerberos* is being *authenticated*, but who can authenticate a user? Only *KDC* of the same *Realm*! So once the *authentication* is made locally, then the *user* can ask some *service* coming from another *Realm*. The main scheme is the following:

Suppose we have a *client* in the *Realm A* which wants to use a *service* provided by the *server* in *Realm B*. First of all the *client* authenticates to its *local KDC* (in particular to the *AS*). Once *authenticated*, the client gets the *TGT* which will be used to ask to the *local TGS* for the *ticket* in order to communicate to the *KDC* of the other *realm*. Once obtained the *ticket*, the client can talk with the *TGS* of the other *realm* in order to obtain the *ticket* for the *server*. These cause *KDCs* store information only about users of the *local realm*, so first of all the client has to authenticate locally then a *TGS* can provide *tickets* only for *local services* or *ticket* for *TGSs* of the other *realms*, not for *services* of other *realms* all this for *scalability* reasons.



9 Authentication based on public keys & X.509 & PKI

The idea is to use **signed messages** containing *challenges* or *timestamps*, and the *signatures* can be verified using the **public key**. The problem is that is how can *Alice* be sure that the *public key* stored is the one of *Bob* and not the *public key* of an *attacker* (that if gives its own *public key*, then can be authenticated as another user), so we need to use a **trusted third party** that guarantees **correctness of public keys**.

9.1 Needham-Schroeder public key

We call K_{PX} the *public key* of an user X, and Sig_C the digital signature of C (the *trusted authority*). **Needham-Schroeder** tried implementation of *authentication* through *public key*. The scheme is simulating the presence of a *trusted authority* that provides *public keys* to users:

- A to C: $\langle A, B \rangle$;
 - So, In order to be **authenticated** Alice can ask to the *authority* “I am Alice, I want to talk to Bob”;
- C to A: $\langle B, K_{PB}, Sig_C(K_{PB}, B) \rangle$;
 - The reply is the **identity** of B, its **public key**, and the **digital signature** of this whole *message* (so the *encryption* of the whole message by means of *authority's private key*, so if we succeed in *decrypting* with *authority's public key* we can trust the content of the *message*);
- A checks **digital signature** of C, generate **nonce** N and sends to B: $\langle K_{PB}(N, A) \rangle$;
- B decode the *message* and send to C: $\langle B, A \rangle$;
 - So B will decode the *message* with its own *private key*, and will ask to the *authority* in order to check the **identity** of A, “I am Bob, I want to talk to Alice”;
- C to B: $\langle A, K_{PA}, Sig_C(K_{PA}, A) \rangle$;
 - So the *authority* will reply with the *identity* of A, its *public key* and the *digital signature* of the *message*;
- B checks *digital signature* of C, retrieve K_{PA} generate *nonce* N' and sends to A: $\langle K_{PA}(N, N') \rangle$;
- A will decode the *message*, checks N and sends to B, $K_{PB}(N')$;
 - So at the end with this last message, **both the parties are authenticated**;

9.1.1 Needham-Schroeder public key Attack

This is a **Man in the middle attack**, in which just like *Alice* and *Bob*, *Trudy*, the *attacker*, can talk to the *server*. This *attack* is made in two sessions:

- **R1:** *authentication* between *Alice* and *Trudy*;
- **R2:** *authentication* between *Trudy* (that pretends to be *Alice*) and *Bob*;

The **attack** can have place only on a certain precondition: *Trudy* must be able to induce *Alice* to start an **authentication session** with *Trudy*.

- *Trudy* has succeed to convince *Alice* to issue an **authentication** with her, so *Alice* sends $K_{PT}(N, A)$ to *Trudy*;
- Then the *attacker* can use this *message* in order to start a **communication** with *Bob*, so will send to him: $K_{PB}(N, A)$;
- Then *Bob* will reply to *Trudy* (thinking that she is *Alice*) with $K_{PA}(N, N')$;
- Of course *Trudy* **cannot decrypt** this, so she will forward the *message* $K_{PA}(N, N')$ to *Alice*;
- *Alice* will **decrypt** it, and will send to *Trudy*: $K_{PT}(N')$;
- At this point *Trudy* has got the **nonce** N' , she can send to *Bob* $K_{PB}(N')$ so she won his **challenge**;

So with this **attack** *Bob* thinks that he is talking with *Alice*, but instead there is *Trudy* in the *middle*. The only required thing is that *Trudy* is able to force *Alice* to issue an *authentication* with her, and if they belong to the same enterprise or something like this it is not so difficult because *Trudy* maybe has some reasons to invite *Alice* to be *authenticated* by *Trudy*.

9.1.2 Needham-Schroeder public key fixed variant

There is so a **fixed variant**, that permits to avoid the *attack* just described, that is very similar to the previous one, in which the only thing that changes is that when *Bob* is sending back the two *nonce* instead of $\langle K_{PA}(N, N') \rangle$ will send $\langle K_{PA}(B, N, N') \rangle$ so he send his **identity**. We can avoid the *attack* cause *Trudy* cannot send to *Alice* the identity of *Bob* while she is talking to *Alice*, in fact if *Trudy* will forward this message *Alice* will know that she is talking with *Bob* and not with *Trudy*.

9.2 X.509 Standard

X.509 is a **standard** for *authentication* of different types, which means that they define *protocols*, not *encryption* or *hashing functions* specially related to protocol's steps. In this standard 3 *authentication protocols* have been defined:

- **One-Way authentication protocol:**

- Is very simple, if *Alice* wants to be authenticated by *Bob* she sends a single *message* that contains: the **digital certificate** of *Alice* (taken by the *trusted authority*), the **digital signature** of D_A and the *message* D_A that contains:
 - * *Timestamp* t_A ;
 - * *Identity* of *B*;
 - * The *session key* K_{AB} encrypted with the *public key* of *Bob*: K_{PB} ;
- So, *Alice* must know in advance the key of *Bob*, so we can see this like *Alice* is the client and *Bob* the server, since this scheme is not symmetric. The *message* sent by *Alice* is: $\langle CertA, D_A, Sig_A(D_A) \rangle$ where D_A is: $\langle t_A, B, K_{PB}(K_{AB}) \rangle$;
- **Nonce** can also be included in the *message* in order to detect *replay attacks*;

- **Two-Way authentication protocol (*mutual authentication*):**

- There will be a *message* from *Alice* to *Bob* ($A \rightarrow B$), and another *message* from *Bob* to *Alice* ($B \rightarrow A$). In the two *messages* D_A and D_B will appear something like $K_{PB}(k)$ (for *Alice*) and $K_{PA}(k')$ (for *Bob*), these are **proposed session key**. This because it is not rare to encrypt *messages* $A \rightarrow B$ with a *session key* and $B \rightarrow A$ with another *session key* and both parties must know both *session keys*. We have to notice that in the first message *Alice* is sending the *proposed session key* encrypted by *public key* of *Bob*, but without having the *digital certificate* of *Bob*, and this is *unrealistic*;
- So during $A \rightarrow B$ *Alice* will send $\langle CertA, D_A, Sig_A(D_A) \rangle$ where $D_A = \langle t_A, N, B, K_{PB}(k) \rangle$;
- Instead from $B \rightarrow A$ *Bob* will send $\langle CertB, D_B, Sig_B(D_B) \rangle$ where $D_B = \langle t_B, N', A, N, K_{PA}(k') \rangle$;

- **Three-Way authentication protocol**

- The most robust version, in which the authentication is based on *nonces*, and it is useful for *unsynchronized clocks*. We have the exchange of **three different messages** (O represents a *timestamp* since it is *optional* in this *protocol*) :
 - * *Alice* will send to *Bob* $\langle CertA, D_A, Sig_A(D_A) \rangle$ where $D_A = \langle O, N, B, K_{PB}(k) \rangle$;
 - * *Bob* reply to *Alice* with: $\langle CertB, D_B, Sig_B(D_B) \rangle$ where $D_B = \langle O, N', A, N, K_{PA}(k') \rangle$;
 - * The last *message* is from *Alice* to *Bob*: $\langle B, Sig_A(N, N', B) \rangle$ so she send the two *nonces* and the identity of *Bob* signed by the **digital signature**;

- **Canadian Attack:**

- Before the protocol was:
 - * $B \rightarrow A : N_B$;

- * $A \rightarrow B$: $\langle CertA, N_A, N_B, B, Sig_{-A}(N_A, N_B, B) \rangle$;
- * $B \rightarrow A$: $\langle CertB, N_A, N_B, A, Sig_{-B}(N_A, N_B, A) \rangle$;
- But with the **Canadian Attack**:
 - * Trudy act like Bob so: $T \rightarrow A : N_T$;
 - * $A \rightarrow T$: $\langle CertA, N_A, N_T, B, Sig_{-A}(N_A, N_T, B) \rangle$;
 - * Trudy act like Alice so: $T \rightarrow B : N_A$;
 - * $B \rightarrow T$: $\langle CertB, N_A, N_B, A, Sig_{-B}(N_A, N_B, A) \rangle$;
 - * Trudy now can reply to Alice: $\langle CertB, N_A, N_B, A, Sig_{-B}(N_A, N_B, A) \rangle$, so Trudy is now **authenticated**, the presence of N_B and not N_T d'amore says has the same role as the use of Bob in step 6 of original NS protocol;

9.3 PKI: Public Key Infrastructure

Certificates are provided by trusted **Certification Authorities**. The specific *Certification Authority (CA)* provides *certificates* just for the *users* in its domain, when someone asks for the *public key* of an *user*, the *CA* replies with the *public key* of the *requested party* signed by the *private key* of the *CA*. This also means that we have only to remember just one *public key*, that is the **CA's public key**. It's important to note that if *CA* is not *trusted* all its *certificates* are useless, *keys* are not used forever and the length of the *keys* is related to security level.

9.3.1 X.509 Certificate's Fields

The **X.509 standard** describes the structure of a **Digital Certificate**. The scheme is the following:

Signed fields	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>Version</td></tr> <tr><td>Certificate serial number</td></tr> <tr><td>Signature Algorithm Object Identifier (OID)</td></tr> <tr><td>Issuer Distinguished Name (DN)</td></tr> <tr><td>Validity period</td></tr> <tr><td>Subject (user) Distinguished Name (DN)</td></tr> <tr> <td>Subject public key information</td><td style="text-align: center; padding: 0 10px;">Public key Value</td><td style="text-align: center; padding: 0 10px;">Algorithm Obj. ID (OID)</td></tr> <tr><td>Issuer unique identifier (from version 2)</td></tr> <tr><td>Subject unique identifier (from version 2)</td></tr> <tr><td>Extensions (from version 3)</td></tr> <tr><td>Signature on the above fields</td></tr> </table>	Version	Certificate serial number	Signature Algorithm Object Identifier (OID)	Issuer Distinguished Name (DN)	Validity period	Subject (user) Distinguished Name (DN)	Subject public key information	Public key Value	Algorithm Obj. ID (OID)	Issuer unique identifier (from version 2)	Subject unique identifier (from version 2)	Extensions (from version 3)	Signature on the above fields
Version														
Certificate serial number														
Signature Algorithm Object Identifier (OID)														
Issuer Distinguished Name (DN)														
Validity period														
Subject (user) Distinguished Name (DN)														
Subject public key information	Public key Value	Algorithm Obj. ID (OID)												
Issuer unique identifier (from version 2)														
Subject unique identifier (from version 2)														
Extensions (from version 3)														
Signature on the above fields														

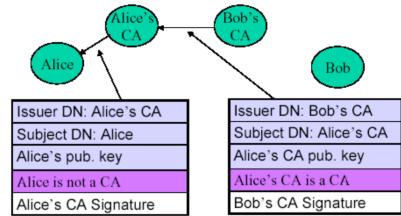
The **certificate** is composed by many **fields**, this cause *fields* are related to the management of such *digital certificate*, the practical management. For instance there is the **serial number field**: every *certificate* is having a *serial number* that is unique among all the *digital certificates* of the same *CA*. Then there is the *validity period*. There is also the *name* of the *user* (the *identity*), the *public key* of the *user*, and information about the *algorithm* in order to use that *public key*. The *extensions field* can contain *extensions*, that are *additional fields*. All the *fields* are signed by the **certification authority** but in order to check the *digital signature* you need the *public key* of the *CA* that is given by another *CA*, but you want to trust also that different *CA* so this generates a *chain* and this is a problem that we will be studying how to solve it.

- **VERSION**: there are three versions, *version 1*, *version 2*, *version 3*. If *version 1* is used we find 0 in this filed, if *version 2* we find 1, if *version 3* we find 2;
- **SERIAL NUMBER**: integer number concatenated with the *CA's name* and this is a unique identifier;

- **SIGNATURE ALGORITHM**: it describes the **algorithm** used for signing this *certificate*;
- **ISSUER**: it is the name of the *CA* creating the *certificate*, it is a name that follows the *X.500 standard*;
- **VALIDITY**: it contains 2 *subfields*, the time the *certificate* becomes *valid* and the time for which after it the certificate is *not valid* anymore, so it is an **interval of validity**;
- **SUBJECT**: name (according to *X.500 standard*) of the entity whose *key* is being certified;
- **SUBJECT PUBLIC KEY**: it contains two *subfields* that are the *algorithm* with which the *public key* has been created, and the *public key* itself.
- ...

9.3.2 Hierarchy of CAs

Now let's come back to the problem of how *Alice* and *Bob* can check their *certificates*, if *Alice* and *Bob* use the same *CA* there is no problem, but if they use different *CA* then we can have a problem. In general *CA*'s form a **hierarchy**, for example it may be that the *public key* of *Alice* is certified by the *certificate* provided by *Alice's CA*, and the *certificate* provided by this *CA* is certified by *Bob's CA*. Note: certifying a *certificate* of a certain *CA* means providing the *public key* of that *CA*.



9.3.3 Certificate Revocation

We also need to **revoke certificates** for several reasons, since *certificate* are **valid** for a limited time (*CA*'s want to be paid), and they can be revoked before the *deadline* if *user's secret key* is not *safe* anymore, or *user* is not certified by *CA*, or the *secret key* of the *CA* is now compromised.

In order to implement this whole feature there are the so called **certification revocation lists (CRL)**. The *CRL* has a certain format specified again by **X.509 standard**. There is an important part of the structure in the middle, where there are the entries. Each entry has the *Certification Serial Number* (the *serial number* of the *revoked certificate*), the *Revocation Date* (when the *certificate* was removed), and the *CRL entry extensions*. The last fields is the **signature** of the **digital certificate**. There are two fields that are very important: **THISUPDATE** that represent the date of the publication of the list, and **NEXTUPDATE**, the expected date of the next *CRL* to be issued.

Signed fields
Version of CRL format
Signature Algorithm Object Identifier (OID)
CRL Issuer Distinguished Name (DN)
This update (date/time)
Next update (date/time) - optional
Subject (user) Distinguished Name (DN)
CRL Entry Certificate Revocation Date CRL entry extensions
CRL Entry... Serial... Date... extensions
....
CRL Extensions
Signature on the above fields

9.3.4 OCSP

OCSP or Online Certificate Status Protocol, it is a new *protocol*, an alternative to *CRL* more *agile*, used for obtaining the **revocation status** of an *X.509 digital certificate*. If the *certificate* is *Version 3*, there is an *extension* where the browser can get an *URL* and connect to it in order to make *queries*, and this is done according to this *protocol OCSP*.

9.3.5 PGP: Pretty Good Privacy

PGP is a *trust model* for *email certification*, in which there aren't any *trusted CA*, so it's like a **distributed approach**, in which each *user* acts like a *CA* and decides for himself. The *certificates* will contain *email addresses* and *public key*, and they are *signed* by *one or many users*, so if you *trust* a sufficient number of the *user* signing a *certificate* then you assume the *certificate* is good. Each *user* keeps info on *public keys* of other *users* and *signatures* of these *keys*, together with *trust value* of the *key*. The creator believed in a **Web of Trust**, that is something that is proving the *correctness* of the associations between a *public key* and an *identity* not by means of a *digital certificate* issued by a *CA* but by means the **trust** obtained from some other *users*.

10 Passwords

Passwords are so common in *computer systems*, despite their **insecurity**. The main problem is that *humans* have not capabilities for remembering long and secure *passwords*, and in many cases they use *passwords* containing *words* belonging to some *dictionary* (words which are meaningful), and so they are vulnerable to **dictionary attacks**, and also **Trojan Horses attacks** that can be performed while the *user* types the *password*. In this context, *computers* are much more secure since they can store long and secure *passwords/keys*, even *encrypted*.

A **dictionary attack** is an attack in which an *attacker* take a long list of words from a long list of *words* taken maybe from the web, and they try all possible words. *Crackstation.net* allow you to write an *hash* and they will send you back the *password* (done by *precomputation, lookup tables*). In **Eavesdropping** the *adversary* is *sniffing*, so even if you send the *password* encrypted, the *attacker* can just store the *encrypted password* and use it, without knowing the *plain password*. This is why **authentication** should be **different** every time.

10.1 Unix Password Hash

One can think of inventing a *password* and **hashing it**, then using the *hash* as *password*, you can also store somewhere the *hash*. So the idea is that the *password* typed by the user is converted to a *key* of 56 bit with 8 parity bits added (*DES key structure*). At the beginning **UNIX** used this approach, storing only the first 8 characters of the *key* (when *ASCII* was the standard, which used 7 bits in order to represent characters). What changed then is that *DES* was used, and was performed 25 times on a 8 bytes string of zeroes using the *key* created as mentioned above, so the *key* stored in the *system* was $DES_K(DES_K(\dots DES_K(00000000)))$ for 25 times.

There is a file */etc/passwd* in which each *row* is a *user*, where there are collected useful information, there are collected also the **encrypted passwords**, and every *user* can read such *file*. In the modern *Unix* implementation the *passwd* file does not contain anymore the *encrypted passwords* because it was a weakness since one could copy that file and do some precomputation on stored encrypted passwords.

In order to avoid *dictionary attack*, and in order to increase *security*, when a *user* is storing the *password*, the system generates a 12 bit random number, the **salt**, that is concatenated to the *password*, this is the **salted password**, that will be encrypted. The *salt* is *per-user generated* and is

stored as *clear text*, so the *attacker* can no longer use *dictionary attack* because of the *salt* (actually he can but it requires more **computational effort**). What we want is to reach an *approach* in which *users* only have to remember their *password*, without further heavy crypto-information, and then we want to use *cryptography* in order to make this approach secure. We use this approach in such a way that no *security information* is kept at the *user's machine* and someone impersonating either *party* will not be able to obtain any *information*.

10.2 Lamport's Hash

Lamport invented the **OneTimePassword**. It works in the following way:

- *Alice* is choosing a *password*, *Bob* is the server that stores the *password*;
- Instead of storing the *password*, *Bob* stores the name of *Alice*, a number n a relatively large number, and the **n-times hash** of the *Alice's password*;

So *Bob* stores: $\langle \text{username}, n, h^n(\text{psw}) \rangle$;

- Then when *Alice* has to *authenticate*, she sends *Bob* the *password* hashed $n - 1$ times;

So she send: $x = h^{n-1}(\text{psw})$;

- So *Bob* computes the *hash* of what receives and checks whether the *result* is the same as what *Bob* stores in the database for *Alice*;

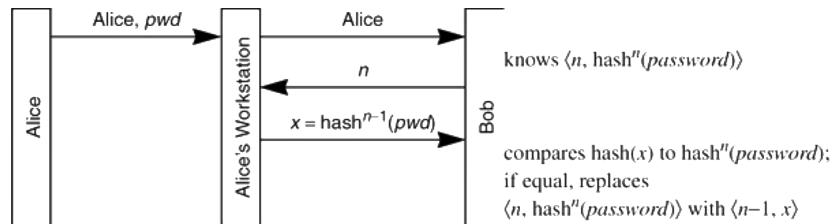
Bob computes $h(x) = h(h^{n-1}(\text{psw})) = h^n(\text{psw})$ equal to what *Bob* has stored;

- Then when there is a **matching**, n is decremented by one, and he replace the old *n-times hash of the password* with x (that is $n - 1$ times hash of the *password*);

So: $\langle \text{username}, n, h^n(\text{psw}) \rangle \rightarrow \langle \text{username}, n - 1, x \rangle$;

- Note that this prevents **replay attack** since next time *Alice* will send the $n - 2$ times hash of *password* (the *attacker* cannot guess the previous hash knowing the next one), and so on;

- There is also the **salted version**, where the *password* is concatenated with the *salt*, so $h(\text{psw}|\text{salt})$. This approach is good because *Alice* can always reuse the same *password*, the *salt* is stored at *Bob's* at setup time, and *Bob* sends salt each time along with n ;



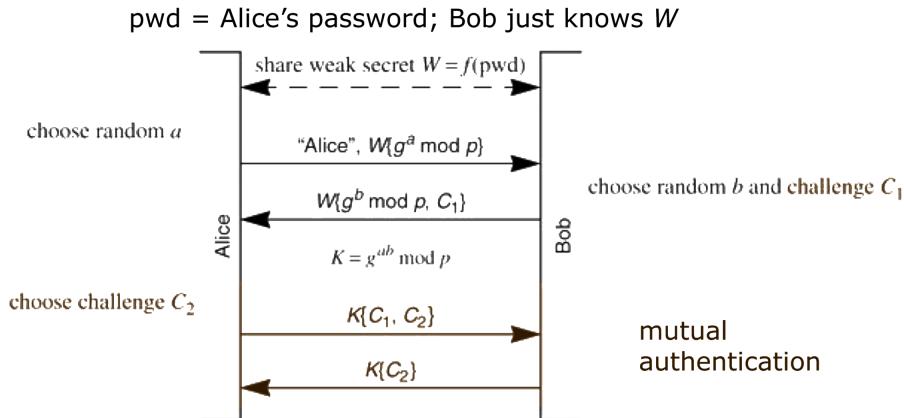
There is an attack called **Small n attack**, in which when *Bob* has to send to *Alice* the number n , *Trudy* will impersonate *Bob* and can send to *Alice* an n' such that $n' < n$, and *salt*, so what happens is that *Alice* sends to *Trudy* the n' times *hash of the password*, so *Trudy* can impersonate

Alice until n is equal to n' so for $n - n'$ times. In order to avoid this, the *Alice's workstation* shows submitted n to *Alice* to verify the *approximate range*. Instead if *Alice workstation* is too **dumb** to do *hashing*, *Alice* is given a list of *hashing* from n so she uses each *hash* once, in such a way that this *attack* is prevented.

10.3 Encrypted Key Exchange, EKE

The problem is that it is still possible to use a dictionary attack if weak keys are chosen. So we introduce an authentication approach called Encrypted Key Exchange, or EKE, deeply based on Diffie-Hellman, in which we have mutual authentication and which define a session key. In this scenario two parties share a weak secret W , that is a function that that only the two parties know, $W = f(\text{psw})$.

- Alice chooses a random a , and will send to Bob: $W(g^a \bmod p)$;
- Bob is able to decrypt it case he knows the secret W , then will choose another random b and a challenge C_1 , then will reply to Alice with: $W(g^b \bmod p, C_1)$.
- In the meanwhile Bob chooses the key as the DH protocol states, $K = g^{ab} \bmod p$;
- Alice will be now able to generate the key too, so will send back to Bob: $K(C_1, C_2)$, where C_2 is another challenge;
- Then Bob will reply only with $K(C_2)$, so now both the parties knows the key and are mutual authenticated;



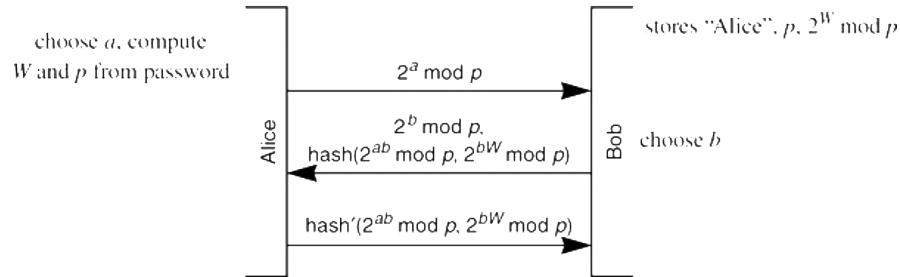
This *solution* solves the **man-in-the-middle** weakness of *Diffie-Hellmann*, and **EKE** is strong with respect to *replay attacks* (since a is changed every time), and *dictionary attacks* even if the chosen *password* is weak, cause the choice of random a doesn't allow the *attacker* to compute g^a . The *authentication* is **strong**, and the *attacker* can act in place of *Alice* only if knows the *password*.

10.3.1 SPEKE & PDM

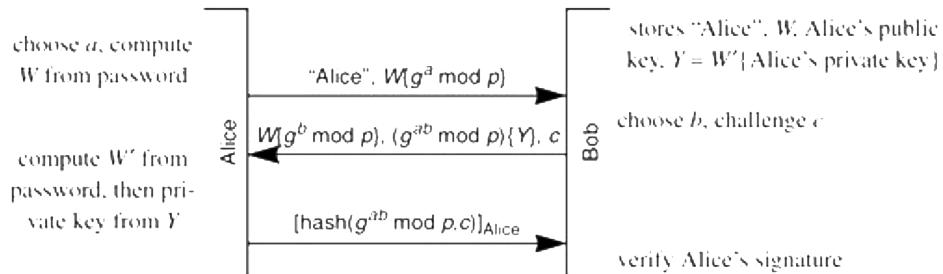
SPEKE, or **Simple Password Exponential Key Exchange**, is a *variant*, which uses W instead of g , so *Alice* and *Bob* will exchange $W^a \bmod p$ and $W^b \bmod p$, and the **session key** $K = W^{ab} \bmod p$.

PDM, or **Password Derived Moduli**, is another *variant* in which we choose p depending on *password*, and $g = 2$.

EKE, *SPEKE*, and *PDM* can be modified in order to have a **augmented property**, the idea is for *Bob* to store a *quantity* derived from the *password* that can be used to verify the *password*, with *Alice's machine* that must know the *password*. So **augmented PDM** for example:



So *Bob* (the *server*) stores p and $2^W \bmod p$ where W is still a *hash* of *user's password*. So, instead of requiring *server* to do an additional *DH exponentiation*, it does an **RSA verify operation** which is less expensive. Instead **augmented EKE**:

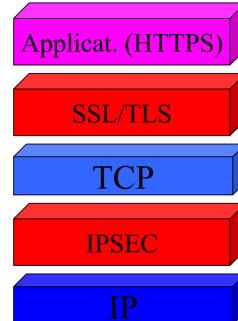


In which *Bob* stores Y which is *Alice's private key* encrypted with a function of her *password*, different *hash* of the *password* than W , and also stores **Alice public key** corresponding to the *encrypted private key*. *Alice* sends the usual first *EKE message* with her *DH value encrypted* with W . *Bob* reply with his *DH value* encrypted in W and also Y encrypted with the *session key*. Then *Alice* will extract Y by decrypting it with the *session key* to obtain her *private key*. *Alice* signs a *hash* of the *session key* and the challenge c , so *Bob* verify her **signature** using the stored *public key*. This achieves **mutual authentication** and **augmented property**, since *Alice* has got her *private key* from the decrypt thanks to the *public key*.

11 IPSEC

We started by saying "*we use cryptography for information security*", by *listing requirements for information security (confidentiality, data integrity, authentication...)* and we showed how to use *cryptography* for achieving those requirements. We say that a **protocol** is an implementation of some idea, but it is more than just *cryptography* because a *protocol* also provides details about how to decide a *key*, in what format you should exchange *information* etc., so we aim at a way to do, not at a **real protocol**. Now we start to see some *security protocols* **IPSEC**, **SSL/TLS** but the idea is that we don't want to be able to develop an *implementation* of the *protocol* due to the high number of details these *protocols* we are going to discuss have. You will be asked to make **choices**, the committer will ask you "*you have to design a secure application*", but what is means **secure**? The committer replies with "*I don't know, it is up to you to decide what is secure*". What we mean is that we **do not want** to know what kind of *encryptor* to use, we want to know for example whether **IPSEC** is able to grant **data integrity**, or **authentication**, or both and so on. This is called **black-box analysis**, sometimes at the exam *D'Amore* ask such kind of *analysis* and many people they open the box and tell "*ok this works by deciding the DH key exchange...*" but this is not a *black-box analysis*.

In the *classical protocol stack* we have what is called the **application level**, what is called the **transport level TCP**, then the **network level IP** and then lower levels that we are not considering. **IPSEC** works between *TCP* and *IP*, so between transport level and network level, so **TCP** uses **IPSEC** (remember that using in this context means that the *higher level* uses the *lower level*). **IPSEC** will protect *IP packets* at each hop, in fact there is a *shared key* among *two routers* that are connected by a *link*, so all traffic in **encrypted** (even *IP headers*), but it require *cooperation* among *routers* and a significant *computational effort*. **IPSEC** is very important cause is a *security protocol* than can be used by all applications.

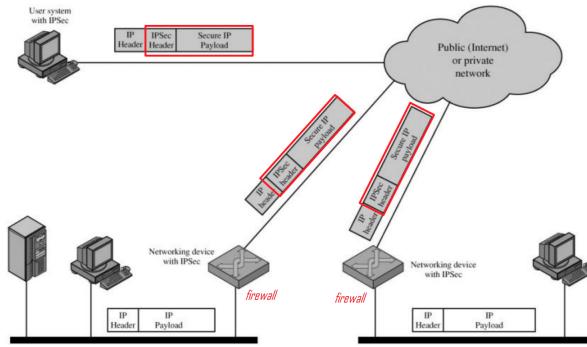


IPSEC is used for:

- **Authentication:**
 - So we are able to understand who are the *two parties* that are exchanging *messages*, and also implies *data integrity* in some sens;
- **Confidentiality:**
 - Refers to protecting *information* from being accessed by *unauthorized parties*, so only the *people* who are authorized to do so can gain access to *sensitive data*;
- **Key Management:**
 - A standard for generate *session keys* used during the *communication*, and other keys, but we are not study this feature;

IPSEC implements all this but this not means I want all of them, maybe I want only *authentication*, or only *confidentiality*. Previously **IPSEC** was a standard mandatory for **IPv6**, like if **IPSEC** was

a *subset* of *IPv6*, but now it's an *optional*, so now we will ignore *IPv6*, we only consider **IPv4** since it is also the most used.



This figure shows some uses of **IPSEC**, like between the *ending point* of a *LAN* and the *starting point* of another *LAN*. This means that within a *LAN*, *packets* are normal packets, but *packets* between different *LANs* are protected by **IPSEC**, because basically we expect to know who is in our *LAN* (maybe our home) and who is in the other *LAN* (maybe our office), but what is between the two *LAN's* is unknown. There are various benefits of **IPSEC** like:

- You can setup your *gateway* (firewall/router) in order to provide **strong security** for the traffic *out-coming* and *incoming* your *gateway*, so you do not need to protect the whole *local traffic*, it is sufficient to protect the *gateway*;
- You cannot **bypass** the *gateway* just because it is not working, the *attacker* is not able to change the behavior of the *gateway* from a remote position;
- It is **transparent** to *applications*, so changing things at *low level* is okay because *applications* will continue to work in the same way;
- It is **transparent** to *end users*, so it permits to realize a **VPN**, *virtual private network*;
- The principal feature is that **it can encrypt and/or authenticate all traffic at the IP level**;

Basically **IPSEC** is implemented as an **extension header** that follows the *main IP header*. This *header* is composed by two part:

- **AH: or Authentication Header**, that is the *header* that provides *data integrity* and *authentication*, it's based on **MACs** (in particular *HMAC*) and requires a *shared secret key*;
- **ESP: or Encapsulating Security Payload**, that it the *header* used for *encryption*;

You can choose to use one of them, the other one, or both (that's why we said **IPSEC** provides many *services* but it is not mandatory to choose them all). Actually *ESP* can also provide some *authentication features*, that overlap with the ones of *AH*, therefore *ESP* can be divided into *ESP* and *ESP_a*.

IPSEC implicitly offers a lot of *services* as:

- **Access Control:** prevents *unauthorized use* of resources by means of *authentication*;
- **Connectionless Integrity:** *data integrity* without having a *connection*, **IPSEC** is able to check changes to individual *IP datagrams*;
- **Data Origin Authentication:** it verifies the *identity* of the claimed source of *data*, providing *security* also against *replay attack*;
- **Confidentiality:** **IPSEC** can provide *confidentiality*, encryption of data, but it is optional;
- **Limited Traffic Flow Confidentiality:** prevents the *attacker* from analyzing *data* in order to retrieve some *information* about the *datagrams* incoming and outgoing our interface;

Like already told, we can use **ESP** with *auth* or without it, we can use both **AH** and **ESP**, or one of them, but often is not interesting combining *AH* and *ESP_a*:

	AH	ESP (encryption only)	ESP (encryption plus authentication)
Access control	✓	✓	✓
Connectionless integrity	✓		✓
Data origin authentication	✓		✓
Rejection of replayed packets	✓	✓	✓
Confidentiality		✓	✓
Limited traffic flow confidentiality		✓	✓

11.1 Security Associations (SA)

Security Association is a *one-way association* between *sender* and *receiver* that affords *security* for traffic flows. It is defining the parameters of sending *datagrams* from *Alice* to *Bob*, instead the message from *Bob* to *Alice* will be described by another association. All *security associations* are stored into a *database (SABD)*, and the *standard* states that for every *entity* (in particular for *every interface*) there must be implemented two *SAs*, one for *incoming traffic* and one for *outgoing traffic*. A **Security Association** is defined by 3 main parameters:

- **SPI (Security Parameter Index):** it is a pointer to information about the security you are going to use;
- **IP Destination Address;**
- **Security Protocol Identifier:** tells whether you are using ESP or AH (if you want both, you will need two SA);

If you want to have a **bidirectional traffic**, since a *SA* is *one-way*, of course you need two *Security Associations*. There are also other *parameters*:

- **Sequence Number Counter:** counter similar to the *TCP* one, it is used to generate *sequence numbers* for *AH* and *ESP headers*;
- **Sequence Number Overflow:** it is a *flag* saying whether the *counter* goes overflow;

- **Anti-Replay Window:** used to determine whether *traffic* is authentic or replayed;
- **AH Information:** what *authentication algorithm* is used, which is the *key* and so on;
- **ESP informations:** what are the *authentication* and the *encryption algorithms*, what is the *initialization value, keys* and so on;
- ...

It can suffer of **Fragmentation attack**, a typical *DoS attack*, in which I'm sending *data* fragmented in *datagrams*, you know that *fragments* have *headers* containing bits that say “*this is last frag*” and so on such that the *receiver* can reconstruct the order of *packets*. The concerned *attack* is based on faking the sequence number of *fragments*, so the *attacker* changes the offset or the bit that says whether the *fragment* is the last or not.

Security Policy Database, or **SPDB**, it is an important *database*, this is not the one collecting the *SAs*, each *entry* of such *database* is associated with a *set of IP* and a *set of Upper Layer protocol fields* that you may be using from above, such fields are called **Selectors**, then each *entry* is pointing to a *SA* for that type of *traffic*. **Selectors** are used for implementing *filters*, so you can analyze the pattern of the *traffic* against the *filter*. In order to realize these type of *filters* the process is like this one: compare the numbers you are reading in the *several fields* with the *pattern* you are storing in the *entries* of the *database*, and when you find a *match*, the match will tell you what is the *correct association* (if no match, it means bypass of *IPSEC*). So **SPDB** specifies the *policies* that determine the disposition of all *IP traffic inbound and outbound* from a *host* or a *security gateway*, instead **SADB** is a *security association table* that contains parameters of the *security associations*.

11.2 Transport & Tunnel Modes

IPSEC can be used in two different modes: **Transport Mode** and **Tunnel Mode**, they are two *modes of operation* and the difference is that one is more *secure* than the other, but since the most *secure* needs more in terms of *computational effort*, so we can decide which one we will use.

11.2.1 Transport Mode

The *original datagram*, with **IP header** and the **payload (packet)**, will not be touched (the *packet* is *protected*), so we add an *extra header*, the **IPSEC header**, between them. So the structure is:

| IP Header | IPsec | [packet] |

While you are sending the **packet** over the *internet* the many *routers* that are processing the *packet* they are seeing *datagram*, they will see only **IP header** and **payload** and process them as normal *fragments*, when the *packet* reaches the *destination*, the *destination* will process the **IPsec header**, this is **end-to-end protection**. Since the *original IP header* is not touched, the **routing path** is intact but this is not completely true because there are some exceptions (you can protect not only the *payload* but also some part of the *header*, this with *additional header*).

11.2.2 Tunnel Mode

Here the original payload and the IP header are encrypted, and they become the payload of a new datagram having a new IP header and the IPSEC with them, so the structure is:

| New IP Header | IPsec | [old IP Header | packet] |

Let's imagine that you want to connect to your *workstation* in your office from your *computer* at home in a *protected way*, then **Tunnel mode** is what you need because you will **encrypt** the full *datagram* (included *header* that will contain the *IP address* of your *workstation*), and in the new *header* you will put the *IP address* of the *gateway* of the *LAN* of your office, then when the *datagram* reaches the *gateway*, it will decrypt the *payload* and it will get from the *original header* the *IP address* of the *workstation*. With **tunnel mode** you are hiding in the *original IP header* the *IP/port number* of destination. Take care of the fact that in this mode the *final fragment* is **much bigger** than the *original* one and you cannot always afford such effort.

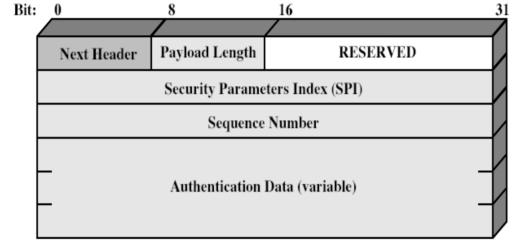
The **transport mode** is preferred in the cases of **end-to-end communications**. It means that you are connecting the two *hosts* by your *private link*, your *personal link*, without using the *internet*, so you do not need a *secure tunnel* as the other mode does. The **tunnel mode** is preferred for **network-to-network communications**, *host-to-network communication* and *private host-to-host communications* like a private chat. A question for the exam can be, since in *tunnel mode* we encrypt the *original header*, how we can be sure that the *datagram* will reach the *destination*? The answer is that the new *header* will contain the *address* of the *gateway* which implements *IPSEC*, so it knows that it will find in the *original header* the *address* of the computer in the *LAN* where the *gateway* is connected.

- **AH + Transport mode:**
 - You will be authenticating the *payload* and some portion of the *IP Header*;
- **AH + Tunnel mode:**
 - You will be authenticating the whole *original IP packet* and some portion of the new *header*;
- **ESP + Transport mode:**
 - You are encrypting the *original payload*;
- **ESP + Tunnel mode:**
 - You are encrypting entire *IP packet* (*original payload* and *IP header*);
- **ESP_a + Transport mode:**
 - You are encrypting the *payload* and you are authenticating the *payload*, not the *IP header*. This is the difference between ESP_a and AH.
- **ESP_a + Tunnel mode:**
 - You are encrypting and authenticate all the *original IP packet*;

11.3 Authentication Header AH

AH provides support for **data integrity** and **authentication** of *IP packets*, but it doesn't provide support for **confidentiality**. It is based on use of a *MAC* and users must share a *secret key*, the structure is the following:

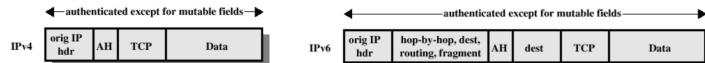
Next Header identifies the *higher-level protocol* that is using this *protocol*. Then there is the **SPI** (*Security Parameter Index*), if you remember we defined the *SPI* like something that was describing some *technical details* about the *keys* and other *techniques* used in order to implement the *cryptography* needed to ensure security, but the way such bits are interpreted is not officially defined, so you can setup your scheme. **Sequence Numbers** is self explaining. **Authentication Data** is explaining information about *authentication*, for example which *hashing function* and so on.



This *protocol* is protecting the *payload*, it applies the **MAC function** on the whole *payload*, but also some part of the *IP header*. In particular only the *non-mutable fields* (there are *mutable* and *non-mutable fields*, for example fields like the one that informs about *fragmentation* can vary packet to packet) are **authenticated** and **secured**. In this way AH can provide **data integrity** and **data origin authentication** (the *IP address* cannot be modified as this always invalidates the *hash value*). However, it is incompatible with *NAT*, because once the *packet* reaches the *gateway*, the *gateway* cannot change *IP of destination* in order to exchange it with the *private one* within its *network*, because this will invalidate the *authentication*.

AH operates on top of **IP**, using the *protocol number* 51. Therefore when the *packet* is going over the *internet* it is seen as a *normal datagram*, when a *generic router* receives a *packet* in which in the *IP Header* in the *IP Protocol field* there is the number 51 it means that the *router* will have to pass the *packet* to the *AH layer* upperlying it, which will extract the *AH header* and that can process the *real payload*.

- When AH is used on *packets* with **Transport Mode**, the outcome is a *packet* that has an *original IP header*, the *AH header* and finally the *payload* (which is composed by *TCP header* and the *Data*):



- When AH is used in the **Tunnel Mode**, the *original datagram* (*original IP header*, *TCP header* and *Data*) becomes the new *payload*, then there is the **New IP header**, and between them there is the *AH header*.

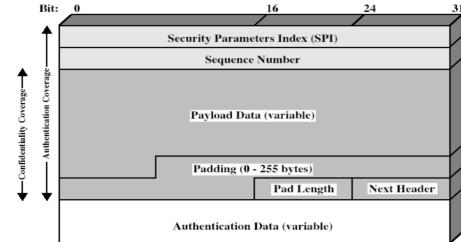


What happens is that here the whole **original datagram** is *protected*, so also the *mutable fields* of the *IP header*. But then there is the *New IP header* for which only the *non-mutable fields* are protected. So it can be used in such a way that I specify as destination in the *new IP header* the *IP address* of the *gateway* of my enterprise, and once the *gateway* checks the *integrity* of the *whole packet*, it can remove the *new IP header* and the *AH header* and use normally the *original packet* within the *private network* of the enterprise.

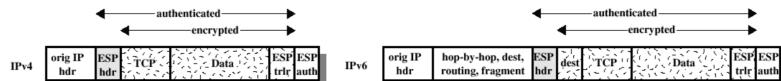
11.4 Encapsulating Security Payload ESP

ESP is providing **encryption**, it is supporting all modern *encrypting algorithms* like *AES*, *Triple-DES* and so on, and often it is using *CBC*. If you are using *ESP_a*, the difference from *AH* is that when you use the *authentication feature* of *ESP* the *authentication/data integrity* is made only on the *payload*, not even the *non-mutable fields* of the *IP header* are included. This allows the *ESP* to be compatible with **NAT**. In some cases *ESP* provides also limited *traffic flow confidentiality*, traffic flow is something an *attacker* is interested to understand, he wants to see the type of traffic outgoing your computer, you want to avoid this so you need *traffic flow confidentiality*. The structure is:

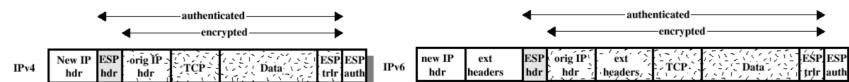
Again you can see *SPI* and *Sequence Number*, then there is some *variable-length Data* and the *padding* after it, then we find the *pad length* and the *next header* which describes what is the *protocol* that will elaborate these *packets* at higher level and at the end there is the *Authentication Data*.



- When *ESP* is used in **Transport Mode**, then the whole *payload* and the *trailer* is encrypted, and we have the *original IP header*, the *ESP header*, the *payload* and the *ESP authentication* portion in case of *ESP_a*. The *ESP* wraps the *payload* between an *header* and a *trailer*:



- When *ESP* is used in **Tunnel Mode**, the whole *original datagram* is *encrypted*, and so the final structure will be composed by *New IP Header*, *ESP Header*, *payload* and *ESP auth* portion in case of *ESP_a*:



The difference between **Transport mode ESP** and **Tunnel mode ESP** here is that with *Transport mode* we are encrypting only the *payload*, so the whole *path* of the *data* is visible, what is not

visible is the *information* that our *IP packet* brings, while with *Tunnel mode* we are also encrypting the *header*, so the *real destination* and informations as *offsets* and *message length* will not be visible.

11.5 Combining Security Associations: SAs

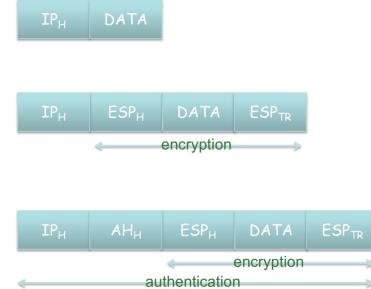
SAs can implement either **AH** or **ESP**, but if you want to implement both of them you need to combine two **SAs** and the result of this combination is called **security bundle**. There are two types of *security bundle*:

- **Transport Adjacency**: we apply more than one *protocol* to the same *IP packet* without *tunneling*, but only with one level of combination;
- **Iterating Tunneling**: we apply multiple levels of *security* with multiple *tunneling*;

These two methods can be also be combined. **Encryption** and **authentication** can be combined in order to transmit a *packet* that has both **confidentiality** and **authentication** with several approaches: **ESP with authentication** (**ESP_a**), **Transport adjacency** and **Transport-tunnel bundle**. The simplest case is **ESP_a**, in which we apply ESP to data to be protected and after we append the *authentication data field*, and we can have two sub-cases (for both cases, *auth* applied to the *ciphertext*, so it happens after *encryption*): **Transport Mode ESP** in which *auth* and *encryption* are applied to *IP payload*, but *IP header* is *not protected*, and **Tunnel Mode ESP** in which *auth* and *encryption* applied to the entire *IP packet*;

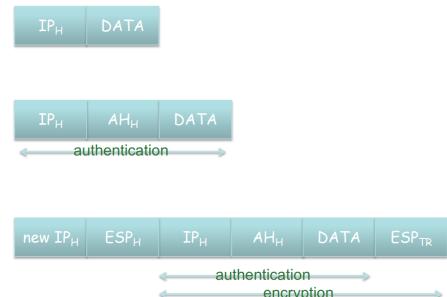
11.5.1 Transport Adjacency

Here we are still assuming we want *authentication after encryption*. This is made by combining **ESP** and **AH**, so we combine two **SAs**. The **inner SA is ESP** (no *authentication*), the **outer SA is AH**. This mean that the *encryption* is applied to the *IP payload* and so it is added an *ESP header*, and then all the result is authenticated, so an *AH header* is added. The advantage is that with this we are *encrypting the data* and we are *authenticating the data* but also some *fields of the header* like source and destination IP addresses.



11.5.2 Transport-Tunnel bundle

Here the *authentication* is applied **before encryption**, so it's impossible to alter *authentication* data without *decryption*. So, we have the *authentication* applied to the entire *IP packet* through an **inner AH transport SA** (an *AH header* is added), and then we *encrypt* the whole out-coming *IP packet* by an **outer ESP tunnel SA**, so an *ESP header* is added (and also a new *IP header*).



At one *exam* it has been asked the following question, what **security association** are best for the following cases:

- *Bob* at home wants to print a pdf document on the LaserPrinter located in the *private network* of his corporate:
 - In this case Bob needs only to connect to the *Gateway* of the *corporate* and he needs a **tunnel** of course (*host-to-network communication*), then the *packet* can travel in the *private network* of the corporate in an *unprotected way*. Indeed *Bob* can just ask for *ESP_a* in *tunnel mode* because (*ESP* guarantees that the *document* is *encrypted* till it reaches the *gateway*, and the *authentication* feature of *ESP* guarantees that the *encrypted document* will not be *modified*, then once the document reaches the *gateway* it can be sent in clear to *Bob's printer* since it will travel within the *private network* of Bob's corporate);
- *Alice* the boss is at home, and needs to save into her Personal Computer in the office a *document* that is **confidential**:
 - She needs more *SA*, so once the *packet* reaches the *gateway* and the *gateway* extracts the *payload*, this outcome should be **still protected**, so she needs a *SA* between *her* and the *gateway* and a *SA* between the *gateway* and her *computer* in the office, therefore she needs **tunneling ESP** for the first part (*Alice* → *Gateway*) and **ESP** for the *second path* (*Gateway* → *PC*), so she needs two level of **encryption**.

12 SSL/TLS

Now we will introduce a layer between *TCP* and *Application layer* (watch figure at page 58), this is **TLS**. **TSL** and **SSL** (its predecessor) are *cryptographic protocols* that provide *security* for *communications* over *networks*. In *TCP* the *sender* and the *receiver* are **ports**, *port numbers*. When an *application* is working and it is using the *network* a *port* is associated to it. With *IPSEC* we provided *security* from *port to port*, now we are securing a specific *application*, so a specific *port*. In most cases we can prefer *TLS* over *IPSEC*, because *IPSEC* is a way to *secure* all the *communications* between *hosts*, *TLS* secures a specific *application*. *TLS* is not invisible to applications, instead *IPSEC* is.

Another way of describing this is **end-to-end security**, where *end* has not a formal definition, it depends on the cases, we often mean a *party* that is *communicating*. *Whatsapp* is offering *end-to-end communication*, it means that the *Server* is not able to understand the *message*, since they are *encrypted*. When we find **HTTPS** in our browser when we are surfing on a certain *site*, it means that the *connection* between us and that site is *secure* because it relies on *TLS*.

TLS is designed to work under the *web browsing*, of course you can use it for other *applications* but the *main use* it is for *browsing*. **TLS** is providing **confidentiality**, **data integrity** and **authentication**, your *web browser* will be sure about the identity of the *web server* it is communicating with. But *TLS authentication* is **unilateral**, since only *Server* is *authenticated*. In fact **mutual authentication** requires to install a **certificate** on your browser, so the *server* can authenticate you.

There can be two cases:

- *Client* and *Server* exchange *keys* according to a *Private/Public Key*;
- *Client* and *Server* use a *pre-shared secret key*, this is also called **TSK-PSK**;

When we make a *connection* through *TLS* we have what is called **TLS Handshaking**, we will see in details it later, but the main goal is the following:

1. The *peer* starts a *conversation* for understanding what *algorithms* *client* and *server* can both support, then they agree with some *algorithms* to be used;
2. Then they do **key exchange** and **authentication**;
3. After that they can start to run *normal conversation, encrypted* of course with the chosen **symmetric cipher** and **messages** are also *authenticated*;

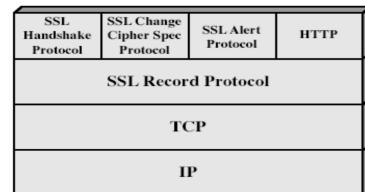
At the very beginning, *client* and *server* start talking about what **ciphers** they know. The *server* has an ordered list of choices (ordered by *security criteria*), and from the known *algorithms* *client-side* the *Server* chooses the best one. You can also use *authentication*, and after *authentication* you can derive **session keys**, if it is not based on *public keys* you can base it on some *shared secret*, like is the case of *PSK*.

The **handshaking** of the *TLS protocol* is the most *complex* part, it is the part that may fail, and if it fails the *connection* is not established. *Handshake* begins when *client* connects to *TLS-enabled server* requesting a **secure connection** and presents a list of *supported ciphers* and *hash functions*, then from this list the *server* will choose the strongest one, and after will notify the *client* with its decision, and will send back its *identification* in the form of a *digital certificate X.509*, that the client can check the *authenticity* by contacting a *CA*.

The *client* will generate a **session key** by creating a *random number RN*, encrypted with *server public key K_S* and will send the result to the *server*, that is the only one able to *decrypt* it with its own *private key* (so only *server* and *client* have access to this *data*). A *third party* may only know the *public key* of the *server*, but he doesn't know the *random number* from which both parties will generate *key material* for *encryption* and *decryption*. This concludes the **handshake** and begins the **secured connection**.

12.1 SSL Architecture

According to the *original design (SSL)*, the *protocol* was based on two different *layers*: the *first layer* is the one which performs the *setup* of the *connection*, so in this *phase* all the *useful parameters* in order to **secure** the *conversation* are *exchanged*. The other layer is the **Record Protocol**, which acts after the *setup session*, in which we actually *encrypt* and *decrypt* the *real data* of the *conversation*.



- **SSL Session:**

- Is the result of an *handshaking*, the two *parties* they establish some *parameters*, and all such *parameters* describe a *session*, now within the *session* you can establish some several *connections*, using the same *parameters* (so it is **stateful**). We reuse *parameters* cause you do not want to start a new *handshaking* for every *connection*. So an *SSL session* is a **logical association** between the *client* and the *server*;

- **SSL Connection:**

- Just an exchange of messages within a session (stateless) so more connections can take place in one single session and it's made to save some computation.

So it's important to note that *sessions* \neq *connections* in fact, a session made by handshaking can have *multiple connections* (*peer-2-peer*) which one persistence *session*. Now a *svario di parametri che non ricorderò mai quindi pisciati*.

The most *important protocol* offering *encryption*, *confidentiality* and *authentication* and *data integrity* is the **Record Protocol** which have two main services:

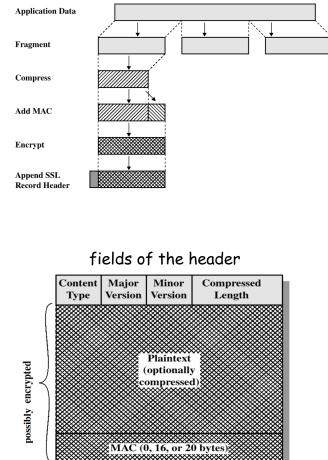
- **Confidentiality:** obtained by using symmetric encryption with a shared secret key defined by handshake by using encryption algorithms like: *IDEA*, *RC2-40*, *DES-40*, *DES*, *3DES*, *Fortezza*, *RC4-40*, *RC4-128*, and the message is compressed before encryption;
- **Message Integrity:** obtained by using a MAC with shared secret key, similar to HMAC but with padding;

This is the scheme of **Record Protocol** which takes place after the *Handshaking phase* so once all the *MACs* and *Cipher Specs* have been decided. As we can see, the *application data* is *compressed*, *authenticated* and *encrypted*, and the the *SSL Record Header* is appended to the *encrypted fragment*.

This instead is a **single record** of the *Record Protocol*. The last part of the *record*, the *MAC*, is optional and it depends on whether *data integrity* is being used or not. *Content type* is defining what is the *upper protocol* asking the *service* to the *record protocol*, so it defines the *protocol* towards which this *record* is sent. *Major* and *Minor version* are specifying the *upper and lower versions* of *SSL/TLS*. *Compressed Length* is specifying the *length* of the *compressed record*. Then follows the **encrypted part**, which is the *Plaintext* and the optional *MAC*.

The **SSL payload** is the normal *information* transferred by the *protocol*. There are 4 kinds of payload:

- **Change Cipher Spec Protocol:** it is just one byte, it acts like a **signal** saying “*ok, we are done with our handshaking, now let's start the game by encrypting/decrypting and authenticating messages*”;



- **Alert Protocol:** composed by two bytes, used to specify alerts like *fatal* or *warning*;
- **Handshake Protocol:** one byte of *Type*, three bytes of *Length* and then the *content*;
- **Other Upper-Layer Protocol:** one byte of *Opaque Content*;

12.2 Handshake Protocol

The **Handshake Protocol** is the most important part of the *TLS protocol*. It is complex and somewhat long, indeed when people, normal users, are connecting to a *TLS enabled website*, they can feel a small *delay* that is related to the *handshaking phase*. The *handshaking phase* has 4 different phases:

- **Hello phase:**
 - In this phase the *Client presents* itself to the *Server*, giving to him some parameters:
 - * **Version:** last supported version of *SSL/TLS* supported by the *client*;
 - * **Client Random:** 32-byte (256-bit), composed by 32 bit of *timestamp* and 28 bytes *random*. The whole 32-byte are representing the **Client's Master Secret**.
 - * **Session ID:** if 0 it is a *new session*, if $\neq 0$ means we are resuming some old *connection* or to open a *connection* within an *existing session*;
 - * **Cipher suite:** list of *algorithms* supported by *client*, in decreasing preference order;
 - * **Compression algorithms:** list of acceptable algorithms;
- **Server Hello:**
 - In this phase the *Server* responds to the *Client*, deciding for him the *algorithms* to be used. If *client* and *Server* have no **matching ciphers** (they have no *ciphers* in common) then a **handshake failure alert** is raised. Then the *server* is sending informations to the client offering him its *digital certificates*, and informations for the *Server Key Exchange*, all this in order to construct the *shared key*. The *Server* may also ask the *client* for a *certificate*.
- **Client offers informations:**
 - The *client* answers for the *Servers* requests of the previous phase, so he offers it the *requested certificate*, and completes the **Key Exchange** (so sends the **Client Key Exchange message**). Then the *Client* confirms the end of its *handshake session*.
- **End of handshake:**
 - Also the *server* confirms the end of its *handshake session*;

12.2.1 Fixed Diffie-Hellman

Remind that the original **Diffie-Hellman** is vulnerable to **man-in-the-middle attack** if the *parties* are not *authenticated*. In the case of the *Fixed Diffie-Hellman* what happens is that the *server's certificate* contains *DH public parameters* signed by the *CA*, and the *client* provides its *DH public key parameters* with a *certificate* (if *client authentication* is required) or in a *key exchange message*. This method results in a **fixed secret key** between two *peers*, based on the *DH calculation* using the *fixed public keys*.

12.2.2 Ephemeral Diffie-Hellman

Used to create **one-time secret keys**, and the *DH public keys* are exchanged signed using *Sender's private RSA or DSS key*. The *receiver* can use the corresponding *public key* in order to verify the *signature*, and *certificates* are used to authenticate the *public keys*. This can be considered the most secure one since the *authenticated key* is *temporary*.

12.2.3 Anonymous Diffie-Hellman

It's the base **DH algorithm**, so **without authentication**, each *party* will send its *public DH parameters* to the other, but this approach is vulnerable to *man-in-the-middle attack* so it is not allowed anymore since TLS 1.2.

12.3 Attacks against TLS

12.3.1 Downgrade Attack

An *adversary* cheats the *server* and other *clients* in order to use an **older protocol**, so in this way he can use *older algorithms* or *parameters* that he can manipulate and that are not more *secure*, in such a way that he can modify the *traffic flow* in order to accomplish a **man-in-the-middle**.

12.3.2 Heart Bleed

Heartbeat was an *extension* of the *TLS protocol*, that offers a way to keep alive *secure communication* without need to restart the *connection* each time based on exchange of *messages*. But the *older version* allocate a **memory buffer** (on *server-side*) for the *message* to be returned based on the *length field* in the *requesting message*, without regard to the actual size of the *payload*, so an *adversary* cheats on the *length field*, so he will receive a huge amount of *data* from the *secure server*.

13 Firewalls

Firewalls have nothing to do with *cryptography*, they offer a different kind of *security*. The idea is to separate the *local network* from the *internet* in such a way that every *data* entering the *local network* will be **analyzed** by the *firewall*. Commonly it is the *incoming data* that has to be analyzed, *packets outgoing the network* are not so important, but actually also they can be analyzed, because some patterns can be recognized.

The *local network* can be made such that the *firewall* affects all the *traffic* coming from the *outside world* except for the so called **Demilitarized Zone (DMZ)**, which is a zone in our *local network* accessible from *outside*, and it is useful when our *local network* provides some *Server* (which cannot be reached if a *firewall* is placed between it and the *internet*). **Firewalls** can be applied more than one *firewall*, a common behavior is to have one general *firewall* for the *whole network*, and other special *firewalls* for certain *machines*, so *PCs* can have their *personal firewall*.

Firewalls can be categorized in 3 different *types* each one working at a different *level* of the *network*. Of course the *higher* is the *level*, the higher is the *security* offered by the *firewall* (we

will see why with **fragmentation attack**). In fact a *firewall* that is working at a *low level* it will inspect *single packets*, but it will not be able to recognize *malicious data* at *application level*:

- **Packet Filtering:**

- In which **filtering** is done by inspecting *headers* of the *packet* (even *payloads* in some cases), usually *stateless*, sometimes *stateful*;

- **Proxy Gateway:**

- The *firewall* acts like a *a gateway*, in the sense that all *incoming traffic* is directed to the *firewall* and all *outgoing traffic* appears to come from the *firewall*. It can be categorized in:

- * **Circuit-level:** the *firewall* works at lower level, so it is invisible to *application level*. Splices and relays two *TCP connections* in such a way that only the *validity* of the *connection* is checked (through a *table*) and not the *content* of it;
 - * **Application-level:** in which every *application* has its own *proxy firewall*. Splices and relays *app-specific connections* like the one of *browsers*, and has a high level of *control* wrt the *circuit level*, but the *overhead* is higher;

- **Personal Firewall:**

- In which *firewall* it's set on the *single PC*;

13.1 Packet Filtering

We will focus only on **Packet Filtering Firewall** in which for each *packet*, the *firewall* decides whether the *packet* has to proceed or not. An interesting thing about *firewalls* is the mode of operating: **stateless** or **stateful**. *Stateless* means every *packet* is analyzed one **independently** from the other, *Stateful* means the *firewall* maintains some information about *packets* transiting through it. Usually the *header* of the *packet* is analyzed, in order to inspect the *IP source*, but it can be possible that further analysis is needed, so the *firewall* may inspect the *payload* of the *packet* in order to get the *TCP port* and take further decisions. So resuming the **Packet Filtering** is based on filtering **IP numbers** and **TCP ports**.

Being **Stateful** means that the *firewall* keeps *information* about some session, for example a *TCP session*. In such a way the *firewall* is able to understand what every *packet* means within that *session*, therefore we can apply a kind of *session filtering*. It will mean that the *firewall* takes different *decisions* depending on the particular *packet* in the context of a session, of course such a *firewall* must be placed at an higher level with respect to *IP level*.

An example of **filtering packets** is for example analyzing all *packets* coming from 192.168.4.* where '*' means any number. If we know a certain *subnet* is **dangerous**, let's say the *subnet* 10.25.68.0/24, we can add a *rule*, write something like "10.25.68.* BLOCK" in order to *block* all the *traffic* coming from that *subnet*. Of course we can specify further details, for example we can block specific **IP address** together with a *specific port*.

However **Packet Filtering** is vulnerable to **IP Spoofing** since the *attacker* can change its *IP*

bypassing the rules of the *firewall*. A well known attack against *firewalls* is **Fragmentation Attack**, the idea is simple, fragment a *packet* into two *sub-packets* such that they seem harmless alone, but once *reassembled* on the *upper layer* they constitute a *malicious fragment*. Also **DoS attacks** can be performed, like the one against the *3-way handshaking* of *TCP* and the *packet filtering firewall* will not be able to recognize such an *attack*.

13.2 IPtables

In *Linux distributions* there is a useful tool that provides *firewall* features, we are talking about **IPtables**. It performs **stateful** analysis on *session filtering*, and performs also other features like *NAT tables*, but it works at **packet level**, not *application level*. An important concept of *IPtables* is the concept of **chain**: a *chain* is a list of **rules**, processed in a *sequential way*, which determine whether a *packet* has to be *accepted* or *dropped*. We will focus only on three different kind of chains: **Input**, **Forward** and **Output**. The meaning is the following:

- A *chain* called **Input** is a list of *rules* that consider *incoming packets*;
- A *chain* called **Forward** is a list of *rules* that consider all those *packets* that were *routed* to us but are not to be *delivered* for us so they just have to be *forwarded*;
- A *chain* called **Output** is just the *converse* of the *Input chain*, so it deals with *outgoing packets*;

When we are implementing a **personal firewall** then *Input* and *Output chains* are meaningful, instead *Forward chain* is not important. On the contrary, when *IPtables* is working on a **gateway**, then *Forward chain* is the most important.

Working with *IPtables*, a *decision* associated to a specific *rule* is called **target**, that specifies what to do if that *packet* has matched with that *specific rule*. The most common targets are **accept**, **drop**, **queue**, **return**. *Accept*, let the *packet* through, *Drop*, drop the *packet*, *Queue*, pass the *packet* to *userspace*, *Return*, stop with this *chain* and resume the next *rule* in the previous *calling chain*. Of course **custom targets** can be written and it will be implemented as a *sub-chain*, so other *rules* with other *targets*. Here the *return target* makes sense, where it means that we have to *come back* to the *upper chain* and *continue analyzing*. It is like a *custom target* is a sort of routine that we write.

Then there is the concept of **external module**, that can be *imported* for implementing *filters* for a *specific protocol*, and we can change *rules* within the *module*. This because of course it requires too much effort to configure *IPtables* from scratch, so we can rely on *built patterns* and then we modify them according to our *preferences*.

13.2.1 Examples of IPtables

First of all, let's see the **command line** usage of *IPtables*:

-A	Used to <i>append</i> a chain-rule specification (es. -A INPUT, -A OUTPUT, -A FORWARD)
-p	Specifies the protocol (es. -p TCP, -p UDP, ...)
-i	Specifies the input interface (es. -i eth0)
-o	Specifies the output interface (es. -o eth1)
-s	Specifies the source address (es. -s 192.168.1.104)
-d	Specifies the destination address(es. 192.168.1.124)
--sport	Specifies the range of source ports (es.--sport 1024:65535)
--dport	Specifies the range of destination ports (es. --dport 1024:65535)
-j	Specifies the target (es. -j ACCEPT, -j REJECT, -j DROP)

- **Example 1:**

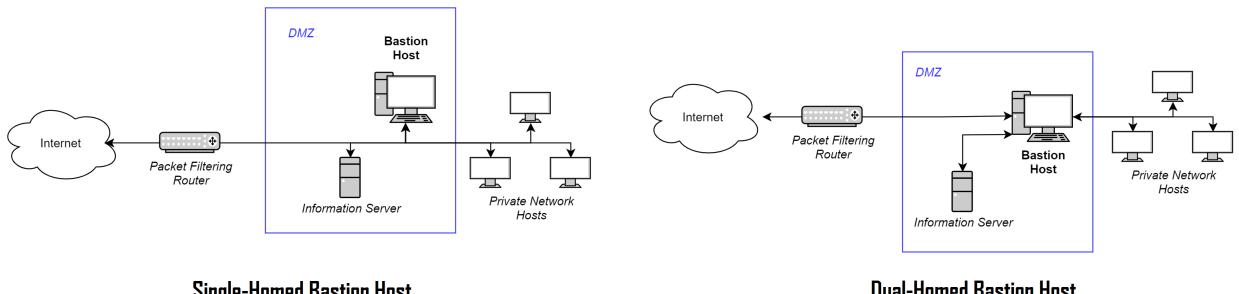
- Let's assume that *eth0* is the interface of a *router* towards public internet, we need to **block** all *incoming traffic* from *eth0*:
- > **iptables -A FORWARD -i eth0 -j DROP;**
- So we have specified to add another rule with -A, the chain is a Forward, and specifying the *interface* *eth0* with -i and specifying to Drop all the packets on this interface. With Drop we will not send back any response;

- **Example 2:**

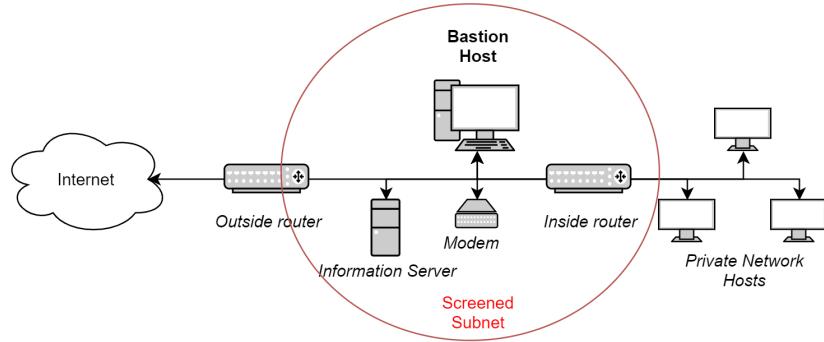
- Accept *packet* from *outside* if they refer to a **TCP connection** started within the *network*:
- > **iptables -A FORWARD -i eth0 -m state --state ESTABLISHED -j ACCEPT;**
- So we have specified to add another rule with -A, the chain is a Forward, and specifying the *interface* *eth0* with -m state --state established that allows to decide whether the connection originated from the inside or outside;

13.3 Bastion Host

Bastion Host is a *hardened system* implementing **application level gateway** (so need a *proxy firewall* for each *application*) behind *packet filtering*. It uses just *trustable operating systems*, and runs only few *applications* (so all other *services* non-essential are turned off). So each *proxy* will support only a *subset* of *application's command*, and the *traffic* is analyzed (in order to check if there are any *attacks*), and also the *disk access* is *restricted*. All *traffic flows* through the *bastion host*, the *packet router* will allow only *external packets* to enter only if their *destination* is the *bastion* and *internal packets* to leave only if the *origin* is the *bastion*.



On the left there is the **Single-Homed Bastion Host** in which *external connections or users* can reach the *servers* but they cannot reach the *internal network* because it is blocked by the *Bastion Host*. It's important to note that in this case if the *packet filter* is compromised then *traffic* can flow to *internal network*. On the right there is called the **Dual-Homed Bastion Host** the difference is that there aren't *physical connection* between *internal and external server*.



Another solution is **Screened Subnet**, in which only the *screened subnet* is visible to the *external network*, the *internal network* is invisible. We can see that we have another *firewall* between *bastion host* and the *private hosts*. Great attention should be dedicated to the *traffic* that enters in the *DMZ* can if an *attacker* control the *bastion host* then he can enter the *internal LAN*.