

# Cybersecurity

Leonardo Fontanelli

November, 2020

## Abstract

Queste dispense sono state prodotte a partire dalle note di Francesco Barbarulo e Antonio Le Caldare. Si riferiscono al programma dell'anno accademico 2018/2019. È stato realizzato un merge delle suddette dispense e delle slide del professor Dini. È disponibile la repository Git del tex file a questo indirizzo: <https://github.com/fontanellileonardo/CYBERSECURITY-Notes> ed è possibile forkarle e integrarle con del materiale aggiuntivo per i successivi anni. Secondariamente, le appendici (che corrispondono alla parte di laboratorio) non sono state toccate rispetto alla versione iniziale delle dispense e lasciate in lingua Inglese.

## Contents

<b>1 Symmetric Encryption</b>	<b>1</b>
1.1 Principio di Kerchoff . . . . .	1
1.2 Definizione di Cipher . . . . .	1
1.3 Sicurezza di un cipher (definizione informale) . . . . .	1
1.4 Shannon . . . . .	1
1.4.1 Perfect secrecy . . . . .	1
1.4.2 Teorema di Shannon . . . . .	2
1.5 One-Time Pad . . . . .	2
1.5.1 Proprietà . . . . .	3
1.5.2 Proprietà dello XOR . . . . .	3
1.5.3 Proprietà di OTP . . . . .	3
1.5.4 Problemi di OTP . . . . .	3
1.6 Stream ciphers . . . . .	4
1.6.1 Sicurezza computazionale . . . . .	4
1.6.2 Pseudo-random number generator . . . . .	5
1.7 802.11b WEP . . . . .	5
1.8 Block ciphers . . . . .	6
1.8.1 Computational security vs Attack complexity . . . . .	7
1.8.2 Exhaustive key search . . . . .	7
1.8.3 Confusion and diffusion . . . . .	8
1.8.4 Data Encryption Standard (DES) . . . . .	8
1.9 Note sulla Criptoanalisi . . . . .	12
1.10 Encryption Modes . . . . .	13
1.10.1 Electronic Codebook (ECB) . . . . .	13
1.10.2 Cipherblock Chaining (CBC) . . . . .	14
1.10.3 Cipher Feedback (CFB) . . . . .	14
1.10.4 Output Feedback (OFB) . . . . .	14

1.10.5 Counter Mode (CTR) . . . . .	15
1.10.6 Ciphertext Stealing Mode (CTS) . . . . .	15
1.10.7 Padding (PKCS #7) . . . . .	15
1.11 Advanced Encryption Standard (AES) . . . . .	15
<b>2 Hash function</b>	<b>17</b>
2.1 Proprietà informali . . . . .	17
2.2 Proprietà di sicurezza . . . . .	17
2.3 Attacchi alle Funzioni Hash . . . . .	18
2.3.1 Black box attacks . . . . .	18
2.4 Costruzione di una Funzione Hash . . . . .	20
2.4.1 Schema Merkle-Damgard . . . . .	20
<b>3 Message Authentication Codes (MACs)</b>	<b>21</b>
3.1 Algoritmo di generazione MAC . . . . .	22
3.2 MACs from hash function . . . . .	23
3.3 Come costruire le funzioni MAC . . . . .	23
<b>4 Side Channel Attack</b>	<b>26</b>
<b>5 Key establishment</b>	<b>28</b>
5.1 Pairwise Keys . . . . .	28
5.2 Trusted Third Party - TTP . . . . .	29
5.3 Diffie-Hellman . . . . .	30
5.3.1 Descrizione del protocollo nel dettaglio . . . . .	31
5.3.2 Attacchi al protocollo . . . . .	33
5.3.3 Active attack . . . . .	33
<b>6 Public key encryption</b>	<b>34</b>
6.1 Security properties . . . . .	34
<b>7 RSA</b>	<b>35</b>
7.1 Key generation algorithm G . . . . .	35
7.2 Encryption E . . . . .	36
7.3 Decryption D . . . . .	36
7.4 Square-and-multiply algorithm . . . . .	36
7.5 RSA problem . . . . .	36
7.5.1 RSAP . . . . .	36
7.6 RSA security . . . . .	37
7.6.1 Small plaintext problem . . . . .	37
7.6.2 Malleability . . . . .	37
7.6.3 Low exponent attack . . . . .	37
7.6.4 Common modulus attack . . . . .	38
7.6.5 Chosen-plaintext attack . . . . .	38
7.6.6 Adaptive chosen-plaintext attack . . . . .	38
7.6.7 Countermeasure . . . . .	39
<b>8 Digital signature</b>	<b>40</b>
8.1 Schema generale . . . . .	41
8.2 Schema di firma digitale attraverso RSA . . . . .	41
8.2.1 Attacchi all'algoritmo . . . . .	42
8.3 Proprietà della funzione hash . . . . .	42
8.4 Non ripudiabilità vs autenticità . . . . .	43

<b>9 Certificate</b>	<b>44</b>
9.1 Certification Authority (CA) . . . . .	44
9.1.1 Generazione di un certificato . . . . .	45
9.1.2 Threshold crypto . . . . .	45
9.2 Verifica di un certificato . . . . .	46
9.3 Certificate revocation . . . . .	46
9.4 Campi del certificato . . . . .	47
9.5 Diffie-Hellman MITM solution . . . . .	47
9.6 CA delegation . . . . .	47
9.7 Trust model . . . . .	47
9.7.1 Modello centralizzato . . . . .	47
9.7.2 Modello enterprise - cross-certification . . . . .	48
9.7.3 Modello browser web . . . . .	48
9.7.4 Modello PGP . . . . .	49
<b>10 Perfect Forward Secrecy (PFS)</b>	<b>50</b>
10.1 Pre-Shared Key Ephemeral DH . . . . .	50
10.2 Ephemeral RSA . . . . .	50
10.3 Station-To-Station (STS) protocol . . . . .	51
<b>11 BAN logic</b>	<b>52</b>
11.1 Formalismi . . . . .	52
11.2 Concetti preliminari . . . . .	52
11.3 Postulate 1: message meaning rule . . . . .	52
11.4 Postulate 2: Nonce verification rule . . . . .	53
11.5 Postulate 3: Jurisdiction rule . . . . .	53
11.6 Objectives . . . . .	53
<b>12 Kerberos</b>	<b>54</b>
12.1 Protocol . . . . .	54
12.2 Ticket Granting Server (TGS) . . . . .	54
12.3 Delegation problem . . . . .	55
12.3.1 Proxy tickets . . . . .	55
12.3.2 Forwardable TGT . . . . .	56
12.3.3 Referral TGT . . . . .	56
<b>A Buffer overflow</b>	<b>57</b>
A.1 Countermeasures . . . . .	57
A.1.1 Data Execution Prevention . . . . .	57
A.1.2 Address Space Layout Randomization . . . . .	58
A.1.3 Stack canaries . . . . .	58
<b>B Secure coding</b>	<b>60</b>
B.1 Undefined behavior . . . . .	60
B.2 Taint analysis . . . . .	60
B.3 Sanitization . . . . .	60
B.4 Strings . . . . .	61
B.5 Pointer subterfuge . . . . .	61
B.6 Dynamic memory management . . . . .	61
B.7 Vector iterators . . . . .	61
B.8 OS command injection . . . . .	62
B.9 TOCTOU race condition . . . . .	62

B.10 Integers . . . . .	63
B.10.1 Unsigned sanitization . . . . .	63
B.10.2 Signed sanitization . . . . .	63
<b>C Network security</b>	<b>65</b>
C.1 Penetration test . . . . .	65
C.2 Exploitation . . . . .	65
C.3 Network scanning . . . . .	65
C.3.1 SYN scanning . . . . .	65
C.3.2 UDP scanning . . . . .	65
C.3.3 Service scanning . . . . .	66
C.4 TCP/IP stack fingerprint . . . . .	66
C.5 Network sniffing . . . . .	66
C.6 Fuzzing . . . . .	66
<b>D Malware</b>	<b>68</b>
D.1 Common features . . . . .	68
D.1.1 Infection and self-replication . . . . .	69
D.1.2 Persistency . . . . .	69
D.1.3 Concealment . . . . .	69
D.1.4 Damage . . . . .	70
D.2 Anti-Virus software . . . . .	70
D.2.1 Fred Cohen's result . . . . .	70
D.2.2 Signature-based detection . . . . .	71
D.2.3 Sandbox-based detection . . . . .	71
D.3 Reverse engineering . . . . .	72
D.3.1 Decompilation . . . . .	72
D.3.2 Obfuscation . . . . .	73
<b>E Web security</b>	<b>74</b>
E.1 Web application mapping . . . . .	74
E.2 Authentication . . . . .	75
E.2.1 Strong credentials and password change . . . . .	75
E.2.2 Secure credential transmission . . . . .	75
E.2.3 Secure credential storage . . . . .	75
E.2.4 Fail-Open Flaws . . . . .	76
E.2.5 Verbose error messages . . . . .	76
E.2.6 Account locking and CAPTCHAs . . . . .	76
E.2.7 Account recovery . . . . .	76
E.3 Session management . . . . .	77
E.3.1 Secure token transmission . . . . .	77
E.3.2 Session termination . . . . .	77
E.4 Code injection . . . . .	77
E.4.1 Tautology . . . . .	78
E.4.2 Finding SQLI bugs . . . . .	78
E.4.3 UNION operator . . . . .	79
E.4.4 Using inference . . . . .	79
E.4.5 Piggybacked query . . . . .	80
E.4.6 Countermeasures . . . . .	80
E.4.7 Injection vectors . . . . .	80
E.4.8 LDAP injection . . . . .	80
E.5 Cross-site scripting (XSS) . . . . .	81

# 1 Symmetric Encryption

## 1.1 Principio di Kerchoff

Un sistema di criptazione deve essere sicuro finché non viene desecretata la chiave. Il sistema deve basarsi solo sulla segretezza della chiave.

## 1.2 Definizione di Cipher

Un cipher definito su:  $k$  insieme delle chiavi,  $P$  insieme del plaintext,  $C$  insieme dei ciphertext e una coppia di "efficienti" algoritmi  $(E, D)$  tali per cui:

$$E : P \times K \rightarrow C$$

L'algoritmo  $E$  prende come input il plaintext (testo in chiaro) e una chiave  $k$  per produrre un Ciphertext  $c$  (testo cifrato).

$$D : C \times K \rightarrow P$$

L'algoritmo  $D$  prende come input il Ciphertext  $c$  e la chiave  $k$  per riottenere il Plaintext originale.

$E$  può essere randomizzata,  $D$  è sempre deterministica.

Da queste definizioni, possiamo definire un'equazione di consistenza:

$$\forall p \in P, k \in K : D(E(k, p)) = p$$

## 1.3 Sicurezza di un cipher (definizione informale)

Un cfrario simmetrico è sicuro *se e solo se* per ogni coppia  $(p, c)$ :

1. Dato  $c$ , è "difficile" determinare  $p$  senza conoscere  $k$  e viceversa. (**cipher-text only attack**)
2. Dato  $c$  e  $p$  è "difficile" determinare  $k$ , anche se è utilizzata solo una volta. (**known plaintext attack**)
3. L'avversario riesce a determinare  $c$  partendo da un plaintext  $p$  arbitrario. (**choose-n-plaintext attack**)

Per *difficile* si intende che non esiste un algoritmo efficiente in grado di trovare una soluzione in un tempo accettabile.

## 1.4 Shannon

L'idea di Shannon era che il ciphertext non doveva rivelare nessuna informazione sul plaintext.

### 1.4.1 Perfect secrecy

Uno schema di criptazione  $(E, D)$  ha la Perfect Secrecy se

$$\forall p \in P, c \in C : Pr(P = p | C = c) = Pr(P = p)$$

In pratica, conoscere il ciphertext (associato al plaintext) non aumenta le probabilità all'avversario di risalire a  $p$ , perché la distribuzione di  $P$  rimane la stessa.

Un cfrario, quindi, è perfetto se la probabilità a priori che il mittente invii un messaggio  $p$ , è pari alla probabilità a posteriori che il messaggio inviato, sia effettivamente  $p$ , se il crittogramma  $c$  transita sul canale.

Un'immmediata conseguenza della definizione è che, impiegando un cifrario perfetto, la conoscenza complessiva dell'attaccante, non cambia dopo che egli ha osservato un crittogramma arbitrario  $c$ , transitare sul canale.

Assumendo che:

- Il cifrario non sia perfetto.
- $P_r(P = p) = a$  con  $0 < a < 1$

Queste assunzioni implicano:  $P_r(P = p|C = c) \neq P_r(P = p)$ , quindi si possono verificare 3 casi:

1. Se  $P_r(P = p|C = c) = 0$  l'attaccante, osservando  $c$ , deduce che il messaggio spedito non è  $m$ , mentre prima sapeva che sarebbe potuto transitare  $m$  con probabilità  $a$ .
2. Se  $P_r(P = p|C = c) = 1$  si può addirittura dedurre che il messaggio spedito è proprio  $m$ .
3. In tutti i casi intermedi, l'attaccante raffina la sua conoscenza sul possibile messaggio spedito. La conoscenza del crittoanalista rimane quindi inalterata se e solo se  $P_r(P = p|C = c) = P_r(P_p)$ .

#### 1.4.2 Teorema di Shannon

Un cifrario perfetto ha  $|K| \geq |P|$ . Ovvero, il numero delle chiavi non deve mai essere più piccolo del numero di plaintext producibili.

È una condizione **necessaria** ma non sufficiente.

La dimostrazione viene fatta per assurdo. Assumendo che:

1.  $|K| < |P|$
2.  $|C| \geq |P|$  perché la funzione E deve essere invertibile. (due plaintext non possono produrre lo stesso ciphertext)
3.  $|C| > |K|$  per le conseguenze del punto 1 e 2.

Scegliendo  $p_0 : Pr(P = p_0) \neq 0$ . Questo  $p_0$  potrà produrre R ciphertext, uno per ognuna delle R chiavi. R è uguale a  $|K|$ , ma tuttavia, per il punto 3, deve esistere un  $c_0$  che non è l'immagine di  $p_0$ . Questo porta a:

$$c_0 : Pr(P = p_0|C = c_0) = 0$$

Le due probabilità sono differenti e non rispettano la *legge di Shannon*

### 1.5 One-Time Pad

Claude Shannon (1917) formalizzò, quindi, il processo crittografico perfetto, mediante un modello matematico e concluse, che il cifrario inventato da Vernam, possedeva le proprietà per essere perfetto. Il cifrario di Vernam, è detto anche **One-time Pad**. Questo nome, si riferisce alla sequenza della chiave, come se fosse scritta su un blocco di appunti (pad), e indica come tale sequenza, venga progressivamente utilizzata, per la cifratura e non sia riutilizzabile (one-time). Fu usato, intensivamente, durante la Guerra Fredda, da spie della CIA e del KGB, ma anche dalla cosiddetta Linea Rossa, il famoso collegamento telefonico diretto Washington - Mosca.

- messaggio  $m \in \{0,1\}^t$ . Messaggio in chiaro di t bits.
- chiave  $k \in \{0,1\}^t$ . Chiave di t bits scelta in maniera casuale (truly randomly chosen)
- **Funzione di Encryption:**  $E(k, m) = m \oplus k$
- **Funzione di Decryption:**  $D(k, c) = c \oplus k$

### 1.5.1 Proprietà

- **Proprietà di consistenza:**

$$c \oplus k = (m \oplus k) \oplus k = m$$

- **OTP è un cipher perfetto se:**

- $\forall m, m' \in M : \text{len}(m) = \text{len}(m')$ . Ovvero, tutti i messaggi hanno la stessa lunghezza.
- $\forall m \in M : \Pr(M = m) \neq 0$
- $k \xleftarrow{\text{random}} K$ . Ovvero,  $k$  è scelta randomicamente ed è utilizzata solo una volta

### 1.5.2 Proprietà dello XOR

- Sia  $Y$  una variabile random su  $\{0, 1\}^n$
- Sia  $X$  variabile random **uniforme e indipendente da Y** su  $\{0, 1\}^n$

Allora, si può dimostrare che:  $\Rightarrow Z = Y \oplus X$  è una variabile **uniforme** su  $\{0, 1\}^n$

#### Dimostrazione

$$\begin{aligned} P(Z = 0) &= P(X = 0 \wedge Y = 0) \vee P(X = 1 \wedge Y = 1) \\ &= P(X = 0) \cdot P(Y = 0) + P(X = 1) \cdot P(Y = 1) \\ &= 0.5Y_0 + 0.5Y_1 = 0.5(Y_0 + Y_1) = 0.5 \end{aligned}$$

### 1.5.3 Proprietà di OTP

- **Vantaggi:**

1. È unconditionally secure, cioè non può essere violato anche se l'avversario ha potenza computazionale infinita.
2. Le operazioni di criptazione e decriptazione sono molto veloci. In quanto basta eseguire una semplice operazione di XOR bit a bit.

- **Svantaggi:**

1. Le chiavi sono molto lunghe
2. Ogni chiave va utilizzata una sola volta
3. È malleabile

### 1.5.4 Problemi di OTP

1. Se uso la stessa chiave due volte, avrò che:  $c_1 = p_1 \oplus k$ ,  $c_2 = p_2 \oplus k$  Se l'avversario prova a calcolare  $c_1 \oplus c_2 = p_1 \oplus k \oplus p_2 \oplus k = p_1 \oplus p_2$  potrà avere, quindi, informazioni sul plaintext (violando Shannon).

Il caso più significativo è quello di 2 (o più) plaintext che hanno parti di testo in comune: in tal caso, l'operazione di *XOR* tra i due ciphertext prodotti risulterà essere 0. L'avversario è quindi in grado di ottenere informazioni sui due plaintext.

2. Supponendo che l'avversario conosca una coppia  $c, p$  (attacco known-plaintext), in tal caso potrà calcolare:  $k = c \oplus p$ . Se la chiave  $k$  (che è stata compromessa), viene riutilizzata, allora l'avversario sarà in grado di decifrare il nuovo ciphertext. Questo è il secondo motivo per cui non bisogna riutilizzare le chiavi.

3. L'algoritmo è malleabile (come visto precedentemente): l'avversario può manipolare il ciphertext  $c$  in maniera efficace al fine di modificare il plaintext associato. Si tratta di un problema di integrità piuttosto che di confidenzialità.

Un avversario vede  $c = m \oplus k$ . Quindi, potrebbe generare  $c' = c \oplus r$ . Al ricevente:

$$m' = c' \oplus k = (c \oplus r) \oplus k = ((m \oplus k) \oplus r) \oplus k = m \oplus r$$

La perturbazione  $r = m \oplus m'$ .

L'avversario ha, quindi, introdotto una modifica controllata  $r$  al plaintext senza che il destinatario se ne sia accorto.

Come risolvere il problema delle chiavi troppo lunghe? Possiamo utilizzare un generatore di numeri casuali, il quale output può essere controllato con un seed, e usarlo per generare *key stream*. I generatori reali non producono delle sequenze perfettamente randomiche, infatti sono chiamati *generatori di numeri pseudo-casuali*. Tuttavia, in questa situazione avremo che  $|k| < |p|$ , la quale non rispetta il teorema di Shannon: l'algoritmo non è più perfettamente sicuro.

Il trucco che si può usare, è quello di non far capire all'avversario che stiamo generando dei ciphertext a partire da un generatore di numeri pseudo-casuali. L'algoritmo necessario a distinguere l'output del PRNG da uno TRG dovrebbe avere una complessità molto alta. Quindi la nuova proprietà "di Shannon" da definire, va definita sulla base della complessità computazionale.

## 1.6 Stream ciphers

One-Time Pad è un caso particolare dello Stream Cipher. In quel caso, abbiamo un RNG perfetto. Stream Cipher è un'approssimazione di One-Time Pad.

L'idea è quella di sostituire la chiave random con una chiave pseudo-random. Il generatore pseudo-random  $G$ , è una funzione efficiente e deterministica.

Un generatore pseudo-casuale, è un algoritmo che, partendo da un valore numerico iniziale, scelto arbitrariamente (*seed*), genera una sequenza di bit, apparentemente casuali (*key-stream*).

- seed  $k$  su  $\{0, 1\}^s$
- *key stream* su  $\{0, 1\}^n$
- pseudo-random number generator  $G : \{0, 1\}^s \rightarrow \{0, 1\}^n, s \ll n$
- $c = E(G(k), m) = G(k) \oplus m$
- $m = D(G(k), c) = G(k) \oplus c$

Il cipher non è perfetto per il Teorema di Shannon, perdo la caratteristica di sicurezza perfetta, per la quale la chiave deve essere grande quanto il messaggio.

Si necessita quindi di una nuova definizione di sicurezza:

### 1.6.1 Sicurezza computazionale

Il concetto di "sicurezza computazionale" è diverso da "sicurezza perfetta", dato che quest'ultima assicura sicurezza qualsiasi sia la potenza computazionale dell'avversario. La sicurezza computazione è sicura dato che il miglior algoritmo conosciuto, richiederebbe un ammontare di risorse che vanno oltre le possibilità dell'avversario. Non esiste nessun algoritmo efficiente per distinguere un output di un PRNG da uno TRG. Il cifrario è computazionalmente sicuro se e solo se:

- PRNG ha **buone statistiche**, ovvero, deve essere in grado di generare un key stream che sembri quello generato da un processo random uniforme

- PRNG è **non predicibile**

La predicitività è importante, perché se il generatore è predicibile, l'avversario potrà ottenere il key stream partendo da una piccola parte della chiave. Un esempio può venire da OTP: supponendo di ottenere la chiave dei primi  $n$  bit, se il generatore è predicibile, allora possiamo calcolare, con una certa probabilità, la parte di key stream precedente e successiva e ottenere, poi, il plaintext.

### 1.6.2 Pseudo-random number generator

**Linear Congruential Generator (LCG)** È un generatore molto semplice, implementato similmente nella libreria *glibc* per la funzione *random()*.

$$R_0 = \text{seed}$$

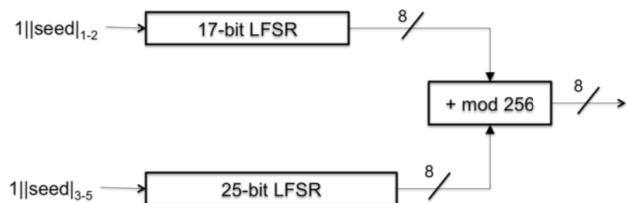
$$R_i = A \cdot R_{i-1} + B \bmod p$$

Produce streams con buone statistiche, ma è predicibile dato che è presente una relazione lineare tra i campioni.

La chiave potrebbe essere la coppia  $(A, B)$ , che potrebbe essere scoperta se un avversario conoscesse i primi bytes del plaintext e del ciphertext.

**Linear Feedback Shift Register (LFSR)** È realizzato con dei registri, delle operazioni di shift e lo XOR. Questo algoritmo non è predicibile quanto LCG, tuttavia è periodico: una volta individuato il periodo, è possibile ricavare tutti gli altri. Non è quindi adatto per applicazioni sulla security.

**Content Scrambling System (CSS)** È costruito a partire da LFSR:



- $seed = key = 5 \text{ bytes} = 40 \text{ bit}$
- $17\text{-bit LFSR: } 1||seed[0]||seed[1]$
- $25\text{-bit LFSR: } 1||seed[2]||seed[3]||seed[4]$

Provare tutte le possibili chiavi: **Exhaustive Key Search** ha una complessità di  $O(2^{40})$ . L'attacco **Known-plaintext attack**, ha una complessità di  $O(2^{17})$ : se un avversario conosce i primi 20 bytes del plaintext e del ciphertext, può arrivare a conoscere i primi 20 bytes del keystream. Quindi, può calcolarsi il 25-bit LFSR andando a calcolare la differenza tra il seed random 17-bit LFSR e il keystream calcolato. Quindi, deve controllare se l'output 25-bit LFSR è consistente con il polinomiale, che è conosciuto.

## 1.7 802.11b WEP

- pre-condivisa  $k$
- $IV$  su  $\{0, 1\}^{24}$  scritto nello standard, richiesto che sia "fresco"
- $ks = RC4(k||iv)$ , dove RC4 è il PRNG

- $m = m || \text{crc}(m)$
- $c = m \oplus ks$

Come detto, ogni volta deve essere generato un nuovo vettore di inizializzazione (IV). Come generarlo? Con un contatore (inizializzato a zero), oppure si può generare una sequenza random con un generatore. Lo standard non dice nulla riguardo a come generare il vettore, dice soltanto che  $\text{sizeof}(IV) = 24\text{bits}$ . Questo ha una forte implicazione: dopo  $2^{24}$  frames, il vettore di inizializzazione ricomincia da capo, quindi il keystream tende ad essere ripetuto. Questo comportamento deve essere evitato in crittografia.

## Drawbacks

- $ks$  dipende soltanto da  $IV$ . Come detto, dopo  $2^{24}$  frames si ripete ciclicamente e  $ks$  è riutilizzato.
- l'utilizzo di CRC, invece di una funzione hash imbrogliata da una perturbazione.
- utilizzo di "related keys"  $k||iv, iv = 0, 1, 2, \dots, 2^{24} - 1$ . RC4 è buono a meno che non vengano utilizzate chiavi "related".

## 1.8 Block ciphers

Non lavoriamo più bit a bit, ma su blocchi di bit. Si può, in tal modo, implementare un cipher mantenendo la perfect secrecy? La funzione  $E$  ora diventa una funzione che ha lo stesso dominio di  $p$  ed è invertibile (mappa, quindi, ogni blocco unicamente ad un altro blocco). Tutti i possibili blocchi di  $n$  bit sono  $2^n$ . Per avere un cipher perfetto dobbiamo implementare tutte le permutazioni dei blocchi, da cui il numero di chiavi è uguale a  $2^n!$ . La lunghezza in bit della chiave è calcolabile come  $\log_2(2^n!) \approx n2^n$ . La lunghezza della chiave varia, quindi, in modo esponenziale rispetto al numero di bit per blocco ( $n$ ).

Tuttavia, tipicamente,  $n$  è abbastanza grande per evitare che l'avversario possa effettuare un attacco a dizionario: ciò implica che, per implementare un cifratore perfetto, bisogna implementare un numero grande di chiavi (esponenziale). Nella pratica, quindi, non è possibile implementare la perfect secrecy nei sistemi di cifratura a blocchi.

Nella pratica il cipher è considerato buono se l'avversario non riesce a distinguere da un true block cipher.

Il plaintext è diviso in blocchi di lunghezza fissata  $n$ :

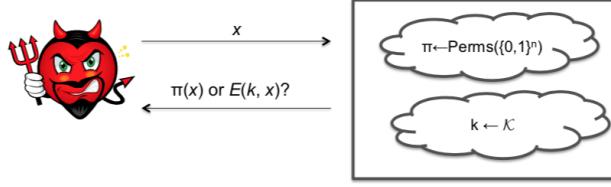


- $E : \{0, 1\}^n \rightarrow \{0, 1\}^n$
- $D : \{0, 1\}^n \rightarrow \{0, 1\}^n$

E è una **permutazione**. Un block cipher perfetto implementa tutte le possibili permutazioni  $2^n!$

$$\text{key size} = \log_2 2^n! \approx n \times 2^n$$

La chiave sarebbe troppo grande. In pratica, la funzione di criptaggio che corrisponde a una chiave scelta randomicamente ( $E(k, x)$ ), dovrebbe apparire come una permutazione scelta randomicamente ( $\pi(x)$ ).



La grandezza del blocco dovrebbe essere grande abbastanza da rendere un attacco dizionario infattibile in termini di complessità di storage, quindi se  $n = 64$ , un avversario dovrebbe salvare  $2^{64} \times (8 \times 2)$  bytes (8 bytes per il plaintext e ciphertext).

### 1.8.1 Computational security vs Attack complexity

**Computational security** Un cifrario è computazionalmente (praticamente) sicuro quando, il livello percepito di computazione richiesto per romperlo, utilizzando il miglior attacco conosciuto, eccede, di un margine "di sicurezza", le risorse computazionali dell'avversario ipotizzato. Si assume, quindi, che l'avversario abbia una potenza computazionale limitata.

**Attack complexity** La complessità di un attacco dipende da:

- **Data Complexity:** il numero previsto di unità di dati di input richiesti
- **Storage Complexity:** il numero previsto di unità di storage richieste.
- **Processing Complexity:** il numero previsto di operazioni richieste per processare i dati di input e/o riempire lo spazio di storage con i dati.

**Computational security vs Attack complexity** Un block cipher è computazionalmente sicuro se:

- La grandezza dei blocchi  $n$  è sufficientemente grande da precludere una data analysis esaustiva
- La grandezza della chiave  $k$  è sufficientemente grande da precludere una ricerca della chiave esaustiva
- Nessun attacco conosciuto ha complessità di dati e di processing minore, rispettivamente, di  $2^n$  e  $2^k$ .

### 1.8.2 Exhaustive key search

Con  $k > n$  è possibile trovare chiavi false positive. Data una coppia  $x_1$  e  $y_1$ , la probabilità che quella chiave  $k$  sia in grado di mappare  $x_1$  in  $y_1$  è:

$$P = \frac{1}{2^n}$$

questo perché l'algoritmo può essere considerato sicuro e l'output può essere visto come una variabile random uniforme. Dato che il numero di chiavi sono  $2^k$ , il numero di chiavi candidate atteso è:

$$E(\#keys) = 2^k \times \frac{1}{2^n} = 2^{k-n}$$

Se si vuole ridurre il set di chiavi candidate, si deve collezionare altre coppie (relativamente al concetto di *unicity distance*). Se si hanno  $t$  coppie plaintext-ciphertext, il numero di chiavi candidate atteso sarà:

$$E(\#keys) = 2^{k-tn}$$

Un criptaggio multiplo con r-stage di criptaggi ha  $2^{rk-tn}$  chiavi candidate.

### 1.8.3 Confusion and diffusion

In accordo a Shannon, ci sono due operazioni primitive con cui possono essere costruiti algoritmi di criptaggio forti:

- **Confusion:** operazione di criptaggio dove la relazione tra chiave e ciphertext è oscurata.
- **Diffusion:** l'influenza di un simbolo del plaintext è diffusa su molti simboli del ciphertext con l'obiettivo di nascondere le proprietà statistiche del plaintext, utilizzando le permutazioni.

I block cipher buoni dovrebbero avere *confusion* e *diffusion*. Se ce n'è soltanto uno dei due, il block cipher sarà rotto. Quindi, queste due funzioni devono essere usate in combinazione, concatenate.

### 1.8.4 Data Encryption Standard (DES)

DES è deprecato e considerato non più sicuro, per via della chiave che è troppo corta (56 bits). Potremmo pensare di cifrare più volte usando E con chiavi diverse: esiste uno standard che sfrutta questo metodo ed è chiamato 3DES.

L'operazione di cifratura è fatta in questo modo:

$$3E(e_1, e_2, e_3, p) = E(e_1, D(e_2, E(e_3, p)))$$

Nel ricevitore basta effettuare l'operazione inversa. Perché si cifra, poi si decifra e poi nuovamente si decifra? Serve per mantenere la compatibilità con lo standard DES (backward compatibility): basta notare che se  $e_1 = e_2 = e_3$ , il risultato di 3DES è lo stesso di DES usando la chiave  $e_{1/2/3}$ . La complessità dell'attacco dell'avversario, tuttavia, diventa solo  $2^{118}$  e non  $2^{56-3}$ .

Perché non si utilizza una doppia encriptazione invece di 3DES? Perché non è sicura. Questo perché la complessità dell'attacco resta la stessa di quella di DES. Ad esempio:

$$E(e_1, E(e_2, p))$$

L'avversario potrebbe provare ad effettuare un attacco bruteforce sulle chiavi, quindi in tal caso si troverà a generare  $2^{56-2}$  chiavi. Tuttavia, questo è un approccio abbastanza naive, ne esiste uno più efficiente chiamato *Meet-in-the-middle*: si tratta ancora di un attacco known-plain text, nella quale l'avversario conosce una coppia  $(c^*, p^*)$ .

L'avversario può calcolarsi una tabella composta da:

$$\forall i e_i, E(e_i, p^*)$$

Successivamente, per ogni  $j$ , può calcolare:

$$y' = D(e_j, c^*)$$

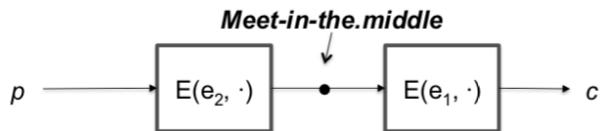
e confrontarlo con gli elementi della tabella precedente: se i due risultati coincidono, allora  $(e_i, e_j)$  è la chiave,

Perché questo metodo è più efficiente di quello precedente? Perché la prima operazione richiede di calcolare e memorizzare  $2^{56}$  coppie  $(e_i, E(e_i, p^*))$  e la seconda richiede di effettuare al più  $2^{56}$  operazioni di decriptazione; le due complessità, in questo caso, si sommano e non si moltiplicano. Per questo la complessità resta  $O(2^{56})$ , rendendo l'algoritmo non più sicuro di DES. Nel caso di 3DES, questo attacco non è un problema, perché la complessità sarebbe pari a  $2^{112}$  e la complessità dello spazio pari a  $2^{56}$ .

2DES

$$k = k_1 + k_2$$

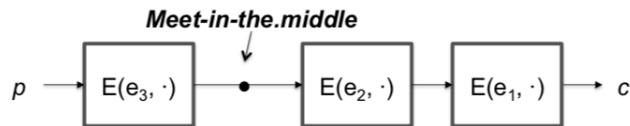
Anche se la nuova chiave è su 112 bits, questa non va a migliorare la sicurezza perché l'attacco *Meet-in-the-middle* (Known-Plain text) ha una complessità  $O(2^{56})$  per trovare  $k_1$  e  $O(2^{56})$  per trovare  $k_2$ , comparando il testo criptato  $X_L$  e quello decriptato  $Y_L$ . Questo porta a una complessità totale di  $O(2^{56})$ . In più, dimezza le performance.



3DES

$$k = k_1 + k_2 + k_3$$

La nuova chiave è su 168 bits e l'attacco *Meet-in-the-middle* ha una complessità di  $O(2^{56 \times 2}) = O(2^{112})$ . Nella versione *EDE*  $k_1 = k_2 = k_3$ .



**Key whitening** È utilizzato in nei cifrari detti "scatole nere" (senza vulnerabilità interne), che utilizzano chiavi piccole.

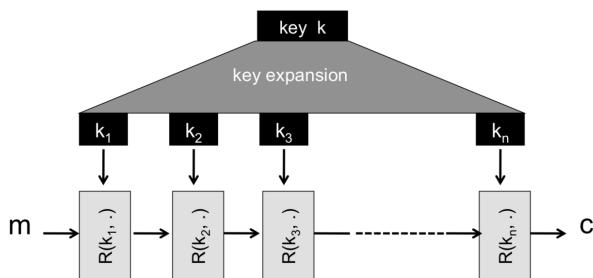
Prevede solo il processo di criptazione più pre-xor e post-xor:

$$c = k_2 \oplus E(k, (m \oplus k1))$$

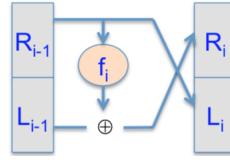
## Complessità degli attacchi

- attacco *bruteforce*:  $O(2^{k+2n})$  computazionale
  - attacco *meet-in-the-middle*:  $O(2^{k+n})$  computational + storage
  - attacco più efficiente: l'avversario conosce  $2^m$  (PT-CT)-coppie (complessità dei dati)  
 $\Rightarrow O(2^{k+n-m})$
  - $n = 64$
  - $k = 56$

DES è un algoritmo iterativo. Per ogni blocco del plaintext, il criptaggio è gestito in 16 rounds che eseguono tutte operazioni identiche. In ogni round è utilizzata una differente sotto-chiave e tutte le sotto-chiavi  $k_i$  sono derivate dalla chiave principale  $k$  dal *key schedule*.



La funzione round è costruita a partire da un *Feistel network*:



$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, k_i)$$

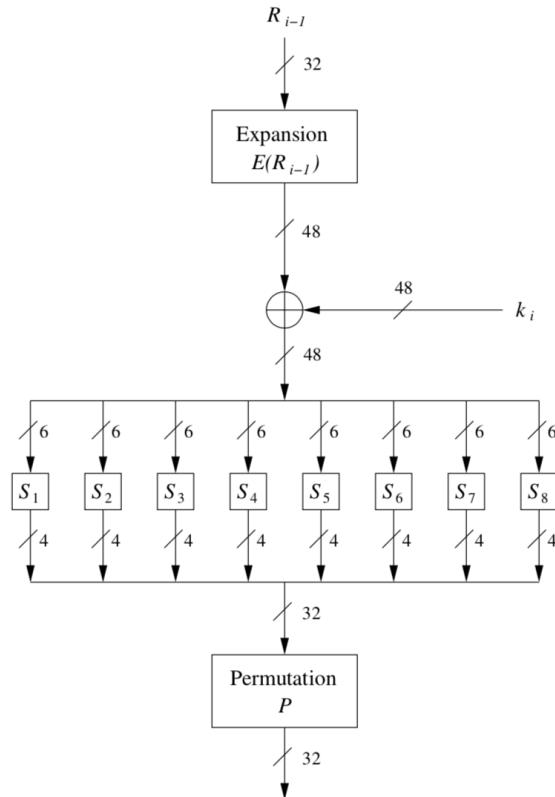
Un vantaggio del Feistel network è che il criptaggio e il de-cryptaggio sono praticamente le solite operazioni, semplicemente attraversare il network in direzione opposta unendo le chiavi in ordine inverso.

$$R_{i-1} = L_i$$

$$L_{i-1} = R_i \oplus f(R_{i-1}, k_i) = R_i \oplus f(L_i, k_i)$$

La struttura Feistel critta (decripta) solo la metà dei bit di input per ogni round.

**Round function  $f$**  La funzione  $f$  è un preudo-random generator con due parametri di input  $R_{i-1}$  e  $k_i$ . Dovrebbe essere non predicibile. Utilizza blocchi predefiniti non lineari e mappa 32 bits di input a 32 bits di output utilizzando una chiave "round" da 48 bit  $k_i$ , con  $1 \leq i \leq 16$ . La *Diffusion* è fatta da una *permutazione IP iniziale* utilizzando una permutazione e duplicazione bit-a-bit, *confusion* è fatta da S-boxes.

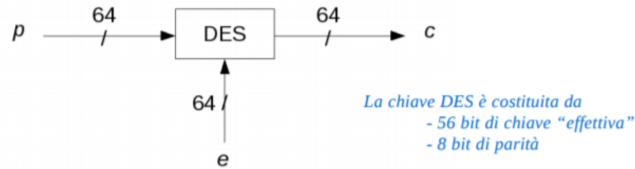


**DES properties** DES è un buon cifrario e soddisfa le seguenti proprietà:

- ogni bit del ciphertext dipende completamente dai bit della chiave e del plaintext
- non ci sono evidenti relazioni statistiche tra il plaintext e il ciphertext
- il cambiamento di un bit nel plaintext (ciphertext) causa il cambiamento di ogni bit nel ciphertext (plaintext) con probabilità 0,5 (RV uniforme).

## Visione delle componenti di DES

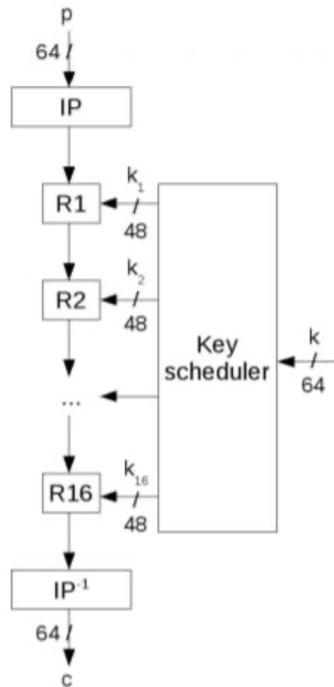
**1 - Visione esterna** La visione esterna di DES è la seguente:



Affinché un cifrario a blocchi sia ideale (perfettamente random), devono essere possibili un numero di permutazioni pari a  $2^n! = 2^64!$ , ma la chiave DES è a soli 56 bit, dunque può dar luogo a sole  $2^{56}$  permutazioni che è minore di  $2^{64}$  richiesto.  
DES non è dunque un cifrario perfettamente random.

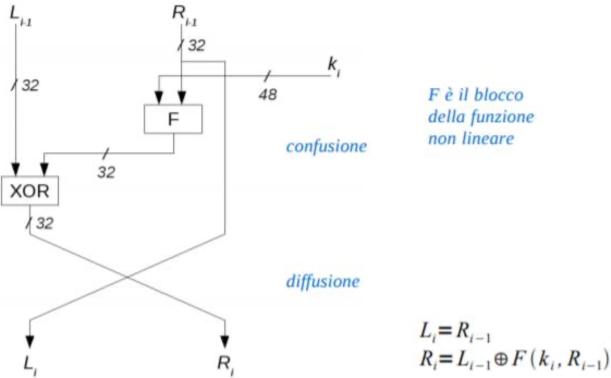
**2 - Visione interna** Il cifrario DES ha la seguente struttura interna in fase di cifratura.

Nella decifratura si hanno i medesimi blocchi, solo che le chiavi sono schedurate al contrario:  $k_{16}$  in R1 e  $k_1$  in R2.



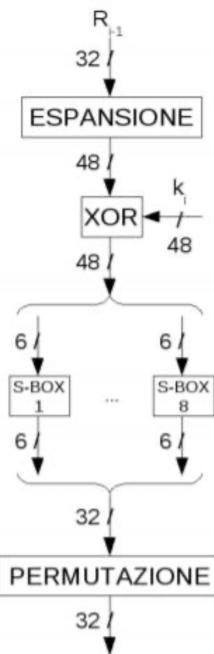
**3 - Visione di un round - Rete di Feistel** Ogni round lavora sul semicifrato del round precedente. Ogni semicifrato viene diviso in 2 parti:

$$\text{SEMICIFRATO}_{(i-1)}(64\text{bit}) = L_{(i-1)}(32\text{bit}), R_{(i-1)}(32\text{bit})$$



L'invertibilità di tutte le  $f_i$  garantisce anche l'invertibilità della funzione finale. La funzione  $f_i$  prende in ingresso sia  $R_i$  che una chiave  $k_i$ . Tutti i  $k_i$  sono generati a partire da  $k$  attraverso una procedura *key expansion*.

**4 - Funzione F non lineare** La sicurezza di DES sta tutta nelle **S-BOX**: sono delle tabelle non lineari. In base all'ingresso viene scelta la corrispondenza nella tabella. Una di esse fu modificata dal NSA americano quando scrutinò DES. Con le S-BOX si maschera il legame tra testo cifrato e chiave e inoltre si aggiunge non linearità.



## 1.9 Note sulla Criptoanalisi

Un buon cipher deve essere in grado di mascherare le proprietà statistiche del plaintext: se queste si ripercuotono sul ciphertext, l'avversario può ottenere informazioni sul plaintext

e, in alcuni casi, quindi, anche sulla chiave. Un esempio storico è quello del *cifrario monoalfabetico*: la chiave consiste in una tabella di lettere e la funzione di criptazione opera sostituendo le lettere del plaintext con quelle nella chiave (permutazione delle lettere). L'avversario può adoperare tecniche di criptoanalisi, ad esempio, partendo dalla lingua nella quale è scritto il messaggio. Questo perché, ogni lingua è caratterizzata dall'utilizzo più o meno frequente di certi caratteri, quindi l'avversario può associarli con i caratteri del plaintext più frequenti, ottenendo il plaintext e in questo caso anche parte della chiave. Uno spazio delle chiavi molto grande non è sufficiente (è solo condizione necessaria) a mascherare le proprietà statistiche del plaintext, quindi la funzione di criptazione deve far in modo che il ciphertext appaia come una serie di caratteri il più possibile aleatoria.

## 1.10 Encryption Modes

Per quanto abbiamo visto dal punto di vista matematico, la funzione di encrypting  $E$  prende in ingresso un plaintext di dimensione  $n$ , una chiave di dimensione  $k$  e restituisce il ciphertext di dimensione  $n$ .

Si vede adesso, come cifrare quando il plaintext ha dimensione maggiore o minore rispetto alla dimensione fissa  $n$  di un blocco.

Plaintext < dimensione blocco( $n$ ):

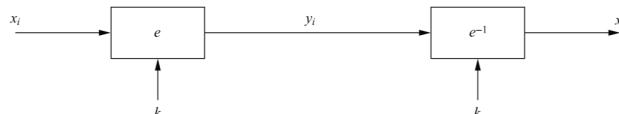
- Padding

Plaintext > dimensione blocco( $n$ ):

- Encryption Modes: ECB, CBC ...

### 1.10.1 Electronic Codebook (ECB)

Dividiamo il plaintext in più parti di uguale lunghezza e cifriamo ogni singola parte, indipendentemente.



#### Pros

- Non è richiesta nessuna sincronizzazione tra blocchi.
- Non c'è propagazione di errori.
- Può essere parallelizzata.

#### Cons

- Blocchi di plaintext identici, risultano in identici blocchi di ciphertext: non nasconde pattern di dati e consente l'analisi del traffico.
- Consente il re-ordinamento dei blocchi e la sostituzione.

Gli svantaggi ci indicano che questo sistema è malleabile: un avversario può registrare un blocco e rimpiazzarlo in un ciphertext successivo, così da alterare, efficientemente, il plaintext associato. Una soluzione può essere quella di associare un timestamp al messaggio, in modo da evitare attacchi di replay: il problema persiste perché l'avversario può ancora intercettare il messaggio, modificare i blocchi che contengono le informazioni rilevanti e poi trasmettere il messaggio alterato. Una encryption mode che risolve questi problemi è CBC.

### 1.10.2 Cipherblock Chaining (CBC)

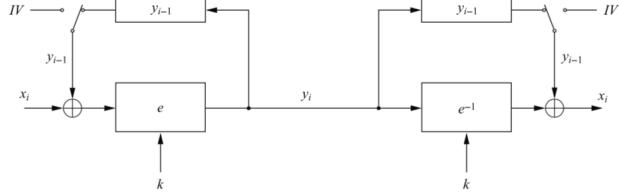
Prima che il blocco di plaintext corrente venga cifrato, viene fatta un'operazione di XOR con il ciphertext ottenuto dal blocco precedente. Per il primo blocco si usa un *Initialization Vector*, il quale serve per randomizzare il ciphertext finale: CBC è, in tal senso, deterministicio, ciò vuol dire che plaintext identici possono produrre ciphertext differenti a parità di chiave. IV è una sorta di nonce. Non sussiste più neanche il problema del riordinamento dei blocchi, in quanto questo porterebbe ad alterare l'intero ciphertext. L'IV va rigenerato ad ogni ritrasmissione (come un nonce) e può essere mandato in chiaro, anche se bisogna assicurarsi che resti integro. Non è necessario un RNG, può bastare anche un contatore.

Con CBC si vanno ad eliminare 2 problemi visti in ECB:

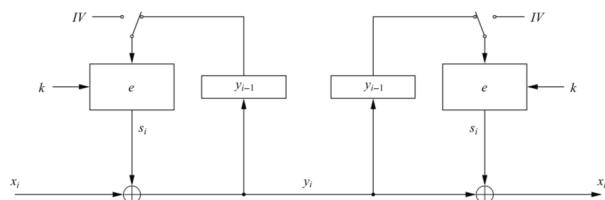
1. IV e lo XOR introducono una randomizzazione: 2 plaintext diversi producono ciphertext diversi.
2. Si rilevano immediatamente riordini o sostituzioni, ma ciò inficia l'operazione di decifratura. Una sostituzione cambia la comprensione dell'intero plaintext ricevuto.

Tuttavia, CBC introduce alcuni problemi:

- L'algoritmo non può essere parallelizzato perché ogni operazione richiede il risultato di quella precedente.
- Un errore nel blocco  $i$ -esimo si propaga in tutti i blocchi successivi.

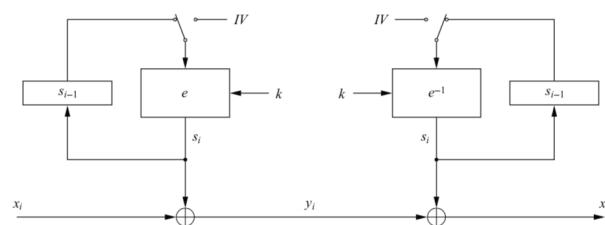


### 1.10.3 Cipher Feedback (CFB)



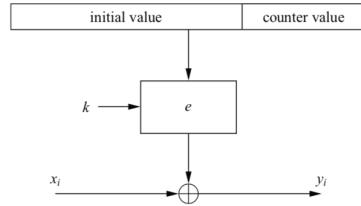
CFB è uno stream cipher blocco a blocco asincrono, perché dipende dal ciphertext precedente.

### 1.10.4 Output Feedback (OFB)



OFB è uno stream cipher blocco a blocco sincrono e consente la pre-computazione, questo perché il blocco successivo non dipende ne da  $PT$  o  $CT$ .

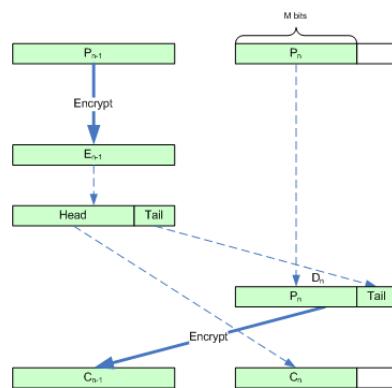
### 1.10.5 Counter Mode (CTR)



Da' un limite superiore all'ammontare di dati che possono essere criptati.

### 1.10.6 Ciphertext Stealing Mode (CTS)

Cripta utilizzando EBC senza la ciphertext expansion.



### 1.10.7 Padding (PKCS #7)

Il padding viene utilizzato per rendere la lunghezza del plaintext multipla della grandezza  $n$  del blocco. In tutti i casi la lunghezza del ciphertext è sempre più grande del plaintext. Questo fenomeno è chiamato *ciphertext expansion*.

Il meccanismo del padding è molto semplice: se il plaintext non è un multiplo della dimensione di un blocco, vengono aggiunti una serie di 3 per completarlo.

L'utilizzo di questa tecnica introduce un problema: se avessi voluto inviare un plaintext con dei 3 in coda? Per risolvere questo problema e se il plaintext è un multiplo della dimensione del blocco, si aggiunge un padding block, composto da soli 8, subito dopo.

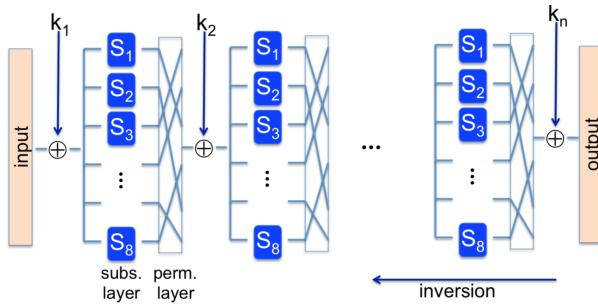
## 1.11 Advanced Encryption Standard (AES)

Similmente a DES, anche in questo caso si usa uno schema di funzioni a round. Tuttavia le reti di Feistel sono sostituite con delle reti di *Subs-Perms*: queste effettuano, per ogni round e in ordine, delle operazioni di byte substitution, row shift e column mix, restando comunque ancora delle funzioni invertibili. Si usa ancora la *key expansion*.

- $n = 128$
- $k = 128/192/256$

key lengths	# rounds = $n_r$
128 bit	10
192 bit	12
256 bit	14

AES non ha una struttura Feistel.



AES consiste di 3 *layers*: AES consists of three *layers*:

- **Key addition layer:** Una chiave rotonda a 128-bit che è stata derivata dalla chiave principale nello schedule della chiave a cui viene applicato uno XOR allo stato.
- **Byte substitution layer (S-box):** Introduce *confusion* trasformando nonlinearmente ogni elemento dello stato utilizzando tabelle lookup.
- **Diffusion layer:** Introduce *diffusion* attraverso le seguenti operazioni: *ShiftRows* e *MixColumn*.

## 2 Hash function

La criptazione asimmetrica ha come scopo quello di garantire la *confidenzialità*.

Le funzioni hash, invece, possono essere utilizzate per *l'integrità* e *l'autenticazione*.

L'autenticazione dell'origine dei dati richiede il controllo di integrità, e viceversa.

$$H : \{0, 1\}^* \rightarrow \{0, 1\}^n$$

Le collisioni avvengono quando almeno 2 input hanno lo stesso *digest*. Una funzione hash sicura è quando le collisioni esistono ma sono "difficili" da trovare.

L'integrità del messaggio è la proprietà con cui i dati non sono stati alterati in modo non autorizzato dal momento in cui è stato creato, trasmesso o memorizzato da una fonte autorizzata.

### 2.1 Proprietà informali

Una funzione hash deve rispettare, in pratica, queste proprietà:

1. essere applicabile a messaggi di qualsiasi grandezza
2. l'uscita deve essere di grandezza prefissata
3. facile da applicare
4. difficile da invertire
5. ogni hash deve essere unico

Le proprietà più problematiche da rispettare, nella realtà, sono le ultime due. L'hash non può essere matematicamente unico se il numero degli input è maggiore del numero degli output possibili: di conseguenza, le funzioni possono presentare degli  $x_0, x_1$  tali che  $H(x_0) = H(x_1)$ , cioè una tipica situazione di *collisione*. Per mitigare questo problema, l'algoritmo dovrà almeno rendere molto difficile trovare una collisione.

### 2.2 Proprietà di sicurezza

- **Preimage resistance o One-Wayness:** Dato un output hash  $z$ , deve essere "difficile" trovare l'input  $x$  tale che  $z = h(x)$ .
- **Second Preimage Resistance o Weak Collision Resistance** Dato un input  $x$  deve essere "difficile" trovare  $x'$  tale che  $h(x) = h(x')$ .
- **Collision Resistance:** Deve essere difficile trovare  $x$  e  $x'$  tale che  $h(x) = h(x')$ .

Generalmente, una funzione hash deve soddisfare almeno 2 su 3 di queste proprietà.

Possono essere categorizzate in:

- **One-way Hash Functions (weak one-way):** soddisfano le proprietà di preimage e 2nd-preimage resistance.
- **Collision Resistant Hash Functions (strong one-way):** soddisfano le proprietà di 2nd-preimage e collision resistance.

Se una funzione hash ha la proprietà di collision resistance, è dimostrabile che questa soddisfi anche la proprietà di 2nd preimage resistance.

Tuttavia, non è vero che avere la collision resistance implica avere anche la preimage resistance: per questo le CRHF posseggono quasi sempre la prima proprietà.

## 2.3 Attacchi alle Funzioni Hash

L'obiettivo dell'attacco è produrre una collisione. Ci si può trovare in due situazioni:

- **Selective Forgery:** l'avversario ha il controllo completo o parziale su  $x$ . (Per esempio, sapendo che  $x$  è un pdf,  $x'$  viene generato considerando il fatto che deve collidere con un file pdf.)
- **Existential Forgery:** l'avversario non ha il controllo su  $x$ .

L'avversario non ha controllo su  $x$ , se l'informazione che rappresenta non è consistente o non persegue lo scopo dell'avversario.

### 2.3.1 Black box attacks

Una categoria di attacchi alle funzioni hash è quella del tipo **Black Box**, nella quale:

- $H$  è considerata come una scatola nera.
- teniamo in considerazione solo la dimensione dell'output
- $H$  è considerata approssimativamente una variabile casuale.

Questi attacchi, quindi, non sfruttano le caratteristiche interne delle funzioni hash:

- Guessing attack contro  $2^{nd}$  preimage:  $O(2^n)$
- Birthday attack contro  $3^{rd}$  preimage:  $O(2^{\frac{n}{2}})$

**Guessing Attack  $O(2^n)$**  L'obiettivo è quello di violare la 2nd preimage resistance, partendo da un plaintext target  $x_0$ . L'algoritmo è il seguente:

```

do
    x <- random()
    while H(x) == H(x0)
    return x;

```

Ogni step richiede di generare un numero random e di trovare l'hash.

Ogni hash calcolato ha una probabilità di  $\frac{1}{2^n}$  di essere il risultato da trovare. L'algoritmo ha, quindi, una complessità di  $O(2^n)$ , principalmente dovuto alle iterazioni. È sostanzialmente un attacco brute force.

### Birthday Attack $O(2^{\frac{n}{2}})$

1. generiamo  $N = 2^{\frac{n}{2}}$  messaggi/numeri casuali, che chiameremo  $x_1, x_2, \dots, x_n$ .
2. calcoliamo l'hash per ogni messaggio, cioè  $t_i = H(x_i)$ , ottenendo  $t_1, t_2, \dots, t_n$ .
3. scorriamo gli hash prodotti per individuare due hash uguali, quindi  $t_i == t_j, i \neq j$ .
4. se non troviamo nessun hash uguale, ripartiamo dal primo punto.

Dal punto di vista della complessità, ogni singola iterazione è complessa  $2^{\frac{n}{2}}$ , tuttavia non conosciamo qual è il numero medio di iterazioni da fare prima di trovare una collisione. Per il momento, si può dire che, osservando le complessità dei due attacchi, si nota che violare la seconda proprietà (trovare un  $x'$  con valore di output uguale all'output di un  $x$  noto), è più difficile violare la terza proprietà (trovare due valori,  $x$  e  $x'$  i cui output sono uguali). È un problema di probabilità. A questo punto ci viene in aiuto il:

## Birthday Paradox

- **Problema 1:** Qual è la probabilità che, in una stanza dove ci sono 23 persone, almeno una sia nata il 25 Dicembre?

$$\underbrace{\frac{1}{365} + \frac{1}{365} + \dots + \frac{1}{365}}_{23 \text{ volte}} = 0,063$$

Questo problema è riconducibile a infrangere la seconda proprietà (Guessing attack).

- **Problema 2:** In una stanza in cui ci sono 23 persone, qual è la probabilità che almeno 2 persone siano nate lo stesso giorno?

Per semplicità, si può risolvere il problema inverso Q (la probabilità che 2 persone non siano nate lo stesso giorno) e calcolare P come:

$$Q = 1 - P$$

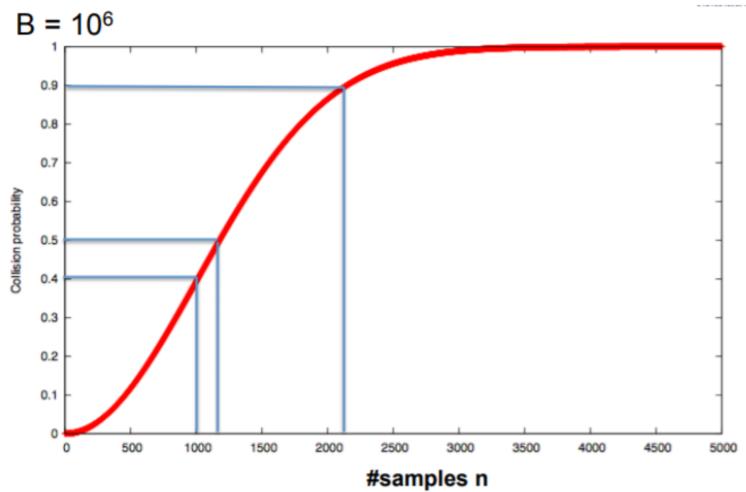
$$\frac{364}{365} \cdot \frac{363}{365} \cdot \dots \cdot \frac{(364 - 23 + 1)}{365} = 0,493$$

$$P = 1 - Q = 0,507$$

La probabilità, quindi, è nettamente superiore a quella del problema precedente.

Questo problema è riconducibile al Birthday attack.

**Teorema formalizzato** Prendiamo  $n$  interi  $r_1, r_2, r_3, \dots, r_n \in \{\exists i \neq j : r_i = r_j\} \geq \frac{1}{2}$  questo teorema ci dice che se abbiamo un intervallo di valori (da  $i$  a  $B$ ) e scegliamo casualmente  $n$  valori, se  $n = 1.2 * B^{\frac{1}{2}}$ , allora si avranno il 50% di possibilità, pescando 2 valori, che siano uguali.



Dal grafico si può notare che per un valore di  $B = 10^6$ , prendendo 1200 valori, si ha la probabilità del 50% di avere 2 valori uguali. Questa probabilità aumenta all'aumentare di  $n$ .

Quindi, si applica questo teorema al birthday attack per determinare il numero di iterazioni si devono effettuare:

$$t_i = H(x_i)$$

$$t_i \in [0.2^n - 1] \text{ dove } n \text{ è il numero di bit in uscita (MD5 128)}$$

$$B = 2^n \text{ massimo valore dell'intervallo}$$

$$N = B^{\frac{1}{2}} \text{ si approssima il numero di valori scelti, togliendo 1.2}$$

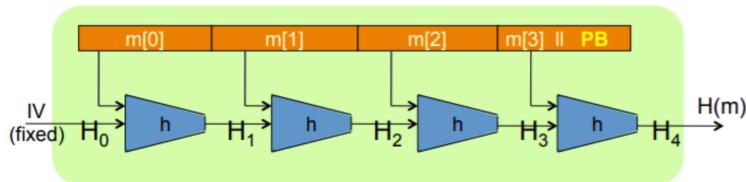
Con questo numero di valori random scelti ( $N$ ), si può ottenere una collisione in una singola iterazione (complessità  $2^{\frac{n}{2}}$ ) con una probabilità di  $\frac{1}{2}$ .

In conclusione, con circa 2 iterazioni posso ottenere una collisione con il birthday attack. Con questo attacco, però, si ha la necessità di utilizzare più risorse in quanto si deve memorizzare tutti i valori di output per poi cercare una collisione fra questi. (*large store complexity*).

## 2.4 Costruzione di una Funzione Hash

### 2.4.1 Schema Merkle-Damgard

Questo schema è utilizzato per costruire una funzione hash  $H$ . Se vogliamo costruire una **Collision Resistant Hash Function (CRHF)** si ha bisogno un  $h$  resistente alle collisioni.

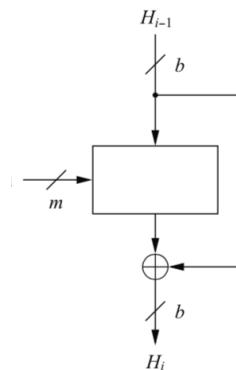


Dove  $h$  è definita *compression function* e  $PB$  è il *padding block*. Da questo schema deriva un teorema molto importante:

**Teorema:** Se  $h$  è resistente alle collisioni allora  $H$  è resistente alle collisioni.

**Dimostrazione:** Una collisione in  $H$  equivale a una collisione in  $h$ .

Per costruire un **CRHF** è sufficiente costruire delle compression function resistenti alle collisioni.



**Schema Davies-Meyer** Questo schema viene utilizzato per costruire  $h$  resistente alle collisioni. È costruito utilizzando un block cipher (AES-128) che prende  $H_i$  come input e  $x_i$  come chiave. L'output finale è ottenuto effettuando un'operazione di XOR che rende  $h$  resistente alle collisioni:

$$H_{i+1} = E(x_i, H_i) \oplus H_i$$

Per rendere l'attacco birthday molto meno infattibile, può essere utilizzato lo *Schema Hirose*. Infatti, prevede l'utilizzo di block ciphers per poter avere un output di  $2 \times 128$  bit.

### 3 Message Authentication Codes (MACs)

L'autenticazione dell'origine del messaggio è un tipo di autenticazione che serve a confermare un'entità come sorgente di un messaggio creato nel passato.

Tale tipo di autenticazione ha come risultato intrinseco anche l'integrità del messaggio. MAC non viene usato per la confidenzialità, l'obiettivo è quello dell'autenticazione: come garantire che un messaggio  $m$  sia stato inviato da una certa entità?

Supponiamo che Alice debba mandare a Bob un messaggio  $m$ : per garantire l'autenticità, Alice produce un tag  $t$  e lo trasmette insieme ad  $m$ . Per produrre  $t$  usiamo una chiave predivisa  $k$  (similmente agli schemi simmetrici) come ingressi per una funzione  $S$ . Quindi:

$$m, t = S(k, m) \rightarrow Bob$$

Bob verifica l'autenticità usando una funzione di verifica  $V$ :

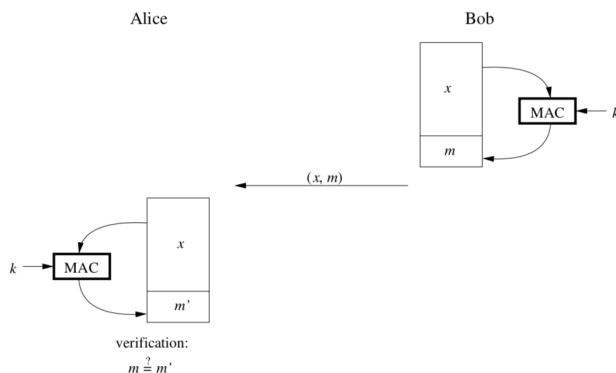
$$V(m, k, t) = true/false$$

La funzione di verifica consiste in calcolare  $t'$  e confrontarlo con  $t$  ricevuto:

$$t' = S(k, m)$$

$$\text{return } t' == t$$

In definitiva, MAC fornisce l'integrità del messaggio e autenticazione.



- $m$ : messaggio in chiaro
- $k$ : chiave condivisa tra le due parti
- $t$ : tag o digest o fingerprint
- $S$ : Algoritmo di generazione MAC
- $V$ : Algoritmo di verifica MAC

Gode delle seguenti proprietà basilari:

- **compressione:** è una funzione che mappa su un numero finito di bytes (output) un ingresso arbitrario.
- **facilità computazionale:** dato un input  $x$ , deve essere facile computazionalmente calcolare  $h(x)$ .
- **resistenza computazionale:** per ogni chiave  $k$ , date zero o più coppie  $x_i, h_k(x_i)$ , è difficile calcolare  $x, h_k(x)$  per qualsiasi  $x \neq x_i$ ; ossia, se non si ha già a disposizione il MAC di un ingresso, senza una chiave non lo si trova facilmente osservando i risultati di altri input. Questa proprietà implica la *key-non-recovery*, non viceversa.

Si osservi come:

- se si conosce  $k$ , la definizione di secure MAC non dice niente a proposito di eventuali resistenza e preimage o 2ND preimage, in quanto se si condivide una chiave siamo in un modello *"mutual trust model"*.
- se non si conosce  $k$ , la funzione  $h$  è preimage, second preimage e collision resistant.
  - per la 2ND preimage e la collision, basta osservare che l'avversario non ha semplicemente modo di calcolare il MAC essendo sprovvisto di  $k$ .
  - per la preimage si ragiona per assurdo: se si ammette il possibile recovery della preimage  $x$  di un hash casuale, si viola la stessa definizione di resistenza computazionale.

L'avversario con le funzioni MAC ha come obiettivo quello di riuscire a creare delle coppie  $x_j, h_k(x_j)$  a partire da alcune coppie note  $x_i, h_k(x_i)$  senza conoscere  $k$ , in tal modo potrà far sì che chi riceve le coppie da lui generate, creda che siano provenienti dalle persone a venti  $k$ .

Anche in tal caso, si hanno due tipo di forgeries (falsificazioni):

- **selective forgery:** l'avversario può produrre coppie testo-MAC di sua scelta.
- **existential forgery:** l'avversario può produrre coppie testo-MAC, ma senza poter decidere/conoscere il contenuto del testo.

Ovviamente, se l'avversario entra in possesso della chiave  $k$ , entrambe le falsificazioni sono possibili. È dunque bene introdurre ridondanza o strutture prefissate con del padding al fine di ridurre le possibilità di forgery dell'avversario.

Obiettivo delle funzioni MAC con output a  $m$  bit e chiave a  $t$  bit è quindi quello di ottenere:

- Una non possibile recovery della chiave a meno di algoritmi di complessità  $2^t$ : **exhaustive key search**
- Una *computational resistance* che dia luogo a forgery con probabilità pari alla maggiore tra  $2^{-t}$  e  $2^{-m}$  di un guessing attack.

La sicurezza dal punto di vista computazionale è raggiunta con un numero di bit per l'output superiore a 64 e tra 64 e 80 per la chiave.

**Come si applica MAC in pratica?** Per garantire anche la sicurezza, viene utilizzato in combinazione alla encryptazione. Ci sono 3 modelli principali e tutti utilizzano due chiavi diverse per l'encrypt ( $e$ ) e l'autenticazione ( $a$ ).

- **SSL:**  $t = S(a, m)$ ,  $c = E(e, m||t)$ ,  $m' = c$
- **IpSec:**  $c = E(e, m)$ ,  $t = S(a, c)$ ,  $m' = c||t$
- **SSH:**  $c = E(e, m)$ ,  $t = S(a, m)$ ,  $m' = c||t$

### 3.1 Algoritmo di generazione MAC

- Deve essere "semplice"
- $S : \{0, 1\}^* \rightarrow \{0, 1\}^n$  è una funzione hash
- **Computational resistance property (key non-recovery):** per una certa chiave  $k$ , dato  $(m_i, t_i)$ , dove  $t_i = S(k, m_i)$ , deve essere "difficile" calcolare  $(m, t) : t = S(k, m)$  per ogni  $m \neq m_i$  senza conoscere  $k$ .

In pratica:

- L'avversario non può produrre tag per nuovi messaggi (a scelta o non)
- L'avversario non può calcolare la chiave partendo dalle coppie  $(m_i, t_i)$
- S, per l'avversario che non conosce  $k$ , deve possedere le proprietà di:
  - *2nd-preimage* e *collision resistance*
  - *preimage resistance* per attacchi chosen-text
- Non c'è nessuna proprietà da rispettare nel caso di un'entità che conosce  $k$ : infatti ci basiamo sul fatto che la chiave sia un segreto che non vada rivelato e che entrambi le parti siano in grado di non comprometterla.

La chiave utilizzata per il MAC (per l'autenticazione) deve essere diversa da quella usata dal cipher simmetrico (per la confidenzialità).

### 3.2 MACs from hash function

#### Secret prefix scheme

$$t = h(k||m)$$

Non è sicuro perché il MAC può essere costruito da  $m$  senza conoscere la chiave.

#### Secret postfix scheme

$$t = h(m||k)$$

Se un avversario trova una collisione nella funzione hash, tale che  $m_0 : h(m) = h(m_0)$ , allora

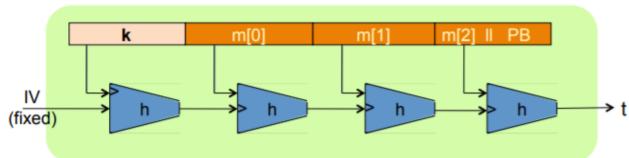
$$h(m||k) = h(m_0||k)$$

### 3.3 Come costruire le funzioni MAC

Si possono utilizzare le seguenti categorie di funzioni:

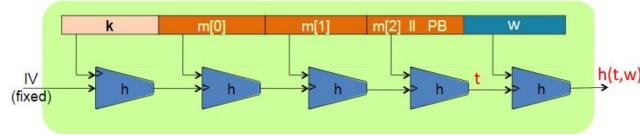
- **Pseudo Random Function (PRF)**
  - *CBC-MAC*
  - *NMAC*
  - *PMAC*
- **Collision Resistance Hash Function (CRHF)**
  - *HMAC*

**HMAC** Utilizza le hash functions. L'implementazione può essere la seguente:



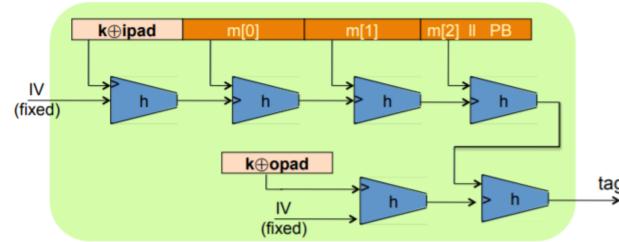
Questo schema di realizzazione è insicuro, in quanto è facile ottenere collisioni.

Si può dimostrare con un caso: supponendo di avere una coppia  $(m, t)$  tale che  $t = S(k, m)$ . Possiamo forgiare l'HMAC firmando un messaggio:  $m^0 = m||padding||$



In tal modo, facciamo sì che  $w$  sia preso in ingresso dall'ultimo blocco della funzione di compressione  $h$ : questo blocco avrà in ingresso  $w$  e  $t$ , cioè il risultato del penultimo blocco  $h$ .  $t$  non è altro che il risultato di  $S(m||padding, k)$ , mentre  $w$  possiamo controllarlo: attraverso la existential forgery.

**HMAC Standard** Il problema precedente può essere risolto applicando un ipad (di input) alla chiave e aggiungendo un blocco finale in cui è presente l'hash di  $k \oplus ipad || m$ .



L'HMAC si calcola quindi come:

$$t = h[(k \oplus opad) || h[(k \oplus ipad) || m]]$$

dove  $opad, ipad$  sono sequenze di bit fissate specificate nello standard (predefinite). Lo standard utilizza SHA-256 (PRF).

TLS: HMAC-SHA1-96

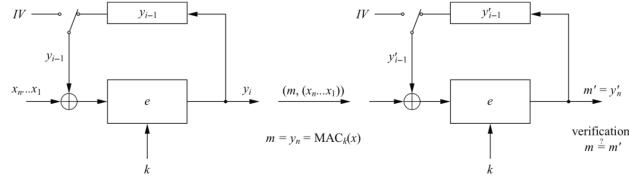
- SHA1 non è resistente alle collisioni, ma HMAC ha bisogno soltanto che la funzione di compressione sia PRF.

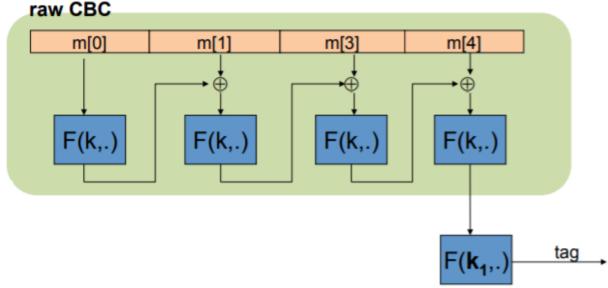
Limiti di sicurezza: Probabilità che dopo  $q$  MACs, HMAC diventi insicuro:  $\frac{q^2}{|T|}$

SHA-256:  $q << 2^{128} \Rightarrow$  bene

Uno dei vantaggi di questa implementazione è che se utilizzo anche uno SHA deprecated, l'HMAC risulta sempre ottimo. Necessita solo che le compression function siano a PRF.

**MACs from block ciphers: CBC-MAC** Metodo CBC:





La  $F$  indica il fatto che può essere una funzione generica, non per forza la encryption. Possiamo provare che questa tecnica può causare collisioni. Per evitarle, viene utilizzata una nuova funzione  $F$  con una chiave differente ( $k_1$ ).

Questa tecnica non viene utilizzata in pratica a causa delle due chiavi. Dopo aver generato un numero di *tag*, la chiave deve essere rigenerata.

Quanti messaggi posso effettuare con CBC-MAC usando le stesse chiavi?

Considerando  $q$  come il numero di messaggi con CBC-MAC con la stessa chiave  $k$ . Può essere provato che dopo  $q$  messaggi, la probabilità  $P$  che MAC diventi insicuro è pari a  $\frac{q^2}{|X|}$

- AES:  $|X| = 2^{128}$ ,  $P < \frac{1}{2^{32}} \Rightarrow q < 2^{48} \Rightarrow$  bene
- 3DES:  $|X| = 2^{64}$ ,  $P < \frac{1}{2^{32}} \Rightarrow q < 2^{16} \Rightarrow$  male

Se  $F$  è una funzione di Encrypting, l'input deve essere della dimensione di un blocco. Quest'ultima implica inoltre l'utilizzo del padding.

Padding deve essere generato attraverso un algoritmo per evitare collisioni.

**Padding** Il padding va scelto con cura:

- Usare degli zero è insicuro perché  $pad(m)$  e  $pad(m||0)$  hanno lo stesso MAC
- Il padding deve essere una funzione invertibile, quindi  $\forall(m_i, m_j) : m_i \neq m_j$   $pad(m_0) \neq pad(m_1)$

Lo standard ISO consiglia di:

- Usare "100...00" come padding, l'1 individua l'inizio.
- Aggiungere un blocco dummy quando la dimensione del messaggio è multipla di quella del blocco.

Il problema di questo schema è che un avversario può creare un'altra coppia  $(m, t)$  senza conoscere la chiave, se la lunghezza del messaggio è soltanto di un blocco. Deve soltanto appendere  $m \oplus t$  al messaggio originale  $m$  cosicché l'ultimo criptaggio produce lo stesso digest con  $m' = m||(m \oplus t)$ :

$$E_k(t \oplus (m \oplus t)) = t$$

## 4 Side Channel Attack

I Side Channel Attack sono un tipo di attacco che ha come obiettivo quello di attaccare l'implementazione degli algoritmi: sfruttano, ad esempio, le relazioni tra gli input (chiave) e l'energia o il tempo richiesto dall'algoritmo, per avere informazioni sul segreto (plaintext, chiave ecc...).

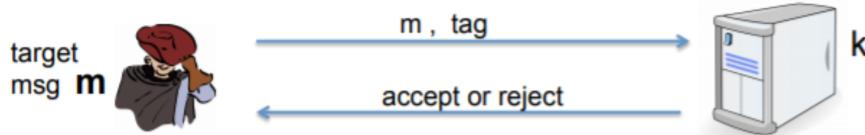
Affinché questo tipo di attacco funzioni, l'avversario deve:

1. conoscere l'implementazione dell'algoritmo
2. avere l'accesso fisico al dispositivo che lo implementa (tipicamente si attaccano le smartcard e sistemi embedded).

Ci sono 3 tipi di attacchi Side Channel:

- **Timing Attack:** cerco una relazione fra la chiave ed il tempo di esecuzione dell'algoritmo.
- **Power Attack:** cerco una relazione fra la chiave e l'energia consumata
- **Failure Injection Attack:** provo a portare il sistema in una situazione di errore o stato particolare per studiarne il comportamento.

Non esiste una vera teoria dietro un side channel attack, dipende totalmente dall'implementazione. Comunque questi attacchi sono possibili e piuttosto pericolosi.



Un avversario vuole inviare a  $k$  un messaggio e farglielo accettare come autentico. Per fare ciò deve inviare un *tag* corretto ( $t$ ). Come può l'avversario forgiare un  $t$  corretto? Sapendo che il MAC è un algoritmo sicuro, è inutile andare ad attaccare direttamente l'algoritmo. Si può utilizzare un particolare *side channel attack*.

**Timing Attack** Viene implementato nella libreria Keyczar Crypto Library (Python). Nel MAC è necessario confrontare il tag, quindi quello che fa  $K$  sostanzialmente è:  $t == S(k, m)$ . I byte vengono confrontati uno ad uno e viene data una risposta alla sorgente, dicendo se il tag fosse corretto o meno. Il confronto, effettuato byte a byte, appena individua due differenti invia il messaggio *rejected*. Da questo possiamo affermare che se i byte differenti sono gli ultimi, la risposta *rejected* viene inviata dopo aver trascorso il tempo massimo di confronto.

In questo modo, l'attaccante può sfruttare questa informazione attraverso un *Timing Attack*: a seconda del tempo di risposta, posso sapere quanti byte del tag ho azzeccato e continuare a cercare quelli successivi.

Ogni byte ha 260 ( $0x00, 0xFF$ ) tentativi e in totale i tentativi sono  $20(\text{dimensione tag}) \times 256$ , che è molto inferiore ai  $2^{160}$ , che è il costo del *guessing attack*.

Per risolvere questo problema, si deve ridefinire l'operatore " $==$ " in un'operazione che ha costo in tempo sempre equivalente.

Si ha, però, un calo delle performance e il compilatore potrebbe andare a creare nuovamente una possibilità di Timing Attack. L'avversario, infatti, potrebbe conoscere il tag che viene controllato. E quindi si effettua un'ulteriore modifica per togliere la dipendenza. Si rieffettua l'HMAC sui tag togliendo la dipendenza e la possibilità di sapere dove cambiare byte all'avversario.

**Fault Injection Attack** RSA è un algoritmo non computazionalmente semplice, quindi, per aumentare le performance, si potrebbe pensare di trovare un’implementazione ottimizzata. Un esempio è l’implementazione del *Chinese Remainder Theorem*, il quale ci permette di cifrare e decifrare con RSA in maniera più efficiente di quella canonica.

Il problema è quello di calcolare, efficientemente,  $y = x^d \text{mod}(n)$ . Il teorema descrive i passi:

1. Calcolare  $x_p = x \cdot \text{mod}(p)$ ,  $x_q = x \cdot \text{mod}(q)$
2. Calcolare  $y_p = x_p^{d \cdot \text{mod}(p-1)} \cdot \text{mod}(p)$ ,  $y_q = x_q^{d \cdot \text{mod}(q-1)} \cdot \text{mod}(q)$
3. Calcolare  $y = a_p y_p q + a_q y_q p$ , dove  $a_p$  e  $a_q$  sono dei coefficienti precalcolati.

Qual è il vantaggio rispetto all’algoritmo canonico?

$y_q$  e  $y_p$  sono le quantità più difficili da calcolare. Usando il metodo *square-and-multiply*, sono richieste  $\#\text{moltiplicazioni} + \#\text{quadrati} = 1.5t$  operazioni, in media. Considerando che ogni operazione di moltiplicazione e quadrato lavora su operandi da  $\frac{t}{2}$  bit, possiamo notare che la complessità dell’algoritmo è  $O(\frac{t^2}{4})$ . Si ottiene quindi un boost di circa 4 volte rispetto all’algoritmo canonico.

Tuttavia, questo algoritmo è vulnerabile ad un attacco di tipo fault-injection: creando un fault nel dispositivo, l’avversario è in grado di fattorizzare  $n$ .

Il problema creato dall’algoritmo è che non è possibile cancellare  $p$  e  $q$  finché non calcoliamo  $y$ , i quali non devono entrare in mano dell’avversario. L’attacco consiste nel causare un fault quando il dispositivo sta calcolando  $y_p$ , lasciando però intatto  $y_q$ , il dispositivo quindi calcolerà:

$$y' = a_p y'_p q + a_q y_q p$$

Una volta ottenuto  $y'$ , l’avversario calcola:

$$y - y' = a_p (y_p - y'_p) q$$

$p$  è stato eliminato dal risultato, a questo punto basta risolvere il problema:

$$\gcd(y - y', n) = q$$

attraverso l’algoritmo di Euclide, diventa risolvibile in modo efficiente.

Questo tipo di vulnerabilità può essere limitata introducendo schermature oppure circuiti di controllo di errori/fault (che però possono rallentare il dispositivo).

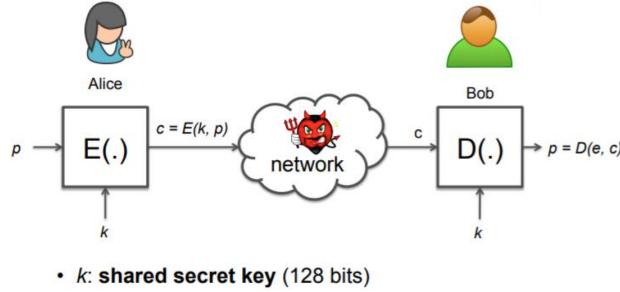
**Power Analysis** L’avversario monitora i consumi di energia del dispositivo per poi usare, ad esempio, tecniche come la *Simple Power Analysis (SPA)* o la *Differential Power Analysis (DPA)*, per carpire informazioni sul segreto. Questo tipo di attacco non è invasivo.

Con il metodo SPA sostanzialmente si usa un oscilloscopio per monitorare, ad occhio, i consumi del dispositivo. Mentre, con il metodo DPA, che è più complesso, si utilizza sia un oscilloscopio e una macchina per la registrazione dei dati dei consumi, successivamente attraverso analisi statistiche, si elimina il rumore e si estraggono i dati significativi.

## 5 Key establishment

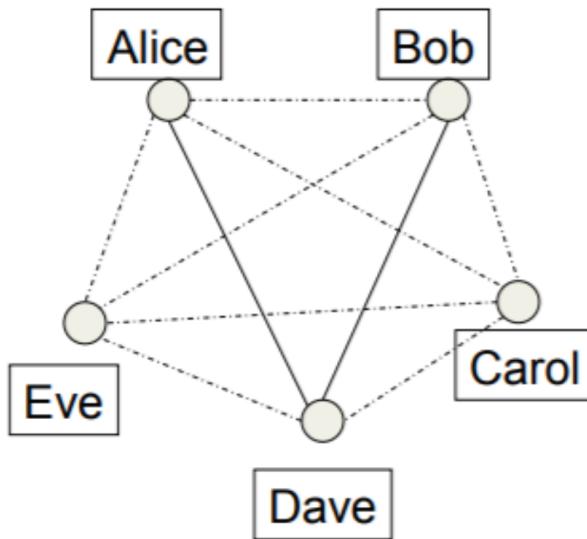
Per come abbiamo visto i sistemi Encrypt/Decrypt, la chiave utilizzata era nota a entrambe le parti. L'idea che ci si deve porre adesso, è su come condividere la chiave attraverso il network. La trasmissione della chiave nel network presenta molte difficoltà in quanto:

- Una trasmissione in chiaro è impossibile, una qualsiasi persona potrebbe leggerla.
- Una chiave criptata, avrebbe comunque bisogno di un'ulteriore chiave di criptazione.



### 5.1 Pairwise Keys

Supponendo di avere un sistema con un certo numero di utenti: ciò che si può fare, è di stabilire delle *Pairwise Key*: ogni utente mantiene una chiave segreta (long-term secret key) per comunicare con ogni utente del sistema.



L'utilizzo di questo approccio introduce alcune proprietà:

- ogni utente deve mantenere  $(n - 1)$  chiavi ( $n$  è il numero di utenti nel sistema).
- In totale, nel sistema abbiamo un numero di chiavi sull'ordine di  $n^2$   $O(n^2)$
- L'ingresso o l'uscita di un utente nel sistema affligge tutti gli altri utenti. Ogni utente deve stabilire/eliminare la chiave per comunicare con quest'ultimo.

Come effettuano gli utenti questo tipo di condivisione? Il problema è ancora attivo, dato che non si sa come condividere la chiave. Un sistema potrebbe essere quello di condividere

le chiavi prima che il sistema sia ON.

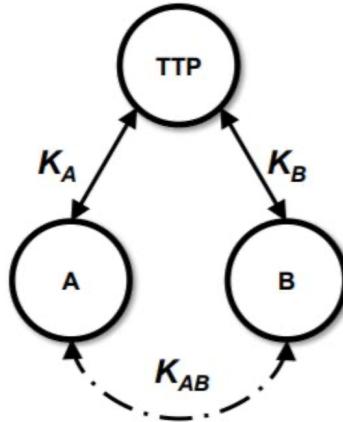
L'utilizzo di questa tipologia di rete ha dei pro e dei contro:

- **PRO:** Se un utente viene compromesso, un avversario riesce a ottenere le sue chiavi, solo le comunicazioni verso questo utente vengono compromesse. Le altre comunicazioni nel sistema non risentono di questo attacco.
- **CONTRO:** Il sistema è poco scalabile: il numero delle chiavi complessive nel sistema è il quadrato del numero degli utenti e l'ingresso o uscita di un nuovo utente affligge ogni utente già presente nel sistema.

Per risolvere il Contro di questo approccio (scalabilità), si utilizza un'architettura centralizzata.

## 5.2 Trusted Third Party - TTP

Ogni utente condivide una chiave (long-term key) con il Trusted Third Party, andando a risolvere i problemi che affliggevano il sistema Pairwise: il numero totale di chiavi da mantenere e l'ingresso/uscita di nuovi utenti.



In questo sistema, ogni utente deve mantenere in memoria la chiave long-term stabilità con il TTP, facendo raggiungere il numero massimo di chiavi nel sistema di  $n$ . Se un utente ha la necessità di comunicare con un altro utente, si stabilisce una chiave temporanea (*short-term key*), sfruttando la long-term key per la comunicazione con il TTP.

Alice: "Voglio comunicare con Bob"  
 Alice → TTP

Il TTP genera  $K_{ab}$  e lo manda (2 volte), cifrandolo sia con  $K_a$  che con  $K_b$ :

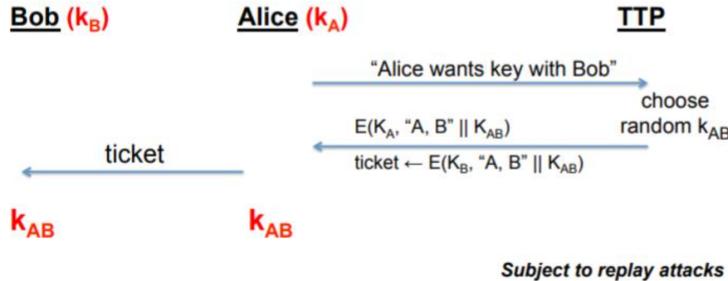
$$E(K_a, "A, B" || K_{ab}), E(K_b, "A, B" || K_{ab}) \\ \text{Alice} \leftarrow \text{TTP}$$

Alice ora può ottenere la chiave, decifrando il messaggio con  $K_a$ . Resta solo da comunicare la chiave a Bob, che è già stata cifrata dal TTP:

$$E(K_b, "A, B" || K_{ab}) \\ \text{Alice} \rightarrow \text{Bob}$$

Un approccio centralizzato introduce i problemi rilevabili in un qualsiasi sistema che utilizzi lo stesso approccio:

- TTP è un single-point-of-failure
- TTP deve essere sempre attivo
- TTP è a conoscenza di tutte le chiavi
  - Teoricamente, potrebbe leggere le conversazioni di ogni utente.



Il secondo campo (ticket), è criptato con  $K_b$ . È un protocollo semplice, ma anche errato dato che non contiene nessuna informazione sul tempo, quindi uno stesso pacchetto può essere riutilizzato: *subject to replay attack*.

Il punto interessante di questo sistema è che usando long-term key, un utente può ottenere una chiave condivisa per comunicare con altri utenti.

Come il Pairwise, questo approccio ha pro e contro:

- **PRO:**

- È facile aggiungere o togliere utenti dal sistema.
- Ogni utente deve mantenere in memoria solo una long-term secret key.

- **CONTRO:**

- TTP deve essere sempre online
- TTP deve essere efficiente
- TTP deve mantenere in memoria  $n$  chiavi.
- Se il TTP viene compromesso, tutte le comunicazioni sono insicure.

Si deve quindi trovare un sistema per generare chiavi condivise senza l'utilizzo di un TTP o condividerle prima che il sistema sia ON (Pairwise).

### 5.3 Diffie-Hellman

Questo protocollo permette di scambiare una chiave in maniera sicura anche usando un canale non sicuro.

Questo protocollo (del 1978) ha posto le basi della criptazione a chiave pubblica. Sfrutta alcune semplici proprietà matematiche:

- La moltiplicazione è un'operazione commutativa:

$$(a \cdot b) = (a \cdot b) \text{mod}(n)$$

- Potenza di potenza è un'operazione commutativa:

$$(a^b)^c = (a^c)^b$$

DH genera alcuni parametri:

- $p$  è un numero primo molto grande (2048 bit)

- generatore  $g : \forall y, 1 \leq y < p, \exists x : y = g^x \text{ mod } p$  (primitive root)

Sia  $p$  e  $g$  sono pubblici. Ogni peer genera la chiave privata e calcola la chiave pubblica attraverso essa. La chiave di sessione sarà:

$$k_{ab} = g^{ab} \text{ mod } p$$

Utilizza due operazioni matematiche:

- **Discrete exponentiation:** Dati  $p$ ,  $g$  e  $x$ , calcolare  $y = g^x \text{ mod } p$  è computazionalmente semplice.
- **Discrete logarithm:** Dato  $g$ ,  $1 \leq y \leq p - 1$ , è computazionalmente difficile determinare  $x, 1 \leq x \leq p - 1 : y = g^x \text{ mod } p$ .

### 5.3.1 Descrizione del protocollo nel dettaglio

- $p$  e  $g$  sono due quantità pubbliche e conosciute da entrambi le parti.
- $p$  è un numero primo molto grande (1024-2048 bit).
- $g$  è un parametro tale che  $1 \leq g < p$

**Protocollo:**

- *Fase 1:* Alice e Bob generano, rispettivamente, due numeri randomici  $a$  e  $b$ .
- *Fase 2:* Alice e Bob calcolano, rispettivamente

$$y_a = g^a \text{ mod } p$$

$$y_b = g^b \text{ mod } p$$

- *Fase 3:* Entrambi si scambiano, in chiaro, i parametri  $y_a$  e  $y_b$
- *Fase 4:* Le quantità ricevute vengono elevate per il numero generato inizialmente, quindi

$$\text{Alice } (y_b)^a = (g^b)^a \text{ mod } p = g^{ab} \text{ mod } p = K_{ab}$$

$$\text{Bob } (y_a)^b = (g^a)^b \text{ mod } p = g^{ab} \text{ mod } p = K_{ab}$$

Sia Bob che Alice adesso hanno la chiave.

- *Fase 5:*  $a$  e  $b$  vengono eliminati, per garantire la sicurezza del protocollo.

**Sicurezza del protocollo:** Le uniche quantità che l'avversario può conoscere sono:

- $p, g$  perché sono pubbliche.
- $y_a, y_b$  perché trasmesse in chiaro dalle 2 parti.

L'unica maniera per calcolare la chiave è risalire ad  $a$  oppure a  $b$ : per calcolare  $a$  è necessario trovare  $a$  tale che  $y_a = g^a \text{ mod } p$  e stessa cosa per  $b$  tale che  $y_b = g^b \text{ mod } p$ .

Entrambe le formule rappresentano operazioni di **discrete logarithm**, le quali, sono computazionalmente difficili da realizzare.

**Complessità del Discrete Logarithm** Per poter essere sicuro  $a$ ,  $b$  e  $p$  devono essere sufficientemente larghi, su  $n$ -bit.

Prendiamo un numero primo  $p$  tale per cui  $p < 2^n$ . Tutte le quantità sono rappresentabili su  $n$ -bit.

L'algoritmo più efficiente per il calcolo del discrete logarithm richiede  $p^{\frac{1}{2}} = 2^{\frac{n}{2}}$  operazioni. Esempi di  $n$  sono da 512 bit in su: l'operazione, in questo caso, richiederebbe  $2^{256} = 10^{77}$  operazioni.

**Complessità della Discrete Exponentiation**  $Y_a = g^a \cdot \text{mod}(p)$ : cosa è  $g^n$ ?

**Grade-School-Algorithm:**  $\underbrace{g \cdot g, \dots, g}_{a-1 \text{ volte}}$

$a$  deve essere "larga" quanto  $p \rightarrow a < 2^n$

Ne esiste uno più efficiente, chiamato *square and multiply*, che si ottiene andando a scomporre l'esponenziazione.

Considerando  $a$  nella sua rappresentazione binaria:

$$a = (a_{n-1}, a_{n-2}, \dots, a_1, a_0)$$

$$a = a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \dots + a_12^1 + a_0$$

Eleviamo  $g$  ad  $a$ :

$$\begin{aligned} g^a &= g^{a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \dots + a_12^1 + a_0} = \\ &= g^{a_{n-1}2^{n-1}} \cdot g^{a_{n-2}2^{n-2}} \cdot \dots \cdot g^{a_12^1} \cdot g^{a_0} = \end{aligned}$$

Fattorizzo tutto eccetto  $g^{a_0}$ :

$$\begin{aligned} &= (g^{a_{n-1}2^{n-2}} \cdot g^{a_{n-2}2^{n-3}} \cdot \dots \cdot g^{a_1})^2 \cdot g^{a_0} = \\ &= ((g^{a_{n-1}2^{n-3}} \cdot g^{a_{n-2}2^{n-4}} \cdot \dots \cdot g^{a_2})^2 \cdot g^{a_1})^2 \cdot g^{a_0} = \\ &= (\dots ((g^{a_{n-1}})^2 g^{a_{n-2}})^2 \dots g^{a_1})^2 g^{a_0} \end{aligned}$$

Dove  $a_i$  appartiene a  $\{0, 1\}$ .

Quando  $a_1$  è uguale a 0, la moltiplicazione non è necessaria, dato che l'esponenziale è uguale a 1. Questo semplifica ulteriormente in termini di numero di istruzioni necessario per questa operazione.

Il costo complessivo massimo è di  $2n$  moltiplicazioni. Con  $n = 1024$  sono necessarie 2048 moltiplicazioni.

L'operazione logaritmica deve essere difficile e invece necessita di una complessità di risoluzione di  $p^{\frac{1}{2}} = 2^{\frac{n}{2}}$  operazioni. Il parametro  $p$  indica quindi quanto è sicuro il sistema di Diffie-Hellman.

Se ad esempio prendiamo  $n = 512$ :

- L'esponenziale richiede 1024 operazioni.
- L'operazione logaritmica invece ne richiede  $2^{256}$  che equivalgono a  $10^{77}$  operazioni.

A seconda della dimensione della chiave, si necessita di più o meno bit per rappresentare  $p$ .

<b>Cipher key</b>	<b>modulus</b>	<b>elliptic curve</b>
<b>size</b>	<b>size</b>	<b>size</b>
80 bits	1024 bits	160 bits
128 bits	3072 bits	256 bits
256 bits (AES)	15360 bits	512 bits

*slow*

L'ottenimento di una chiave di 256 bits necessita un parametro del modulo molto elevato, il quale rende l'operazione molto lenta.

La curva ellittica è un nuovo modo matematico (estensione di DH), che non utilizza l'operazione "mod". Tramite questo nuovo metodo, possiamo ottenere con un numero di

bit necessari molto inferiore, la stessa sicurezza.

DH è sicuro contro un avversario passivo, cioè che si limita a prendere informazioni della rete, ma non contro avversari attivi. Nella crittografia, un attacco attivo su un sistema di comunicazione è uno in cui l'attaccante cambia la comunicazione. Può creare, forgiare, alterare, sostituire, bloccare o reindirizzare i messaggi. Ciò contrasta con un attacco passivo in cui l'attaccante si limita ad origliare: può leggere messaggi che non dovrebbe vedere, ma non altera i messaggi.

### 5.3.2 Attacchi al protocollo

**Passive attack** DH è sicuro nei confronti di un attacco passivo: un avversario passivo può soltanto vedere  $Y = g^x \bmod p$ . Per recuperare la chiave  $x$ , deve risolvere il problema del *discrete logarithm*.

### 5.3.3 Active attack

Un attacco *Man-In-The-Middle (MITM)* può rompere DH. Un avversario può stabilire una chiave di sessione con entrambe le parti e agire come membro attivo. Non c'è niente che linki la chiave pubblica  $Y$  ad un identificatore.

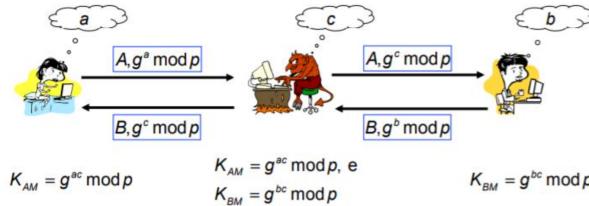
**Man-In-The-Middle (MITM)** Se l'avversario diventa attivo, cioè inizia ad alterare il flusso dei pacchetti, può effettuare un attacco di Man-in-the-middle.

L'avversario si interpone, quindi, tra Alice e Bob, sostituendo i valori  $y_a$  e  $y_b$  trasmessi, con  $y_c = g^c \bmod p$  ricordando che  $p$  e  $g$  sono pubblici, mentre  $c$  è un numero casuale che viene generato dall'avversario. Quindi, Alice otterrà la chiave:

$$(y_c)^a \bmod p = g^{ac} \bmod p = K_{ac}$$

Bob otterrà la chiave:

$$(y_c)^b \bmod p = g^{ab} \bmod p = K_{cb}$$



A questo punto l'avversario può decifrare i messaggi di Alice (usando la chiave  $K_{ab}$ ), per poi cifrarli con la chiave  $K_{cb}$  e trasmetterli a Bob. Lo stesso discorso può essere fatto nel senso opposto. Le due parti non si accorgono nemmeno dell'intromissione dell'avversario.

Il problema, in questo caso, è che i valori  $y_a$  e  $y_b$  non sono associati, in alcun modo, alle identità di Alice e Bob. Quindi, un semplice database centralizzato non è sufficiente a risolvere il problema, in quanto l'avversario potrebbe alterare le chiavi trasmesse agli utenti, in uscita dal database.

Questo, quindi, è un problema di autenticazione che fa sorgere la necessità di introdurre i *certificati*.

## 6 Public key encryption

È un sistema di criptaggio asimmetrico e rispetto al modello simmetrico, è prevista una coppia di chiavi  $pubk, privk$  invece di una sola chiave.

$privk$  è utilizzata dal ricevente e non deve essere compromessa, mentre  $pubk$  è disponibile a tutti.

Si hanno ancora due funzioni  $E$  e  $D$ , che vengono descritte in questo modo:

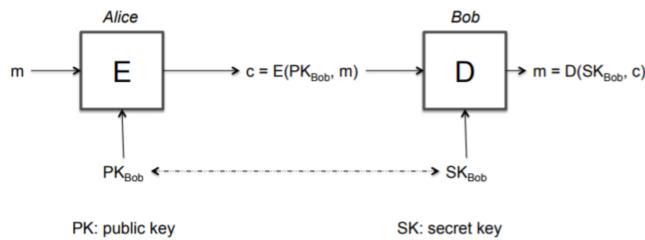
$$c = E(pubk, m)$$

$$m = D(privk, c)$$

La proprietà di consistenza è descrivibile come:

$$\forall privk, pubk \quad D(privk, E(pubk, m)) = m$$

La crittografia pubblica è utilizzata per stabilire la chiave.



Il modello indica il meccanismo generale sul quale si basa la crittografia a chiave pubblica: Alice vuole comunicare con Bob, critta il messaggio con la chiave pubblica di Bob e lo invia. Questo messaggio può ora essere decriptato solo tramite la chiave privata che è in possesso solo di Bob.

### 6.1 Security properties

- Dato  $pubk$  e  $y$  deve essere "difficile" trovare  $x : E(pubk, x) = y$
- Conoscendo  $pubk$ , deve essere "difficile" trovare  $privk$

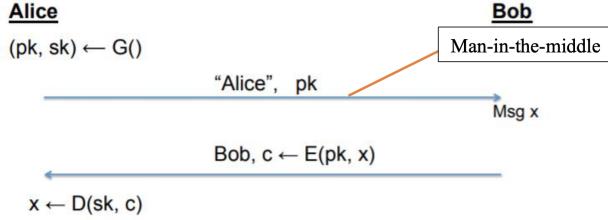
Per "difficile" si intende che non esiste un algoritmo in grado di risolvere quel problema in maniera efficiente.

Non esiste uno schema di public key perfetto

1. Supponiamo che l'avversario intercetti  $y$
2. L'avversario seleziona  $x_0 : P(X = x_0) \neq 0$
3. Calcola  $y_0 = E(pubk, x_0)$
4. **if**  $y == y_0$   $x$  è trovato  
**else**  $P(X = x_0 | Y = y) = 0$

Le probabilità *a-priori* e *a-posteriori* sono differenti e violano la definizione di Shannon di perfect secrecy.

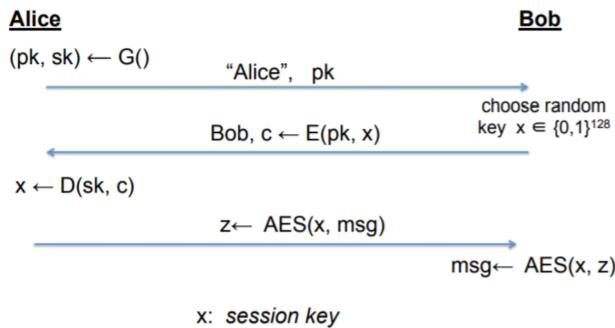
Per quanto riguarda la sicurezza computazionale:



Si utilizza  $E$  per criptare il messaggio. Il problema del Man-in-the-middle è sempre presente ed è risolto tramite l'utilizzo del certificato.

Dalla figura si vede inoltre che il messaggio  $x$  viene criptato interamente, questo però comporta problemi in quanto il messaggio potrebbe essere molto grande, andando a penalizzare le prestazioni del sistema.

Un metodo più efficiente è:



Tramite la crittografia a chiave pubblica, possiamo superare il più grande problema della crittografia simmetrica, cioè la condivisione della chiave, a discapito dell'esistenza del problema man-in-the-middle. Per risolverlo, quindi, basta aggiungere un certificato alla prima transizione.

## 7 RSA

È attualmente il sistema asimmetrico più utilizzato nel mondo. Il suo largo utilizzo è dovuto al fatto che è molto resistente, non può essere attaccato l'algoritmo ma solo l'implementazione attraverso attacchi *side-channel*.

È utilizzato in diverse applicazioni, come la cifratura di piccoli messaggi o forme digitali. Lo schema introduce 3 funzioni:  $G$ ,  $E$ ,  $D$ .

### 7.1 Key generation algorithm $G$

1. Selezionare due numeri primi "grandi" (512/1024 bit)  $p$  e  $q$
2. Calcolare  $modulus n = p \cdot q$  (modulus) e  $Eulero's totient \Phi = (p - 1) \cdot (q - 1)$
3. Seleziona un numero random  $e$  tale per cui:  $1 \leq e < \Phi : MCD(e, \Phi) = 1$  ( $e$  e  $\Phi$  compresse)
4. Calcolare l'intero  $d$ , tale che:  $1 \leq d < \Phi : e \cdot d = 1 \text{ mod } \Phi$ , i.e.  $e \cdot d = r \cdot \Phi + 1$  per alcuni  $r$
5.  $pubk = (e, n)$  e  $privk = (d, n)$
6. Distruggere  $p$  e  $q$

## 7.2 Encryption E

Per generare il ciphertext  $c$  da un messaggio  $m$ , l'algoritmo di encryption segue i seguenti passaggi:

1. Ottenerne l'autentica chiave pubblica  $(n, e)$  del destinatario del messaggio.
2. Considerare il messaggio come un numero intero  $m$  nell'intervallo  $[0, n - 1]$ . Vedremo successivamente che se  $m$  è un numero intero superiore a  $n$  verrà spezzato in più parti, ognuna delle quali con valore inferiore a  $n$ .
3. Viene calcolato  $c = m^e \cdot mod(n)$
4.  $c$  viene a questo punto inviato verso la destinazione.

$$x \in [0, n - 1], y = x^e \mod n$$

## 7.3 Decryption D

Il destinatario alla ricezione del ciphertext  $c$  per ottenere il messaggio originale  $m$  deve utilizzare la sua chiave privata  $(d, n)$  nella seguente operazione  $m = c^d \cdot mod(n)$ .

$$y \in [0, n - 1], x = y^d \mod n$$

Entrambi gli algoritmi utilizzano l'operazione matematica *modular exponentiation*.

## 7.4 Square-and-multiply algorithm

Questo algoritmo è utilizzato per effettuare l'esponenziazione modulare  $x^e \mod n$  con una complessità di  $O(k^3)$

---

```
c ← 1
for all bit in e do
    c ← c2 mod n
    if bit == 1 then
        c ← c × x mod n
    end if
end for
```

---

L'efficienza è raggiunta scegliendo  $e$  con pochi bit uguali a 1, ad esempio  $2^{16} + 1$ .

## 7.5 RSA problem

Il calcolo di  $x$ , conoscendo  $(e, n)$  e  $y$ , è relativo al problema della *fattorizzazione*. Calcolare il componente di decriptazione  $d$  dalla chiave pubblica  $(e, n)$  è computazionalmente equivalente a fattorizzare  $n$ .

### 7.5.1 RSAP

Definizione: Ottenerne il plaintext  $m$  dal ciphertext  $c$ , avendo a disposizione solo la chiave pubblica  $(n, e)$ .

Questo problema (RSAP) che abbiamo detto essere riconducibile alla fattorizzazione, in realtà non è vero. RSAP è *al massimo* equivalente o inferiore alla fattorizzazione, non maggiore.

Osservazioni sulla sicurezza di RSA:

**Teorema:** Calcolare la  $d$  avendo a disposizione la chiave pubblica  $(n, e)$ , è computazionalmente equivalente a fattorizzare  $n$ .

- Se l'avversario può in qualche modo fattorizzare  $n$ , allora può calcolare la chiave privata  $d$  in modo efficiente
- Se l'avversario riuscisse in qualche modo a calcolare  $d$ , potrebbe successivamente calcolare  $n$  in modo efficiente.

**e-th root:** visto che  $c = m^e \cdot \text{mod}(n)$ , allora potremmo pensare di trovare  $m$  facendo:

$$m = \sqrt[e]{c} \cdot \text{mod}(n)$$

- Calcolare la radice  $e$ -th è un problema computazionalmente facile, se e solo se  $n$  è un numero primo.
- Se  $n$  è un numero composto, il problema del calcolo della radice è equivalente alla fattorizzazione.

Si deve quindi aggiungere una proprietà:  $p$  e  $q$  devono essere grandi, primi e inoltre  $(p - q)$  non può essere piccolo.

## 7.6 RSA security

La versione base *Plain RSA*, che è la definizione base dell'algoritmo, non è utilizzata in pratica per vari motivi.

### 7.6.1 Small plaintext problem

Se il plaintext è troppo piccolo, il ciphertext risultante è equivalente al plaintext.

Ovvero, si va ad attaccare lo spazio dei messaggi: l'avversario è in grado di trovare il plaintext dopo un certo numero di iterazioni. Per risolvere il problema bisogna ingrandire questo spazio, ad esempio accodando al messaggio inviato un numero randomico di una certa dimensione: questo numero è chiamato *salt* e serve solo a questo scopo.

### 7.6.2 Malleability

Il problema della malleabilità è dovuta alla presenza dell'*omomorfismo*. Questa proprietà viene utilizzata anche in altri campi, per avere comportamenti avanzati. In questo caso specifico dell'RSA, questa proprietà introduce problemi.

Un avversario conosce  $c = p^e \cdot \text{mod } n$  e può calcolare

$$c' = c \cdot s^e \cdot \text{mod } n = p^e \cdot s^e \cdot \text{mod } n = (p \cdot s)^e \cdot \text{mod } n = p' \cdot \text{mod } n$$

dove  $s$  è un intero tale che  $\text{MCD}(s, n) = 1$ .

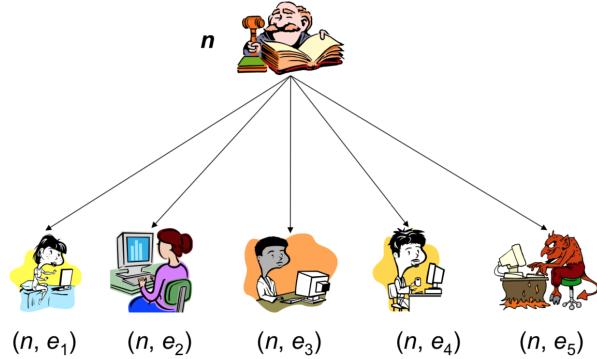
### 7.6.3 Low exponent attack

Sia  $e = 3$  tale che  $c_i = m^3 \cdot \text{mod } n_i$ . Un party invia  $c$  a 3 differenti riceventi che si calcolano  $m^3 = c_i \cdot \text{mod } n_i$ . Se  $n_1, n_2, n_3$  sono coprimi a coppie possiamo calcolare

$$x = m^3 \cdot \text{mod } n_1 n_2 n_3$$

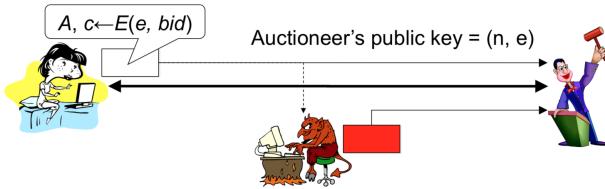
utilizzando il *Chinese Remainder Theorem* (CRT). Dato che  $m^3 < n_1 n_2 n_3$ ,  $x = m^3$ . Un avversario può calcolare  $m$  calcolando la radice cubica di  $m^3$ .

#### 7.6.4 Common modulus attack



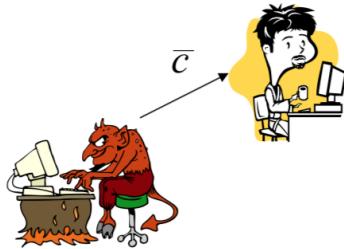
Il server utilizza un modulo comune  $n$  per tutte le coppie di chiavi. L'attaccante può fattorizzare efficientemente  $n$  da  $d_5$  e può poi calcolare tutti  $d_i$ .

#### 7.6.5 Chosen-plaintext attack



L'avversario critta tutte le possibili offerte (per esempio  $2^{32}$ ) finché trova un  $b$  tale che  $E(e, b) = c$ . Quindi, l'avversario manda un'offerta contenente la minima offerta per vincere l'asta:  $b' = b + 1$ .

#### 7.6.6 Adaptive chosen-plaintext attack



Bob decripta ciphertext tranne un certo ciphertext  $c$ . Mr. Loucipher vuole determinare il ciphertext corrispondente a  $c$ .

1. Mr. Loucipher seleziona  $x$  random, tale che  $MCD(x, n) = 1$  e invia a BOB la quantità  $\bar{c} = cx^e \bmod n$ ;
2. Bob la decripta producendo  $\bar{m} = (\bar{c})^d = c^d x^{ed} = mx \bmod n$ ;
3. Mr. Loucipher determina  $m$  calcolando  $m = \bar{m}x^{-1} \bmod n$ .

### **7.6.7 Countermeasure**

Aggiungere ridondanza e randomizzazione al messaggio originale attraverso lo standard *Optimal Asymmetric Encryption Padding*.

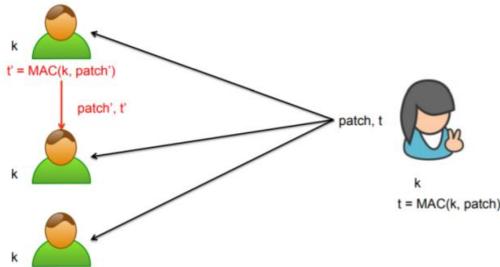
## 8 Digital signature

Digital Signature fornisce integrità e autenticazione dei messaggi, come MAC ma utilizzando un ambiente di chiavi asimmetriche. Garantisce la proprietà di non-ripudiazione perché soltanto un componente ha la chiave privata. Tutti possono verificare la firma. La firma digitale può essere costruita utilizzando RSA:

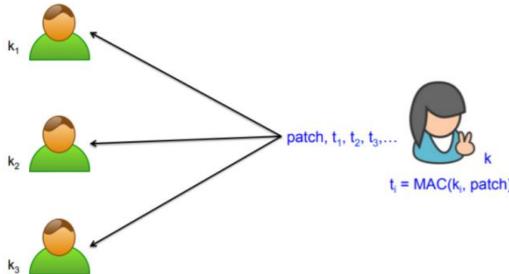
1. Generare una coppia di chiavi RSA: chiave pubblica  $(e, n)$  e chiave privata  $(d, n)$
2. Firma:  $\sigma = m^d \bmod n$
3. Verifica:  $m' = \sigma^e \bmod n$  e controlla se  $m == m'$

La funzione di firma equivale alla funzione di criptazione RSA, mentre l'operazione di verifica equivale a quella di decriptazione. Questa è una proprietà relativa solo a RSA e non vale in generale. Restano ancora valide tutte le altre proprietà.

**Firma digitale attraverso l'utilizzo del MAC** Col MAC, ogni utente conoscerebbe il  $k$  utilizzato per generare il MAC originale  $t$ :



Un utente malizioso utilizzando  $k$  potrebbe rigenerare un nuovo patch e ridistribuirlo. In un sistema trusted, è utilizzabile questo schema, ma non in un e-commerce. Per risolvere questo problema, si potrebbe generare un MAC per ogni utente. In questo caso lo schema sarebbe sicuro, ma risente nelle performance.



Comparazione dei due approcci:

- *Singola chiave condivisa*: un cliente può forgiare un nuovo *tag* rendendolo inutilizzabile in un sistema senza fiducia tra gli utenti.
- *Chiave per ogni collegamento point-to-point*: introduce overhead diventando ingestibile.

A questo punto, si può fare un confronto con la Digital Signature:

- Verificabilità pubblica:
  - DS: chiunque può verificare la firma.
  - MAC: solo chi è in possesso della chiave può verificare.

- Trasferibilità:
  - DS: può inviare la firma a chiunque
  - MAC: non si può.

La Digital Signature non è ripudiabile.

Il firmatario non può (facilmente) negare l'emissione di una firma. Questo è cruciale per applicazioni legali e il giudice può verificare la firma usando una copia della chiave pubblica.

I MAC, non possono fornire questa funzionalità: senza l'accesso alla chiave, non c'è nessun modo per verificare un *tag*. Anche se il ricevente perde la chiave per giudicare, come può il giudice verificare che la chiave sia corretta? Anche se la chiave è corretta, il ricevitore potrebbe aver generato il tag.

La proprietà principale della digital signature è la non replicazione. Nessuno oltre ad Alice può generare  $\sigma$ , in quanto viene generato dalla chiave privata. Nel MAC viene generata attraverso  $k$ , che è mantenuto da tutti gli utenti del sistema introducendo un fattore di replicazione.

**Definizione:** Una firma digitale è un numero che dipende da un segreto noto solo al firmatario (chiave privata) e, in aggiunta, dal contenuto del messaggio che viene firmato.

**Proprietà:** Una firma digitale deve essere verificabile.

In caso di controversia, una terza parte imparziale deve essere in grado di risolvere la disputa in modo equo, senza richiedere l'accesso al segreto del firmatario. Da questo si ottiene che la firma digitale è un meccanismo molto forte in quanto fornisce sia la **verificabilità** che la **non replicabilità**.

## 8.1 Schema generale

La firma digitale è strutturata da 3 algoritmi:

- L'algoritmo di generazione della chiave  $G()$ . Questo prende come input un numero  $n$  e restituisce le due chiavi  $(p_k, s_k)$ .
- Algoritmo di generazione della firma digitale  $S()$ . Prende come input la chiave privata  $s_k$  e un messaggio  $m$ . Fornisce come output la firma  $\sigma$ .
- Algoritmo di verifica della firma  $V()$ . Prende come input la chiave pubblica  $p_k$ , la firma  $\sigma$  e (opzionalmente) il messaggio  $m$ . Fornisce come output *true* o *false*.

Questi 3 algoritmi devono essere consistenti:

$$\text{Per ogni messaggio } m \text{ e chiavi } (p_k, s_k), V(p_k, [m], S(s_k, m)) = \text{true}$$

## 8.2 Schema di firma digitale attraverso RSA

È il più diffuso schema utilizzato per effettuare una firma digitale. È esattamente lo stesso algoritmo RSA utilizzato in fase di criptazione e quindi gode delle stesse proprietà. È l'unico caso in cui uno stesso algoritmo utilizzato per criptazione e firma digitale utilizza esattamente le stesse funzioni:

$$\begin{aligned} (e, n) &\text{ chiave pubblica} \\ (d, n) &\text{ chiave privata} \end{aligned}$$

Le funzioni descritte dallo schema sono:

- Firma:  $S(m, (d, n)) \rightarrow \sigma = m^d \cdot mod(n)$
- Verifica:  $V() \rightarrow \sigma^e \cdot mod(n)$

La funzione firma equivale alla funzione *criptazione* RSA, mentre l'operazione verifica equivale a quella di decriptazione. Dal punto di vista computazionale, le osservazioni fatte sono equivalenti.

### 8.2.1 Attacchi all'algoritmo

Partiamo dal presupposto che il verificatore della firma, deve avere la corretta chiave pubblica.

In accordo alla rottura di RSA, è necessario calcolare  $d$  (parte della chiave privata). Per fare ciò, l'attacco più generale è quello di fattorizzare  $n$ . Se questo modulo è sufficientemente grande, non è possibile ottenere  $d$ .

**Existential forgery** Equivale a riuscire a produrre una firma digitale valida su un messaggio random  $x$ . Il messaggio  $m$  è casuale e potrebbe non avere alcun significato di applicazione. Tuttavia, questa proprietà non è desiderabile.

**Malleabilità** Combinare due firme per ottenerne una terza (existential forgery).

L'attacco:

$$\begin{aligned} \text{Dato } \sigma_1 &= m_1^d \cdot mod(n) \\ \text{Dato } \sigma_2 &= m_2^d \cdot mod(n) \end{aligned}$$

Output  $\sigma_3 = (\sigma_1 \cdot \sigma_2) \cdot mod(n)$  che è una firma valida per  $m_3 = (m_1 \cdot m_2) \cdot mod(n)$

$$\text{Dimostrazione: } \sigma_3^e = (\sigma_1 \cdot \sigma_2)^e = \sigma_1^e \cdot \sigma_2^e = m_1 \cdot m_2 \cdot mod(n)$$

La risoluzione anche qua è l'utilizzo di una sorta di ridondanza o firmare l'hash.

Per risolvere ciò esistono diversi schemi:

- Probabilistic Signature Scheme (PSS) in PKCS#1
- Full Domain Hash (RSA-FDH)

## 8.3 Proprietà dalla funzione hash

Un problema da considerare, utilizzando questo algoritmo, è che  $m$  ha, in genere, una dimensione alta (pensare per esempio a documenti, file multimediali, patch ecc...) e quindi l'operazione di firma potrebbe risultare molto lenta. Per risolvere questo problema si possono utilizzare funzioni hash per sintetizzare  $m$  e poi firmare l'hash ottenuto.

La scelta di utilizzare una funzione hash per la firma digitale può comportare alcuni rischi e mettere in dubbio le garanzie di integrità, non ripudiabilità e autenticità delle firme. Per questo motivo è fondamentale verificare che le funzioni utilizzate rispettino alcuni requisiti: si fa riferimento alle proprietà indicate nel capitolo sulle funzioni hash.

**Pre-image resistance** La proprietà ci dice che dato un hash  $H(x)$  deve essere difficile trovare un'altra  $x'$  che abbia lo stesso hash. Se questa proprietà non è garantita avremo la *existential forgery*. Sia  $\sigma = h(m)^d \cdot mod(n)$ . Se  $h$  non è pre-image resistant:

1. Selezionare  $z < n$  (dimensione del messaggio)
2. Calcolare  $y = z^e \cdot mod(n)$ , di cui  $e$  ed  $n$  sono conosciuti perché provengono dalla chiave pubblica.
3. Trovare  $m' : h(m') = y$
4. Possiamo affermare che  $z$  è la firma digitale di  $m'$

**Second pre-image resistance** Sia  $\sigma = h(x)^d \bmod n$  la firma digitale di Alice. Se  $h$  non è 2nd-preimage resistance, un avversario può affermare che Alice ha firmato  $x'$  piuttosto che  $x$ .

**Collision resistance** Se  $h$  non è collision resistance, un componente non fidato può:

1. Scegliere  $m, m' : h(m) = h(m')$
2. Produrre  $\sigma = h(m)^d \bmod n$
3. Inviare  $(m, \sigma)$  a Bob
4. E affermare che ha effettivamente emesso  $(m', \sigma)$

## 8.4 Non ripudiabilità vs autenticità

**Definizione:** la non ripudiabilità impedisce a un firmatario di firmare un documento e successivamente essere in grado di negare con successo di averlo fatto. L'autenticità basata sulla crittografia simmetrica, consente a una parte di ottenere l'integrità/autenticità di un determinato messaggio in un dato momento  $t_0$ . La non ripudiabilità (basato su crittografia a chiave pubblica), consente a una parte di convincere gli altri in qualsiasi momento  $t_1 \geq t_0$  dell'integrità/autenticità di un dato messaggio al tempo  $t_0$ .

La firma digitale di Alice per un dato messaggio, dipende dal messaggio e un segreto noto solo ad Alice (la chiave privata). Bob verifica la firma digitale attraverso un altro valore: la chiave pubblica. L'autenticità del mittente di un messaggio fornita da una firma digitale è valida solo nel periodo di tempo nel quale è mantenuta la segretezza della chiave privata del firmatario.

Una minaccia che deve essere affrontata è un firmatario che rivela intenzionalmente la sua chiave privata e, successivamente, afferma che una firma precedentemente valida è stata falsificata.

Questa minaccia può essere affrontata da:

- Impedire l'accesso diretto alla chiave.
- Uso di un agente *timestamp* di fiducia.
- Uso di un notaio di fiducia.

## 9 Certificate

I certificati sono il meccanismo principale per affrontare il problema dell'autenticazione della chiave pubblica.

Un certificato, quindi, linka l'identificatore alla chiave pubblica.

$$CERT_A = (Alice, pubkey_A, L_A, SCA(Alice, pubkey_A, L_A))$$

Ogni certificato ha un periodo di validità ( $L_A$ ), dopo e prima il quale ogni firma applicata dalla persona non è valida. I certificati possono essere revocati, e questo avviene quando:

- il certificato scade
- la chiave privata è stata compromessa

A tal proposito, ogni *Certification Authority* pubblica una lista che contiene tutti i certificati revocati, chiamata *Certificate Revocation List (CRL)*. La lista viene firmata, per ovvi motivi, dalla stessa Certification Authority. L'indirizzo a questa lista è indicato in uno dei campi del certificato stesso, tra i quali è presente anche un campo che specifica per quali scopi la chiave pubblica associata può essere utilizzata (es. firma, cifratura).

### 9.1 Certification Authority (CA)

È un'entità (TTP) che si occupa di rilasciare i certificati, verificando l'identità del richiedente. In particolare deve:

- verificare che il nome della persona sia associato alla chiave da certificare
- verificare che la persona possa effettivamente usare la coppia di chiavi presentata
- stabilire le regole e le politiche per la verifica dell'identità delle persone (non è un task semplice, ad esempio in molti stati non esiste neanche il concetto di codice fiscale)

La certificazione si basa sulla *delega della fiducia*. Bob, ad esempio, si fida e delega la certification authority nel verificare l'identità di Alice e attestare l'autenticità della sua chiave pubblica. Se un certificato è firmato usando quello della Certification Authority, allora Bob si fida del certificato.

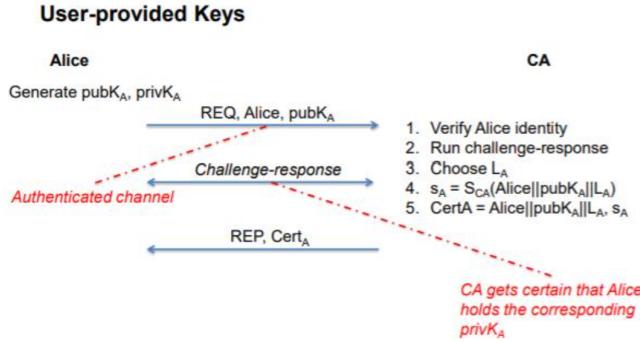
Un certificato, quindi, definisce un link indissolubile tra l'identità del soggetto e la sua chiave pubblica. Tuttavia, non specifica il significato del collegamento, i possibili usi e non garantisce l'affidabilità del soggetto. Il modello più semplice è il *Single CA*, nel quale abbiamo una Certification Authority che rilascia certificati per tutti gli utenti del sistema. I certificati hanno una scadenza, quindi sono associati ad un periodo di validità. Tuttavia, durante questo periodo, il certificato può essere revocato, e ciò succede quando:

- la chiave privata è compromessa
- il soggetto ha cambiato ruolo o cambiato l'organizzazione

La revoca deve essere fatta:

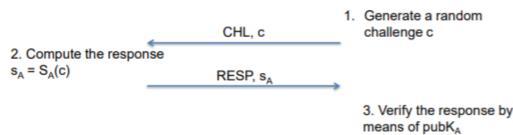
- correttamente, in quanto può essere solo convalidata da entità autorizzate (quali il proprietario e l'entità di certificazione)
- in tempo, deve essere diffusa a tutte le parti interessate nel minor tempo possibile

### 9.1.1 Generazione di un certificato



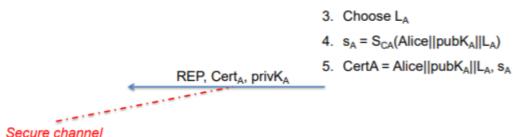
Con l'assunzione che il canale utilizzato sia un canale autenticato, non soggetto al MIM. Alice invia una REQ alla CA, indicando i due parametri che siano collegati, l'identità (Alice) e la chiave pubblica.

La CA a questo punto avvia una *challenge response* per avere la certezza che Alice tenta la corrispondente chiave privata. La challenge response è un protocollo in realtà molto semplice:



La CA calcola un valore random  $c$  e lo invia ad Alice, che a sua volta dovrà criptarlo tramite la chiave privata e rinviarlo. La CA verifica la risposta decriptando con la chiave pubblica già ricevuta.

Una volta effettuato questo protocollo, la CA produce il certificato, firmandolo attraverso la sua chiave privata e lo invia ad Alice attraverso un canale sicuro. ( $L_a$  periodo di validità)

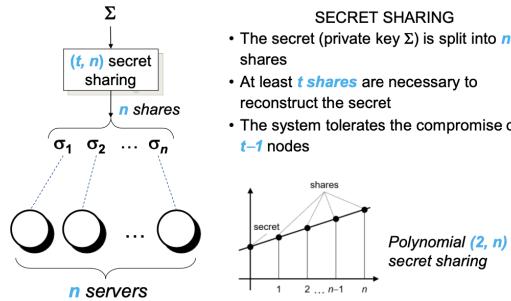


### 9.1.2 Threshold crypto

Nella crittografia a chiave pubblica, ogni utente ha una chiave pubblica e una privata. Si è inoltre visto, che posso usarla sia in cifrario che in firma digitale che in key establishment. La chiave privata in tutto ciò risulta estremamente importante, perciò deve essere protetta sia dal punto di vista della sicurezza, che della disponibilità (non deve essere persa). Una cosa importante riguarda quindi il *backup* della chiave privata.

È stato introdotto nei primi anni 90, un meccanismo di sicurezza di mantenimento della chiave privata, necessaria per la firma digitale, per evitare il backup: il **Threshold crypto**.

L'idea sulla quale si basa questo meccanismo è quella di sfruttare l'omomorfismo e suddividere la chiave privata in  $n$  componenti chiamati *shares*. Per ricomporre la chiave privata sono necessari  $t$  shares, dove  $t < n$ .



Questo meccanismo risulta potente quando unisco la proprietà dell'omomorfismo dell'RSA. Se prendo la chiave privata e faccio un *secret shared*, volendo fare una firma digitale, posso far effettuare le firme a ognuno degli  $n$  proprietari delle porzioni e poi attraverso la moltiplicazione ottenere la firma digitale.

Una tecnica per implementare ciò è attraverso l'utilizzo dei polinomi con aritmetica modulare.

## 9.2 Verifica di un certificato

Bob può verificare l'autenticità della chiave pubblica di Alice, dal suo certificato, in questo modo:

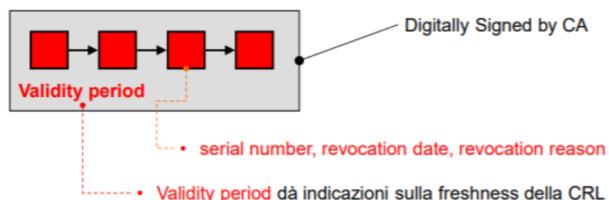
1. Si procura la chiave pubblica della Certification Authority
2. Verifica la validità della chiave pubblica
3. Verifica la firma del certificato di Alice usando la chiave pubblica della Certification Authority
4. Verifica che il certificato di Alice sia valido (cioè che stia nel periodo di validità)
5. Verifica che il certificato non sia stato revocato

Se tutte le verifiche sono positive, allora Bob può considerare la chiave pubblica di Alice valida.

## 9.3 Certificate revocation

Tutti i certificati revocati sono posti in una lista chiamata *Certificate Revocation List* (CRL), tutti i campi sono firmati dalla Certification Authority:

- serial number
- data di revoca
- motivo di revoca



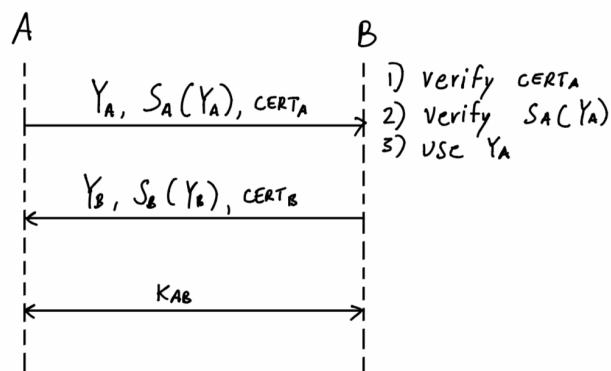
## 9.4 Campi del certificato

La struttura del certificato è standardizzata sotto il nome di **X509**.

È prevista la presenza di una serie di campi: la versione, il numero seriale, il nome dell'issuer, la validità del certificato, la chiave pubblica e così via. Ogni certificato indica un *Nome Distintivo (Distinguished Name)*, il quale è un identificativo unico relativamente all'entità da certificare: questo nome è composto da una serie di campi quali, ad esempio: Country (CO), Organization (O), Organizational Unit (OU), Common Name (CN).

Nei certificati è inserito anche l'indirizzo della CRL. Viene descritto anche l'indirizzo del *Certificate Practice Statement (CPS)*, nella quale l'ente che certifica definisce tutte le politiche e regole per verificare l'identità dei soggetti certificati.

## 9.5 Diffie-Hellman MITM solution



## 9.6 CA delegation

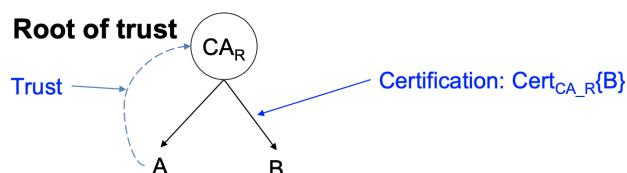
Se Bob vuole una firma certificata, deve avere un certificato come questo:

$$CERT_B = S_{CA}(Bob, pubkey_B, \dots, "CA : yes")$$

## 9.7 Trust model

### 9.7.1 Modello centralizzato

Il modello nella CA è tipicamente questo:

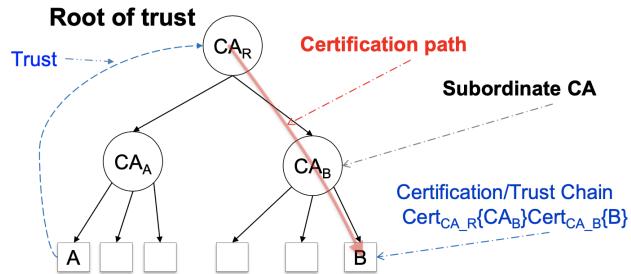


Alice, per verificarlo, ha necessità della chiave pubblica della CA. Alice quando si iscrive alla CA, potrebbe recarsi direttamente nella società della CA o ottenerla in altri modi.

Un utente, quindi, fidandosi della CA, si fida anche di Bob, che è stato già accettato dalla CA. Quindi, la chiave pubblica che ricevo relativa a Bob, posso crederla come effettivamente associata a lui.

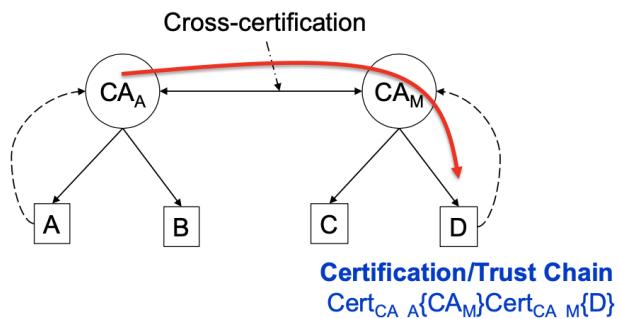
Se il sistema CA, però, è lontano, tutti gli utenti non possono recarsi direttamente sul posto. Per questo è stato introdotto un sistema gerarchico distribuito sul territorio. Il meccanismo è lo stesso, in quanto gli utenti devono avere fiducia della radice CA, che a sua volta delega delle CA.

In questo caso, quindi, il certificato di B è rilasciato da  $CA_B$  che a sua volta è verificata dalla radice:



### 9.7.2 Modello enterprise - cross-certification

È un modello usato in ambito aziendale: si basa su un meccanismo di mutua certificazione. Si hanno due CA indipendenti, non c'è più una radice.



Può essere usato, ad esempio, nel caso di un'azienda produttrice di software deve rilasciare un programma per una piattaforma di un'altra azienda produttrice: in questo caso l'applicazione viene certificata dalla prima azienda e poi dalla seconda. Esiste anche il modello *Hub-and-Spoke*, nella quale c'è una CA che si occupa di verificare unità certificate da altre CA.

### 9.7.3 Modello browser web

Ogni browser installa, localmente, una lista di certificati di CA, alle quali sarà data piena fiducia. È possibile ancora avere CA subordinate. Questo modello introduce dei problemi per via delle politiche che portano alla decisione di integrare nuove CA nella lista (molti produttori di browser chiedono un semplice compenso). Inoltre ci sono stati vari casi di CA root o CA subordinate compromesse, le quali hanno permesso di realizzare attacchi di tipo MIM (in questo caso chiamati anche attacchi di Key-bridging o Crypt-Recrypt). Che tipo di contromisure si possono prendere per evitare questi attacchi?

1. Avere una lista di *Known-good Certification Authority*, oltre alle *presumed-good*: se un sito web è certificato con una *presumed-good* quando già lo è per una *known-good*, allora si avverte l'entità di quest'ultimo certificato (Chrome lo fa già per Google).
2. Avere una lista pubblica di certificati rilasciati dalla CA (ma questo spesso non è possibile).

3. Avere più certificati da un set di CA fidati.
4. se usassimo un meccanismo di *DNS-based Authentication*, potremmo esporre la chiave pubblica tramite questo meccanismo (ciò e nella risposta del DNS)

Da un pò di tempo sono stati introdotti i certificati di tipo *Extended Validation*, per i quali vengono effettuati dei controlli più forti e concreti sulle identità dei soggetti. In tutto ciò esiste un problema di fondo: le CA non sono spinte a migliorare e rendere più sicuro questo modello perché, alla fine, le vittime degli attacchi restano sempre e solo gli utenti.

#### 9.7.4 Modello PGP

Modello non più gerarchico, usato per firmare le mail. È un modello di trust completamente decentralizzato.

Per esempio, supponendo che Alice voglia collegarsi con Bob e volesse inviargli un messaggio confidenziale. Alice non conosce Bob e potrebbe chiedere ai suoi amici se lo conoscono e se gli rilasciano un certificato per validarlo. Alice raccoglie un certo numero di certificati e poi è lei che decide quanta fiducia dare a ciascuno di questi certificati. Alice può quindi effettuare delle policy: se nessuno ha trust massimo, ma almeno 2 trust marginale, allora mi fido di Bob.

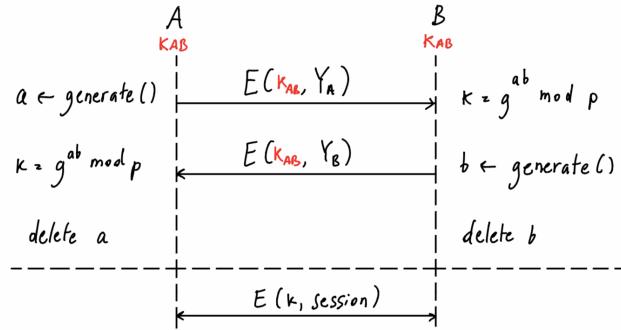
Questo PGP è software libero ed è valido per situazioni personali. La revoca è molto complicata, se so che Bob non è più autentico allora dovrei prendermi la briga di revocare tutti i certificati sulla rete.

# 10 Perfect Forward Secrecy (PFS)

DEF: la divulgazione di materiale per chiavi segrete a lungo termine non compromette la segretezza delle chiavi scambiate nelle esecuzioni precedenti.

## 10.1 Pre-Shared Key Ephemeral DH

La chiave pre-condivisa è utilizzata per l'autenticazione. Le chiavi  $a$  e  $b$  sono effimere, perché sono utilizzate una volta a sessione o messaggio.

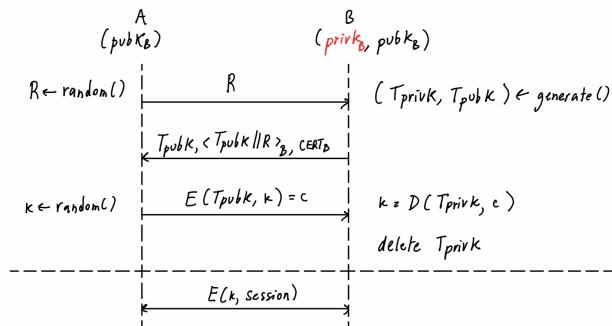


L'idea è quella di usare Diffie Hellman per scambiare una chiave temporanea, la quale viene cifrata per motivi di autenticazione (evitare attacchi man-in-the-middle).

Supponendo che  $K_{AB}$  venga rubata dall'avversario, il quale ha anche registrato i messaggi scambiati nelle sessioni precedenti: quello che può fare è decifrare i primi due messaggi, dai quali ottiene  $y_a = g^a \cdot \text{mod}(p)$  e  $y_b = g^b \cdot \text{mod}(p)$ . A questo punto deve risalire ad  $a$  e  $b$  ma, come abbiamo visto per Diffie-Hellman, è necessaria un'operazione di *discrete logarithm*, la quale è computazionalmente difficile. Si è anche protetti da attacchi di reply, per via della generazione biparte delle chiavi. Le sessioni future possono essere compromesse solo per via di attacchi MIM.

## 10.2 Ephemeral RSA

La chiave privata è utilizzata per l'autenticazione. B genera  $T_{pubk}$  e  $T_{privk}$  sul momento per scambiare la chiave di sessione.

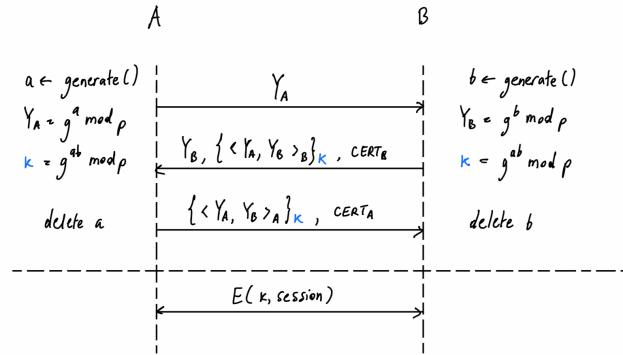


Notare che A, alla fine della fase di scambio delle chiavi, non ha nessuna garanzia che B abbia la chiave nelle sue mani.

La firma serve per provare ad Alice che la chiave pubblica proviene da Bob, mentre  $R$  serve per indicare che non è una vecchia connessione, ma nuova.

### 10.3 Station-To-Station (STS) protocol

Garantisce Perfect Forward Secrecy e autenticazione diretta (o conferma della chiave).



La coppia  $< Y_A, Y_B >$  è firmata in modo da evitare attacchi MITM. Il criptaggio attraverso  $k$  è fatto soltanto per garantire *direct authentication*.

# 11 BAN logic

È una logica che permette di analizzare i protocolli. La logica non può dimostrare che un protocollo sia sbagliato, tuttavia, se non riesci a dimostrare un protocollo corretto, allora quel protocollo va considerato con sospetto. È una logica che cerca di formalizzare un protocollo crittografico cercando di astrarre il più possibile dalla tecnologia.

## 11.1 Formalismi

- $P \equiv Q$ : P crede a X. P si comporta come se X sia vero.
- $P \triangleleft X$ : P vede X.
- $P \sim X$ : P una volta ha detto X.
- $P \Rightarrow X$ : P controlla X.
- $\#(X)$ : X è fresco.
- $P \xleftarrow{K} Q$ : K è la chiave pubblica condivisa tra P e Q.
- $P \xrightarrow{K} Q$ : K è un segreto condiviso tra P e Q.
- $\xrightarrow{K} P$ : K è la chiave pubblica di P.
- $\langle X \rangle_Y$ : X è una combinazione con Y.
- $\{X\}_K$ : X è stato criptato con K.

## 11.2 Concetti preliminari

La BAN logic considera 2 epoche: il presente ed il passato. Il presente inizia con l'inizio del protocollo.

Le credenze apprese nel presente sono stabili per tutta la durata del protocollo. Mentre, credenze del passato potrebbero non essere vere nel presente.

*Se P dice X, allora P crede a X*

## 11.3 Postulate 1: message meaning rule

$$\frac{P \equiv Q \xleftarrow{k} P, P \triangleleft \{X\}_k}{P \equiv Q \sim X}$$

Se K è una chiave condivisa tra P e Q, un P vede un messaggio criptato da K contenente X (e P non ha inviato quel messaggio), allora P crede che X è stato inviato da Q.

$$\frac{P \equiv Q \xrightarrow{k} P, P \triangleleft \langle X \rangle_k}{P \equiv Q \sim X}$$

Se K è la chiave pubblica di Q e P vede un messaggio firmato con  $K^{-1}$  contenente X, allora P crede che X sia stato inviato da Q.

$$\frac{P \equiv Q \xrightarrow{k} P, P \triangleleft \{X\}_{k^{-1}}}{P \equiv Q \sim X}$$

Se Y un segreto condiviso tra P e Q e P vede un messaggio dove Y è combinato con X (e P non ha inviato il messaggio), allora P crede che X è stato inviato da Q.

## 11.4 Postulate 2: Nonce verification rule

$$\frac{P \equiv \#(X), P \equiv Q \sim X}{P \equiv Q \equiv X}$$

Se P crede che Q abbia detto X e P crede che X sia fresco, allora P crede Q e crede a X (in questa esecuzione del protocollo).

Se P crede che X sia stato inviato da Q, e P crede che X sia fresco, allora P crede che Q abbia inviato X in questa istanza di esecuzione del protocollo.

## 11.5 Postulate 3: Jurisdiction rule

$$\frac{P \equiv Q \equiv X, P \equiv Q \Rightarrow X}{P \equiv X}$$

Se P crede Q, che crede a X e P crede che Q sia un'autorità su X, allora P crede anche a X.

Se P crede a Q che dice X e P ha fiducia in Q riguardo X, allora P crede anche a X.

## 11.6 Objectives

### Key authentication

$$P \equiv P \xleftrightarrow{k_{PQ}} Q$$

$$Q \equiv P \xleftrightarrow{k_{PQ}} Q$$

### Key confirmation

$$P \equiv Q \equiv P \xleftrightarrow{k_{PQ}} Q$$

$$Q \equiv P \equiv P \xleftrightarrow{k_{PQ}} Q$$

## 12 Kerberos

È un sistema con TTP, basato sul protocollo Needham-Schroeder, dove uno dei partecipanti, tipicamente utente, collegato alla workstation e l'altro è un server. Si vuole inoltre garantire: *Sicurezza, Availability, Transparency, Scalability*.

Ci sono due segreti a lungo termine condivisi a priori, dove la chiave dell'utente è generata dalla password tramite qualche funzione.

- $k_B$  è parte della configurazione del sistema
- $k_A = f(password)$
- Requirement: singolo sign-on

### 12.1 Protocol

- $L$ : intervallo di validità di  $k_{AB}$
- $N_A$ : nonce
- $WS$ : workstation IP address
- *subkey* utilizzato per criptare i dati

$$M1 : A \rightarrow AS : A, B, t, L, N_A, WS$$

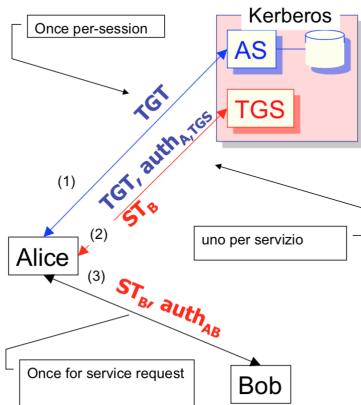
$$M2 : AS \rightarrow A : \{A, B, t, L, k_{AB}, WS\}_{k_B}, \{B, t, L, N_A, k_{AB}\}_{k_A}$$

$$M3 : A \rightarrow B : \{A, B, t, L, k_{AB}, WS\}_{k_B}, \{A, t_A, subkey\}_{k_{AB}}$$

$$M4 : B \rightarrow A : \{t_A, subkey\}_{k_{AB}}$$

L'autenticatore, criptato con  $k_{AB}$  prova che anche A ha la chiave  $k_{AB}$ . Contiene un *timestamp* generato tutte le volte che A vuole comunicare con B. Kerberos richiede clocks sincronizzati. Dato che non è possibile, il tempo  $t_B$  in cui B riceverà l'autenticatore sarà più grande di  $t_A$ . Il *lifetime dell'autenticatore*  $\lambda = |t_B - t_A|$  dovrebbe essere nell'ordine della qualità della sincronizzazione e in questo intervallo l'autenticatore può essere sostituito.

### 12.2 Ticket Granting Server (TGS)



Al login, A riceve il *Ticket Granting Ticket* (TGT) da AS in modo da poter parlare con il TGS, a cui A chiede il *Service Ticket* (ST), necessario per parlare con il sistema dei server.

## 12.3 Delegation problem



Assumendo che un mail server MS, debba salvare le mail di A nel file server FS: MS deve operare come se fosse A.

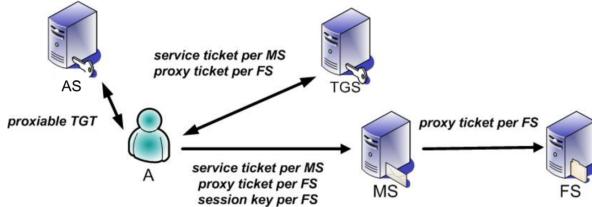
Il mail server dovrà interagire con il file system e depositare l'email da qualche parte. Per fare ciò, il mail server deve avere i diritti adeguati e tipicamente i diritti di root. Storicamente il mail server è stato uno dei principali obiettivi di attacchi essendo un servizio complicato.

L'idea che cerca di introdurre Kerberos è quella di permettere al mail server di scrivere nel file system ma facendoglielo fare con il minimo di privilegi possibili (*minimum privilege principle*). Questo è un principio di cui dobbiamo tener sempre conto: un soggetto possa fare il proprio compito con il minimo privilegio possibile.

Kerberos ha affrontato il problema tramite 2 meccanismi:

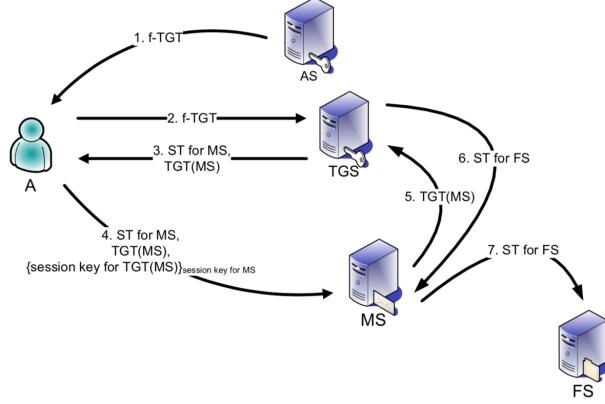
- Proxiable ticket
- Forwardable TGT

### 12.3.1 Proxy tickets



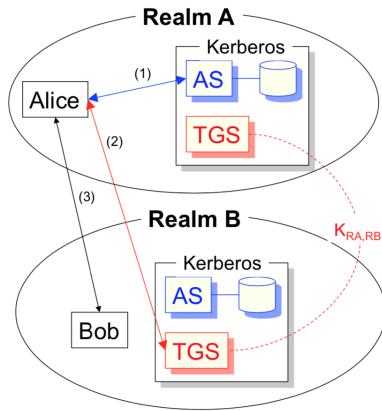
Quello che vuole fare Alice è di delegare il mail server a scrivere nel file system. Per fare ciò Alice chiede il TGT, ma non quello normale. Questo ticket viene chiamato Proxiable Ticket. Proxiable TGT consente ad A di chiedere al TGS un ticket di servizio  $ST_{MS}$  e un ticket proxiable  $PT_{FS}$ . MS potrebbe fare danno soltanto a FS per i limiti specificati nel  $PT$ . Tuttavia, A deve conoscere il numero di ticket necessari da MS per completare il servizio correttamente.

### 12.3.2 Forwardable TGT



Forwardable TGT consente ad A di richiedere al TGS un TGT per MS. MS utilizzerà questo THT per chiedere a TGS ogni *ST* di cui ha bisogno. A non deve conoscere in anticipo il numero di ticket per MS, ma se MS si compromette può abusarne

### 12.3.3 Referral TGT



Referral TGT consente ad A di richiedere ad un TGS di un altro *regno* un *ST* per interagire con un server di quel regno

# A Buffer overflow

The data that overflows the buffer is written in the memory space that happens to be contiguous to the overflowed buffer, containing other data (variables, etc.). Therefore, other data is over-written.

The C and C++ languages are particularly susceptible to bugs leading to buffer overflow vulnerabilities. This is because they do not enforce implicit bounds checking on arrays, and they provide standard library functions (e.g., `gets()`) which do not enforce bounds checking. Moreover, they implement strings as null-terminated arrays of characters. This makes programming in C/C++ more error-prone than in other languages, for example the Pascal language which implements strings as couples `<length, characters>`.

A non-static variable in C/C++ can be allocated either dynamically (with `malloc()` or `new/new[]` operators) or automatically (by declaring it as a local variable in a function). The *heap* is a memory space that contains all the dynamic variables, while the *stack* is a memory space that contains all the automatic variables and the return addresses of the subroutines. A buffer overflow in the heap is called *heap overflow*, while a buffer overflow in the stack is called *stack overflow*. Both heap and stack overflows can read/change the value of other variables. Stack overflow is generally more dangerous, because it can directly change the execution sequence by changing the return address of the subroutines.

Different types of attack can be performed exploiting buffer overflow:

- *Return injection*: the attacker makes the overflowed part corresponding to a valid address such that when the function returns, the execution flow will jump to the wanted instruction;
- *Code injection*: the attacker makes the return address pointing to the first memory location of the injected malware. To mount a code injection, the attacker must know the exact address of the stack, which is not always predictable. To overcome this, the attacker can prepend the malware with a large number of NOP instructions (*NOP slide*). The overwritten return address needs only to jump in the middle of the NOP slide, so it can be approximated. Then, the control flow will drop down to the malware code.

## A.1 Countermeasures

The most common stack overflow protections are *Data Execution Prevention* (DEP), *Address Space Layout Randomization* (ASLR), and *Stack Canaries*. None of these protections is definitive, because they do not make the attack impossible, but only more technically challenging.

### A.1.1 Data Execution Prevention

DEP works by marking each part of the memory as executable or non-executable. The stack is marked as non-executable, so when the victim process tries to execute the malware in the stack, a fault is raised and the process abnormally terminates. Hardware-based DEP has a negligible impact on performance. Note that DEP prevents code injection, but it does not prevent return injection.

DEP can be cheated with *Return-Oriented Programming* (ROP) techniques. In ROP, malware is not injected but «built» with a sequence of victim code fragments, each of which ends with a return instruction (gadgets). The attacker injects the addresses of such gadgets in the stack. At the end of every gadget, the return instruction will make the instruction pointer jump to the next gadget. All the gadgets are located in executable memory, thus DEP is bypassed and the victim process does not abnormally terminate.

Depending on the victim code, it could not be possible to build complex malware (e.g., a shellcode) with gadgets. A simpler technique is to use the gadgets to build an API call that disables DEP on the victim process, which is much simpler. After that, the attacker can mount a «classic» code injection, by placing after all the gadgets’ addresses an additional address pointing to a location of the stack containing a complex malware.

### A.1.2 Address Space Layout Randomization

In old operating systems, when a program was executed, it was always loaded in memory starting from address 0. This makes ROP easy, since the adversary knows deterministically all the addresses of the victim code. With *Address Space Layout Randomization* (ASLR), the operating system decides randomly where to load the program to be executed. All the absolute addresses of the program code are relocated by the same random offset. Since the attacker does not know the relocation address, he cannot perform ROP because he cannot build the sequence of gadget addresses.

ASLR is an additional source of uncertainty for the attacker. However, it does not make the attack impossible, rather probabilistic. Indeed, the possible relocation addresses are few in the typical operating system. For example, Windows has only 254 possible random relocation addresses. The attacker could simply try several times to attack the process, until he guesses the correct relocation address. Moreover, the attacker can leverage *pointer leaks*, that are program bugs that expose the value of pointers. If such pointers point to the code (e.g., a function pointer or the return address of a function call), they reveal the relocation address. Finally, the attacker could do ROP with gadgets from non-randomized memory, for example memory that contains the code of libraries shared between different processes.

### A.1.3 Stack canaries

In the stack canaries technique, the prologue of every function pushes a value (canary) in the stack between the local variables and the return address. If an attacker wants to overwrite the return address with a buffer overflow, then he must corrupt also the canary. The function epilogue checks if the canary still has the correct value, and it raises an error if it has not.

The canary value must be unpredictable by the attacker. Usually, it is chosen at random at program execution, and stored in the thread-local storage. Stack canaries are a compiler-based protection technique.

Stack canaries are currently the most effective defense against stack overflow, because they are hard to bypass or to guess. Stack canaries are typically 4-byte long, so the attacker has a chance over  $2^{32}$  to guess it. Stack canaries slightly slow down the program performance since they introduce some operations at the prologue and at the epilogue of every function. Some compilers optimize the program by avoiding stack canaries in those functions that they consider «safe» from stack overflow vulnerabilities. For example, a compiler could consider safe a function which does not declare local arrays, or declares small local arrays. Although stack canaries are quite effective and efficient, note however that they can defend only against arc injection and code injection. They cannot defend against those types of stack overflows that do not overwrite the return address. For example, an attacker can use a stack overflow to overwrite variables that are stored in locations contiguous to the overflowed buffer. In this way, the attacker can cause an unexpected behavior of the program without corrupting the return address and the canary. In addition, it is still possible to use a stack overflow to simply produce an abnormal program termination, thus causing a denial of service. Moreover, stack canaries cannot prevent *buffer over-reads*, that is overrunning a buffer’s boundary in a read operation. A

buffer over-read can lead to information leaks. Notably, a buffer over-read in the stack can also leak the value of the stack canary, which could then be used to mount successive stack overflow attacks.

## B Secure coding

*Secure coding* studies those programming errors which have caused the most common, dangerous, and disruptive software vulnerabilities in the past, the remediation best practices and the *risk assessment*, i.e., the probability that these errors are exploitable, the possible impact, and the remediation cost. Secure coding aims at protecting customers from money loss due to security incidents, and limiting the patch releases of software.

Secure coding is strongly language-dependent. As a general rule, the lower the programming language level is, the more error prone, and thus the more susceptible to vulnerabilities. C and C++ languages are particularly error-prone, because they are intended to be lightweight and to produce a small codeprint. The programmer, especially if coming from higher-level languages like Java, may assume that the C language implicitly perform checks when it does not. Moreover, C and C++ follow the *power-to-the-programmer* philosophy, that they do not prevent the programmer from doing what needs to be done. So they always consider the programmer to be fully aware of the consequences of the code he is writing. Despite their poor security characteristics, C and C++ are widely used still today, in particular when performances are a requirement (e.g., constrained/embedded devices or high-load servers) and when legacy code must be maintained.

### B.1 Undefined behavior

An *undefined behavior* is a behavior on which the C/C++ standard poses no requirements, for example in case of out-of-bound buffer access, null pointer dereferencing, or signed integer overflow. An *unspecified behavior* is a behavior on which the C/C++ standard gives two or more possibilities, for example the order of argument evaluation in a function call. An *unexpected behavior* is a well-defined behavior, yet unexpected by the programmer, due to incorrect programming assumptions. The majority of software vulnerabilities is based on undefined behaviors. The attacker tries to induce the program to perform undefined behaviors by means of special crafted inputs. Then, the attacker tries to damage somehow the system. Undefined behaviors must be avoided. Unspecified behavior must be known. Unexpected behaviors must be expected.

### B.2 Taint analysis

Data that comes from a source external to the program (network, user input, file, command line arguments, environment variables, other software, etc.) and that has not been sanitized yet is called *tainted data*. The result of an operation over tainted data is tainted data too. An operand or a function argument whose domain (i.e., the set of valid values) is a subset of the domain of its type (i.e., the set of all the possible values) is called *restricted sink*. Undefined, unspecified and unexpected behaviors happen when tainted data is given as input to a restricted sink.

### B.3 Sanitization

*Sanitization* is an operation that removes the taint from a data. Sanitization can be done by *replacement* or *termination*. Replacement replaces out-of-domain values with in-domain values, while termination terminates the execution path of the entire program or the current operation.

## B.4 Strings

In C, there is not a native string type. Strings are implemented by arrays of characters, with a special character (*string terminator*) to indicate the end of the string. The array capacity must always be greater or equal than the string length + 1, to accomodate the terminator at the end. No terminator character is allowed inside the string. The absence of a native string type and the intrinsic insecurity of the C string library functions can lead to many vulnerabilities, some of which are hard to correct. The C++ standard introduces the `std::string` type which is far less error-prone.

Vulnerable	C countermeasure	C++ countermeasure
<code>gets(buf)</code>	<code>fgets(buf, size)/gets_s(buf, size)</code>	<code>getline(cin, buf)</code>
<code>scanf(buf)</code>	<code>scanf("%1023s", buf)</code>	<code>cin &gt;&gt; buf</code>
<code>sprintf(dest, "%s", src)</code>	<code>sprintf(dest, "%1023s", src)</code>	<code>dest=(string)src</code>
<code>strcpy(dest, src)</code>	<code>strcpy(dest, src, size) + terminator</code>	-

## B.5 Pointer subterfuge

The attacker is able to perform an arc injection by overwriting the `func_ptr` variable to point to a different function than `good_func()`. This attack is an example of pointer subterfuge. A pointer subterfuge is the act of maliciously change a pointer to an object or a function before the victim program uses it.

A C/C++ program implicitly uses many function pointers, some of which are susceptible to pointer subterfuge. An example is the *Global Offset Table* (GOT) in Linux, which contains pointers to global objects and functions whose addresses are unknown at compile time. The GOT contains the addresses of dynamic library functions like `strcpy()`, `printf()`, `exit()`, etc. The address of the GOT is fixed at compile time and it is not relocated at execution time. Moreover, the GOT resides in a writable part of memory.

## B.6 Dynamic memory management

In C, the `malloc()` function must always be checked for allocation failure. C++ is safer than C regarding memory allocation/deallocation. Indeed, the `new` and `new[]` operators (by default) raise an exception if the allocation fails, so it is harder to forget to handle such error than using `malloc()`.

However, STL containers like `std::vector` and STL iterators are not much safer than “classic” C arrays, since they follow the same power-to-the-programmer philosophy, and they are efficiency-oriented. Indeed, using `std::vector` with functions that do not check for bounds (i.e. `std::copy()`) does not prevent from out-of-bound accesses.

A possible solution is to use `vector.at()` for accessing a `vector` element.

## B.7 Vector iterators

C++ iterators are very powerful and they permit the programmer to do things impossible with higher-language iterators, for example with Java iterators. One of these things is the on-the-fly modification of containers.

```
void func(std::vector<int>& c){
    for(auto i = c.begin(); i != c.end(); i++){
        c.insert(i, val); // -> i = c.insert(i, val)
        i++;
    }
}
```

```
    }  
}
```

The `insert()` method could cause the reallocation of the `std::vector` internal buffer in case its capacity was not enough to accomodate the new element. In such a case, all the old iterators will be invalidated. The subsequent dereferencing of an invalid iterator constitutes an undefined behavior (typically, a read or write in unallocated memory).

In general, all the methods that invalidate iterators (e.g., `insert()`, `erase()`, `push_back()`, `pop_back()`, etc.) must be used carefully inside iterator-based cycles. The STL standard specifies which method of which container may invalidate which iterator. The standard solution is to use the return value of the `insert()` method, which always returns a valid iterator pointing to the newly inserted element. The `erase()` method return a valid iterator pointing to the element successive to the erased one.

## B.8 OS command injection

The program uses `system()` function for executing shell commands, for instance sending email to an address specified by the user. An attacker could pass an address concatenated to a malicious shell command. The `system()` function is an interface to a command interpreter (the shell), which could receive as input special characters resulting in the execution of commands unexpected by the programmer. In general, strings passed to command interpreters like the shell, external programs, SQL interpreters, XML/Xpath interpreters, etc. should be sanitized before.

Sanitization can be performed with a character *white list* (list of only-allowed characters) or character *black list* (list of disallowed characters). Whitelisting is always recommended, because it is easier to identify “safe” characters than identify “unsafe” ones. If a white list is incomplete, this only constitutes a missing functionality (and not a vulnerability), which can be easily detected. Conversely, if a black list is incomplete, this constitutes a vulnerability, which can be hardly detected.

The call of `system()` is problematic and the CERT secure coding standard always discourages it. Process forking and calling `execve()` or `exec1()` are recommended instead. In addition to unsanitized tainted strings, `system()` is dangerous if:

- A command is specified without a path, and the PATH environment variable is changeable by an attacker. (The PATH variable specifies the default paths where the command processor finds executables to launch);
- A command is specified with a relative path and the current directory is changeable by an attacker;
- The specified executable program can be replaced (spoofed) by an attacker.

## B.9 TOCTOU race condition

A *time-of-check/time-of-use* (TOCTOU) race condition occurs when two concurrent processes operates on a shared resource (e.g., a file), and one process first accesses the resource to check some attribute, and then accesses the resource to use it. The vulnerability comes from the fact that the resource may change from the time it is checked (time of check) to the time it is used (time of use). In other words, the check and the use are not a single atomic operation. The shared resource can be a hardware device, a file, or even a variable in memory in case of a multi-threaded program. This program writes a given file only after having checked that the file does not exist, to avoid overwriting it. However, the time of check is different to the time of use, so a TOCTOU race condition is possible. Assume that an attacker wants to destroy the content of a file over which she has

not write permission. Assume further that the TOCTOU-vulnerable program runs with higher privileges. To induce the program to overwrite the file content, the attacker can create a symbolic link to the file just after the time of check but just before the time of use.

To avoid the TOCTOU race in this case, it is sufficient to use the "x" flag when opening the file (since C11), which forces the `fopen()` function to fail in case the file already exists.

TOCTOU race conditions are typically easy to identify, but not always easy to correct in general. Sometimes it is better to mitigate the vulnerability in other ways, rather than avoiding the TOCTOU race itself. For example, we can simply run the program as a non-privileged user, or we can read/write files only in «secure directories», i.e., directories in which only the user (and the administrator) can create, rename, delete, or manipulate files.

## B.10 Integers

The C standard imposes the `unsigned` integers to be represented with their binary representation, while allows the signed integers to be represented with three techniques: sign-and-magnitude, one's complement, two's complement. The most used representation in desktop systems is however two's complement.

The majority of platforms use two's complement representation, and silently wrap in case of signed integer overflow. Many programmers rely on this usual behavior as if it were the standard one, but it is not. Notably, compilers leverage the assumption that no integer overflow happens in order to implement code optimization. So the actual behavior could vary on the compiler, the compiler version, and the single piece of code. Despite the fact that the integers are used primarily for sizing and indexing arrays, for which the presence of the sign is meaningless, the signed integers are traditionally the most used integer types in C and in many modern programming languages.

### B.10.1 Unsigned sanitization

A general method to sanitize integers before potentially overflowing operations is the following:

1. Identify how an arithmetic operation could overflow, and write the error condition «as is», i.e., as if no overflow would happen within the error condition itself;
2. Algebraically change the condition to avoid overflow inside the error condition itself;
3. Possibly avoid the additional overflows in the error condition just obtained.

Sum	<code>a &gt; UINT_MAX - b</code>
Subtraction	<code>a &lt; b</code>
Increment	<code>a == UINT_MAX</code>
Decrement	<code>a == 0</code>
Multiplication	<code>b != 0 &amp;&amp; a &gt; UINT_MAX/b</code>
Division and modulo	<code>b == 0</code>

### B.10.2 Signed sanitization

Sanitizing operations with signed integers is in general more difficult than unsigned integers. This is because signed operations typically can overflow in both directions: beyond `INT_MAX` and below `INT_MIN`. For example, a signed sum can overflow beyond `INT_MAX` if the operands are both positive, or below `INT_MIN` if they are both negative.

The simplest countermeasure is to sanitize the signed integers to be non-negative, thus making them unsigned. Then, apply the sanitization rules for the unsigned ints seen before. However, sanitizing away the negative values is not always possible, as they may be valid values of the program. Another solution is to tell the compiler to raise an exception on integer overflows, so that the program crashes instead of going in undefined behavior. GNU GCC compiler allows to do that with the `-ftrapv` flag. The `-ftrap` flag forces the integer overflow to raise an exception, which by default results in a program termination. This makes a program more secure, but also less efficient because it causes implicit function calls in every signed operation and prevents some hardware optimizations. Moreover, it does not cover all the signed operations, so it is a partial solution. Another solution is to apply the general sanitization method twice, once for each overflow direction.

Sum	<code>b &gt;= 0 &amp;&amp; a &gt; INT_MAX - b</code> <code>b &lt;= 0 &amp;&amp; a &lt; INT_MIN - b</code>
Subtraction	<code>b &lt;= 0 &amp;&amp; a &gt; INT_MAX + b</code> <code>b &gt;= 0 &amp;&amp; a &lt; INT_MIN + b</code>
Increment	<code>a == INT_MAX</code>
Decrement	<code>a == INT_MIN</code>
Multiplication	<code>(b &gt; 0 &amp;&amp; a &gt; INT_MAX/b)    (b &lt; 0 &amp;&amp; a &lt; INT_MAX/b)</code> <code>(b &gt; 0 &amp;&amp; a &lt; INT_MIN/b)    (b &lt; -1 &amp;&amp; a &gt; INT_MIN/b)</code>
Division and modulo	<code>b = 0</code> <code>b == -1 &amp;&amp; a == INT_MIN</code>

## C Network security

### C.1 Penetration test

A *penetration test* (pen test, in brief) is an authorized simulated attack to a computer system. The aim is to determine the security level of a system and report to the system's owner the possibly found vulnerabilities. A penetration test can be white-box, black-box, or gray-box. In *white-box* penetration tests, the pentester is given full knowledge of the system details, and the organization is aware that a penetration test is ongoing. White-box pen tests are generally cheaper and quicker, and they can find lots of vulnerabilities. However, they are generally less realistic. In *black-box* penetration test, the system details are hidden to the pentester, and (most of) the organization is unaware that a penetration test is ongoing. Black-box pen tests are more expensive, and they usually find only the easiest and most exploitable vulnerabilities. However, they are more realistic, and they also evaluate the reaction capabilities of the security team of the organization. *Gray-box* pen test are a mix of white-box and black-box.

### C.2 Exploitation

An *exploit* is a piece of software that takes advantage of a vulnerability to cause some damage on a computer. Such damage includes executing arbitrary code, escalating privileges, or performing a denial-of-service attack. If the exploit aim at executing code, the malware being executed is called the *payload* of the exploit. Examples of payloads are remote shells, spam-sending programs, self-replicating worms, etc. An *exploit kit* is a collection of exploits and payloads. A famous and widespread exploit kit is Metasploit framework.

### C.3 Network scanning

*Network scanning* is a collection of techniques aimed at discovering hosts and services on a network by sending packets and analyzing the responses. An example of network scanner is *nmap*. Network scanning usually involve *host discovery*, that is enumerating particular hosts on a network, for example those responding to ICMP requests or having a given port open, and *port scanning*, that is enumerating open/reachable ports on a given host.

#### C.3.1 SYN scanning

Port scanning is usually done by *SYN scanning* (aka TCP half-open scanning), which consists in initiating a TCP connection establishment with a given host/port without completing it. The network scanner sends a SYN packet to the target port and observes the response. If the target responds with a SYN-ACK packet, then the port is open to accept connections. The network scanner can possibly abort the connection by sending a RST packet. If the target responds with a RST-ACK packet, then the port is closed and it does not accept incoming connections. If the target does not respond at all, then the port could be filtered by a firewall. SYN scanning has the advantage that it consumes few resources on the target host, because no TCP port is left open.

#### C.3.2 UDP scanning

Scanning UDP ports is more difficult because the protocol does not provide acknowledgement in case of open ports like TCP does. In particular, it is difficult to distinguish the open and the filtered state since many UDP applications will simply ignore unexpected

packets. To perform UDP scanning, the network scanner sends a probe UDP packet to the target port, typically with an empty payload. If the target responds with an ICMP type 3 code 3 packet (port unreachable) or an ICMP type 3 code 13 (administratively prohibited) packet, then the UDP port is closed. If the target does not respond within a timeout, then the UDP port could be open as well as filtered by a firewall.

### C.3.3 Service scanning

*Service scanning* is a technique aimed at determining the type and the version of a server reachable at a given port. It works by sending one or more protocol-specific messages to the port, and analyzing the responses. For example, sending a GET / HTTP/1.1 probe to the port 80 could detect a web server, and the response could give indications about which server and which version of such server. Note that service scanning in some cases can also discriminate between open and filtered UDP ports.

## C.4 TCP/IP stack fingerprint

*TCP/IP stack fingerprinting* (aka OS fingerprinting) is a technique aimed at determining the type and the version of the operating system running on a given host. It works by analyzing implementation-dependent TCP/IP parameters. Some parameters of the TCP protocol are not strictly defined by the specifications, and they are left up to the implementation. Different operating systems, and different versions of the same operating system, set different default values for these parameters. By analyzing these values, the network scanner can distinguish the type and the version of the operating systems and of the TCP/IP stack implementations. A typical parameter used for fingerprinting is the initial Time-To-Live (TTL) of the IP packets, which is usually small for older operating systems and large for newer ones. This reflects the fact that the number of hops of the average TCP connection increases as the years go by, due to the increasing complexity of the Internet. Since the TTL of an IP packet gets decremented at each hop, the network scanner must know the number of hops from the target to infer the exact initial TTL. Another typical parameter is the TCP window size, which tells how much data the receiver can accept before acknowledging the sender.

## C.5 Network sniffing

*Network sniffing* (aka passive network scanning) is a technique aimed at eavesdropping and analyzing traffic in a local area network, in order to steal unencrypted data or gain information like available hosts, open ports, available services, etc. An example of network sniffer is wireshark. Network sniffing can gain the same information that network scanning can, including service type/version and operating system type/version, but in a completely stealthy way. It requires the attacker to be in the same local area network of the target hosts.

## C.6 Fuzzing

*Fuzzing* (aka fuzz testing) is a testing technique that automatically provides to a target software a long series of invalid, unexpected, or totally random inputs with the aim of finding implementation bugs and vulnerabilities. Examples of inputs are network messages, but also keyboard and mouse events or files. The attacker monitors the target software for faults. If the attacker can execute the target software locally, she can monitor also for unexpected internal states and memory leaks. Common examples of fuzzers are *American fuzzy lop* and *Peach fuzzer*.

*Generation-based* fuzzing produces (semi-)random inputs from scratch. On the contrary, *mutation-based* fuzzing produces inputs by randomly modifying some seed input configured by the tester. *Dumb* fuzzing produces completely random inputs, without knowing the input structure. On the contrary, *smart* fuzzing produces inputs basing on a formal model of the input structure.

## D Malware

In the past ('80-'90), the virus writers were principally young male programmers wanting to prove their skills to the community of “black hats”. Nowadays, malware is developed mainly for business, politic, or espionage purposes. Political activists typically write malware to attack a specific website of a company or organization. Criminals develop malware in order to show paid-per-click advertisements, or to steal and sell passwords, credit card numbers, or personal information. Moreover, they can blackmail victims by blocking their computer and demanding a ransom to unblock them, or they can simply sell malware to other parties (black market). Finally, intelligence agencies develop malware to steal strategic information from world leaders, or to perform industrial espionage or industrial sabotage.

Modern malware is hard to classify rigidly, because it usually has different features. An important classification is based on its infection paradigm (if any):

- *Virus* is a piece of code that infects a file, typically executables or any files capable of containing code (e.g., .doc, .pdf, etc.), and it self-replicates to other files. Actually, viruses are rare nowadays, because their infection is easy to detect by anti-virus software. However, the term “virus” is also used to indicate a generic self-replicating malware;
- *Worm* is a program that infects a computer (not a single file, like viruses do), and it self-replicates to other computers through a network. Worms are the most common kind of self-replicating malware nowadays;
- *Trojan horse* (or Trojan) is a malicious program that appears to be legitimate, thus misleading the user about its true intent. It usually does not self-replicate, and it infects the victims by social engineering (e.g., an hacker that sends an email with malicious attachment);
- *Spyware* collects information about the user;
- *Keylogger* records the key strokes to steal passwords;
- *Adware* shows advertising to the user, and it is not necessarily malware. It is considered malware if the user did not give his consent to receive such advertising;
- *Rootkit* grants privileged access to the host system;
- *Dialer* performs phone calls to premium numbers;
- *Botnet* is a network of infected computers receiving orders from a Command and Control point (C&C). It is typically used to launch Distributed Denial of Service (DDoS) attacks against a specific target;
- *Ransomware* typically encrypts user information and then asks the user a ransom in bitcoins or other cryptocurrencies for the decryption key;
- *Scareware* displays messages to cause panic and anxiety in order to lead users into buying other software. Most scareware presents itself as security software, and it deceives users into believing their computer is infected by a virus;
- *Cryptominer* uses victim’s computer to mine cryptocurrencies.

### D.1 Common features

The only feature common to all malware is *infection*, that is, the capability of executing the first time on a host against the will of the user. To do this, malware can leverage poor precautions by the user and/or security vulnerability of some software components.

An infection technique is also called *infection vector*. Malware can be *persistent*, which means that it keeps the host infected after a reboot. Malware can *self-replicate*, that is, it can automatically infect other hosts with copies of itself. Malware can adopt *concealment* techniques, in order to avoid being detected by the user or by anti-virus software. Malware can perform *privilege escalation*, that is gain higher privileges on the host machine in order to bypass access control mechanisms. Finally, malware can *damage* the host and/or other victims.

### D.1.1 Infection and self-replication

There are countless infection and self-replication mechanisms. The most common ones involve social engineering, for example sending emails with Trojan attachments, or deceiving users into downloading infected files from malicious (or poorly controlled) websites. The file names with a double extension (e.g., “.jpg.exe”) leverage the default behavior of Windows systems which hide the real extension “.exe” to the user. The user thus sees the file as having an innocuous file extension (e.g., “.jpg”). Another common infection method is to bruteforce authentication with a database of default or commonly used username/password couples (e.g., admin/admin).

A lot of malware also infects mounted removable drives (e.g., USB memory sticks) and copies itself on them in autorun configuration. Another common method is sending malformed messages to vulnerable processes, in order to leverage bugs (e.g., buffer overflows) that allow arbitrary code execution on the victim machine.

### D.1.2 Persistency

*Persistency* is the ability of self-execute after reboots. Viruses typically prepend themselves at the beginning of legitimate executable files. Worms can leverage many persistency techniques, for example register themselves as start-up programs. In Windows systems this can be done by writing certain keys in the system registry.

Another common persistency technique is *DLL hijacking*, in which malware replaces a dynamic library (DLL in Windows, SO in Unix) with malware code. Every time a legitimate process calls some API function of the library, malware is executed. The malware code, after performing some malicious action, calls in turn the legitimate API function of the legitimate dynamic library, in order to conceal its presence.

Another method is to replace a legitimate interrupt handler with malware code (*handler hijacking*). Every time a legitimate process triggers such an interrupt, malware is executed.

### D.1.3 Concealment

*Concealment* means hiding malware’s presence to the user and to the anti-virus software. In order to hide its presence to the user, malware usually chooses non-explicative or random process names, which go unnoticed among other legitimate process with non-explicative names. The same is done for malware installation files, if any.

Another concealment technique is *process injection*. In this case, the malware injects its code inside another legitimate process and executes it. This is allowed in modern operating systems (Windows, Unix, etc.) mainly for debugging purposes. In this way, malware conceals from detection and accesses the resources of the injected process. In Windows systems, process injection can be done by means of the `CreateRemoteThread()` API function, which creates a thread that runs in the address space of another process. This is also called *DLL injection*, because it requires malware to reside in a .DLL file.

More complex techniques are possible:

1. An application (or the user) wants to view the list of the active processes;
2. In the process list, among legitimate processes, there is the worm's process. The malware wants to hide it;
3. The malware hijacks the library call that retrieves the active process list. It intercepts the system call and cancels its name from the actual list;
4. The application (or the user) views the “clean” list.

#### D.1.4 Damage

Malware can damage the host system or perform unwanted actions. The type and nature of these actions reflects the original intent of the malware writers. If the intent is to prove hacking skills, malware only displays funny messages or does some random damage, for example it deletes random documents or images or audio/video files. If the intent is to do activism, malware can remain silent for long times and then, upon receiving a specific message from a command and control point, it can send lots of requests to some target web site to realize a distributed denial of service (DDoS) attack. This behavior is typical of botnets.

If the intent is to earn money, malware can display unwanted paid-to-click advertisement (adware), or use victim's processing power to mine cryptocurrency (cryptominer), or leak personal data which can be sold afterwards (spyware). Malware can encrypt victim's documents and ask a ransom for decrypting them (ransomware). A typical ransomware downloads a public key from a command and control point, and then it encrypts document files with such a key. Finally, it reveals itself by displaying a message to the victim with the instructions to pay the ransom. The victim cannot decrypt the files without the corresponding private key, which is owned by the command and control point. The ransom must be paid with an anonymous method, typically a cryptocurrency transfer. After the ransom is paid, the attacker gives to the victim a decryption program which embeds the private key.

If the writer's intent is investigation or espionage, malware can record the victim's actions in multiple ways: by microphone, camera, key strokes, browser history, etc.

## D.2 Anti-Virus software

The main functionalities of anti-virus software are:

1. Prevent malware's infection;
2. Detect malware's infection;
3. Remove malware's infection.

The first two features require the ability to recognize malware from legitimate software. Modern anti-virus software has other features, among which firewall, intrusion detection, anti-spam, and anti-phishing.

### D.2.1 Fred Cohen's result

Fred Cohen (the inventor of the word “computer virus”) developed the following theorem:

*“A perfect malware detector is impossible”*

A proof by contradiction follows:

1. Let us suppose a function `is_malware()` exists, which examines a piece of code and returns true whether such a code is malware, false otherwise. Let us suppose that `is_malware()` is perfect, that is it does not give any false negatives nor false positives.
2. Then, it is possible to build a program named `my_program()`, which executes `is_malware()` on itself. It behaves like a malware if the function return false, it does nothing otherwise.
3. `is_malware(my_prog)` cannot return neither true or false. If it returns true, then `my_prog()` will not be malware, thus it will be a false positive. If it returns false, then `my_prog()` will be a malware, thus it will be a false negative.

Hence, such a perfect malware detector is not possible.

### D.2.2 Signature-based detection

The real-life anti-virus software tries to recognize malware by means of two main techniques: *signature-based* detection and *anomaly-based* detection. A signature is a sequence of instructions in the code of a piece of malware, which univocally identifies it. The presence of a signature reveals the presence of malware inside an infected file or a running process.

The signature-based detection relies on a database of malware's signatures. Every suspect file and process are checked to contain such signatures. Signature database must be periodically updated by anti-virus software. Such a method is efficient, gives very rare false positives, and identifies the specific piece of malware, rather than detecting its presence only. The identification is particularly important for the sake of removing it, as different malware requires different removal procedures. This is the most used detection method by anti-virus software. The main drawback is that this method cannot recognize new malware, whose signature has not been isolated yet. Another drawback is that it does not protect against polymorphic malware, as we will see in the following.

Malware can self-encrypt itself in order to avoid signature-based detection. The decryptor and key are stored in the clear before the encrypted malware's code. During the first execution, the decryptor recovers malware's code and executes it. During the self-replication phase, the malware chooses a new random key and encrypts its code. Note that a strong encrypting algorithm is not needed here. The encryption aims only at avoid the detection, rather than protecting the confidentiality. Very simple encryption algorithms like XOR masks are often used.

This concealment technique is easy and efficient, but malware can still be recognized by means of the decryptor code. Such a code does not vary between infections, thus it can be used as a signature. However, self-encryption is still useful to avoid detection since it significantly reduces the code where the signature can be found.

A more advanced technique is *polymorphism*, which consists in changing the malware code without changing its behavior. A simple polymorphism technique is inserting NOPs (NO Operation) at random locations inside code. Another technique is to insert ineffective operations, like increments and successive decrements of the same registers. A third technique is to reassign registers, for example replace EBX to EAX.

### D.2.3 Sandbox-based detection

Sandbox-based detection tries to detect malware by executing suspect software inside *sandboxes*, which emulate real hardware in a controlled and isolated manner. Malware must be unaware to be running inside a sandbox. All the actions of the suspect software are logged, and then the logs are analyzed to detect malware behavioral fingerprints. This

detection technique can protect against self-encrypting malware, polymorphic malware, and variants of known malware. However it is slower and less efficient, so it is rarely used in end-user antivirus software.

Moreover, malware can prevent being run inside a sandbox, or detect it at runtime by accessing non-emulated resources. For example, it can use instructions performed by the Floating Point Unit (FPU). Specific tools exist which replace normal operations with FPU operations. If malware detects to be running inside a sandbox, it typically stops all malicious operations.

## D.3 Reverse engineering

Reverse engineering is the process of analyzing a software system in order to create a higher-level representation of it. Reverse engineering can have many purposes, among which malware analysis. A simple reverse engineering technique is text extraction, which aims at recovering sequences of printable characters from the executable code. It can extract output messages giving hints on the program's functionalities, or sensitive information like hardcoded passwords. If the executable code includes debug metadata, text extraction can also extract the names of the functions declared in the program and in statically-linked libraries, which give further hints on the program's functionalities. An example of text extractor is the "strings" command in Unix.

Library call tracing is another simple reverse engineering technique aimed at tracing all the calls that a program performs to dynamically-linked libraries (e.g., .DLL in Windows, .SO in Unix) with the relative parameter values. Library call tracing acts by inserting *shims* between the target executable code and the shared libraries that it uses. A shim is a library that transparently intercepts calls. Library call tracing cannot trace calls to statically-linked libraries, because they are compiled in the same executable file of the program. An example of library call tracer is the `ltrace` command in Unix.

### D.3.1 Decompilation

*Decompilation* is a technique aimed at creating high-level language source code that (once re-compiled) behaves in the same way of a given executable piece of code. It is thus the inverse operation of compilation. Decompilation is one of the main techniques of reverse engineering, and it is extremely useful for analyzing malware and for cracking software protected by digital-rights management techniques. An example of decompiler is Ghidra.

Decompilation cannot usually reconstruct perfectly the original source code. The quality of the output source code mainly depends on the amount of information that can be extracted from the executable code. The presence of debug metadata helps decompilation since it makes it possible to reproduce the original names of the variables and the functions, as well as the line numbers. The bytecode-format executables used by Java or .NET virtual machines are usually easier to decompile since they contain a lot of metadata and high-level features.

Decompilation is a complex process composed by several phases:

- *loading phase* involves the parsing of the input executable and the extraction of some information like the target architecture (Pentium, ARM, PowerPC, etc.), the employed compiler (gcc, Visual Studio, etc.), and the entry point of the program (i.e., the "main()" function);
- *disassembly phase* translates the architecture-specific binary instructions into an architecture-independent representation called *intermediate representation*;
- *program analysis* recognizes the high-level expressions ( $a=b*c$ , etc.);

- *data flow analysis* recognizes the variables;
- *type analysis* recognizes the variable types;
- *structure analysis* recognizes the control-flow instructions (if, while, switch, etc.).

These phases work by isolating and recognizing idiomatic code sequences, that are sequences of instructions that a given compiler uses to translate high-level instructions.

### D.3.2 Obfuscation

*Obfuscation* is a technique aimed at producing executable code which is harder to reverse engineer. This can be done by removing metadata from executable code, and by compiling in such a way to hide idiomatic code sequences.

## E Web security

The main technology behind any web application is HTTP (HyperText Transfer Protocol). HTTP uses a message-based model, in which a client sends an *HTTP request* and the server sends back an *HTTP response*. HTTP by itself is *stateless*, in the sense that it does not require the server to maintain a state over multiple requests. However, web applications that are on top of HTTP maintain their state both at the server side (e.g., by means of *session variables*) and at the client side (e.g., by means of *cookies*).

The core security problem of web application (like all the types of applications) is that users can submit arbitrary input. However, in web applications this is particularly difficult to address, since the possible inputs are many. In practice, all the content of an HTTP request is tainted: the requested URL, the parameters, the HTTP headers, the request body, etc. In addition, requests can arrive in any sequence, thus we cannot assume a particular order in the page requests, for example the order given by the links in the HTML code. This is true also in a non-adversarial situation, because honest clients can press the back button, or bookmark URLs to access them directly, etc. Another assumption that we have to make is that attackers may not use (only) browsers to attack our web application. There are many tools available that permit attackers to change anything in the HTTP request, read anything in the HTTP response (for example cookies, hidden fields, and HTTP headers) and perform requests in quantity and rates impossible with normal browsers.

### E.1 Web application mapping

There exist different actions for attacking web applications exploiting the HTTP protocol:

- *HTTP proxy* is a server that records, relays and possibly manipulates all the requests and the responses between a client browser and a web server. Attackers can use a local HTTP proxy when attacking web applications, in order to intercept and analyze HTTP requests and responses between her browser and the target application.
- *Web spider* (aka web crawler) requests pages to a target web site, analyzes them to extract links to other pages, and then requests those pages recursively. Advanced spiders are able also to automatically fill up input forms, execute client-side scripts that possibly produce links. Web spiders are used for benign purposes (e.g., mapping a website for a web search engine) as well as for malicious ones. In the latter case, the aim is to completely map the web application in order to discover the functionalities and identify the most convenient attack surfaces. Of course, web spidering cannot discover hidden contents, such that pages that are not linked by the public pages of the website.

The *robots exclusion standard* is a standard method to inform spiders about which pages or areas of a website should not be spidered. It works by providing a “*robots.txt*” file at the root of the website hierarchy, which contains a blacklist of URLs that spiders should avoid. Of course, malicious spiders can ignore robots. Exclusion directives, or even start spidering exactly from those URLs, under the assumption that blacklisted URLs may contain the most sensible information or the most security-critical functionalities.

- *Directory bruteforcing* is a technique that performs a large series of HTTP requests to a target website, trying different paths taken from a database of common directories, or combinations of them. The aim is to discover hidden content which is often security-critical, since it can contain hidden administrative functionalities or new features yet under alpha-testing. Hidden content can also contain backup copies

of dynamic pages. If such copies have a changed file extension (e.g., “.php.bak”), then the web server could not identify them as executable. Thus, requesting these copies would enable the attacker to view the server-side source code, in order to find vulnerabilities. Moreover, hidden content can contain backup archives of dynamic pages, which can leak server-side source code as well.

## E.2 Authentication

There is not a silver bullet to implement a secure authentication solution. Moreover, advanced security in authentication often means increased costs and decreased usability. The security criticality of the particular application must be considered when designing the authentication mechanism. More critical applications (e.g., e-banking) will need more security, less critical applications (e.g., a newsletter) will need more usability. The most secure solution is probably *multifactor authentication*. In multifactor authentication, the user identity is confirmed on the basis of multiple pieces of evidence of different types, for example a password and a physical token. Typically, authentication factors are categorized in three types: what the user knows (e.g., a password), what the user has (e.g., a smart card or a physical token), what the user is (e.g., his/her fingerprints). However, multi-factor authentication is expensive and cannot be applied in every application.

### E.2.1 Strong credentials and password change

The first and most effective countermeasure is to force users to choose strong credentials. Passwords should have a minimum length, they should be composed by lowercase, uppercase, numeric, and typographic characters, and they should avoid names and dictionary words. A password change functionality must be provided, in order to let users change their password in case they suspect it has been compromised.

### E.2.2 Secure credential transmission

After having enforced the password quality, credentials must be protected from compromise. All client-server communications must be protected by well-established cryptographic protocols like HTTPS, especially the requests in which clients transmit their credentials. The web page presenting the login form must be transmitted by HTTPS too. Otherwise, users cannot verify the authenticity of the login page, thus it is not guaranteed that the credentials will be submitted to the real server.

Credentials should never be transmitted via URL parameters, for several reasons. First of all, URL parameters are displayed on screen on the URL bar of the browser, and this could lead easily to credential stealing. Moreover, if the user follows an off-site link leading to another website, the browser will include the old URL together with the parameters in the HTTP request, inside the “Referer” HTTP header. Finally, visited URLs are usually logged in several unsafe places, like browser’s history, server logs, and logs of possible proxy servers. Saving credentials in cookies is generally considered unsafe too, because they can be stolen via client-side attacks, for example cross-site scripting (see after). The safest way to transmit credentials is inside the HTTP body (i.e., through a POST request), which is not displayed on screen, logged anywhere, or transmitted off-site.

### E.2.3 Secure credential storage

Finally, credentials should be stored at server side in such a way that it is infeasible to recover the original value, even in case the server’s database gets compromised. A common way to do that is to store a salted (i.e., randomized) one-way hash function of

the password. An example of hash algorithm for password storing is the *BCrypt* algorithm, which is included in the PHP 5.5.0+ standard.

#### E.2.4 Fail-Open Flaws

The application logic that checks the credentials must be carefully designed to be free of fail-open flaws. A *fail-open flaw* is a vulnerability in which an application grants access (instead of denying it) in case an error or exception is encountered. These kind of flaws are especially prevalent in those programming language strongly based on exception, for example Java. In these cases, a good practice is to surround the core authentication check with catch-all exception handlers which denies access.

#### E.2.5 Verbose errore messages

In case the user fails to login, the error message returned by the application should not give any hints (explicit nor implicit) about which part of the credentials is wrong, because this information could help attackers. For example, messages like “wrong password for user ‘Alice’” tells an attacker that a username “Alice” actually exists, thus leaving space for a user enumeration attack, which could allow the attacker to mount more *dangerous attacks*. The application should always return the same generic uninformative message, for example “invalid credentials”.

#### E.2.6 Account locking and CAPTCHAs

Countermeasures must be taken to avoid bruteforcing against login mechanism, as well as against related mechanisms like password changing, password recovery, etc. Preventing user enumeration significantly mitigates the possibility of login bruteforcing. Another common mitigation is *account locking*, which is a technique by which the authentication is suspended for a specific account for some time (e.g., 30 minutes) after a given number of failed login attempts (e.g., three). Account locking is considered safer than other similar locking countermeasures, for example session locking or IP address locking. With session locking, the application suspends the session after a given number of failed login attempts. However, an attacker can easily start a new session by deleting the session token. With IP address locking, the application suspends the client’s IP address after a given number of failed login attempts. However, an attacker can easily change her IP address.

In addition to account locking, another countermeasure against bruteforcing attacks are CAPTCHAs. A CAPTCHA (Completely Automated Public Turing Test to Tell Computers and Humans Apart) is a challenge-response test used to determine whether the user is human or not. A simple example of CAPTCHA is an image displaying distorted letters. The human user is able to reconstruct the original letters, and thus to solve the challenge. On the contrary, this is a hard problem for software, even if aided by machine-learning techniques.

#### E.2.7 Account recovery

User often forget their passwords. The account recovery functionality must be carefully designed to avoid possible vulnerabilities. The best way to implement an account recovery functionality is to send to the user an email (configured at registration time) carrying a unique, time-limited, unguessable, single-use recovery URL. When the user visits such an URL, (s)he is allowed to set the new password without exposing the old one in any way. To further protect the account recovery functionality, the application could present the user with a secondary challenge before sending the email, for example the name of his/her first school, and so on.

## E.3 Session management

The security of a session management mechanism depends on how well a web application handle the related session tokens. Session tokens must be created in such a way they are unpredictable, and they should not have an internal structure nor meaning (even obfuscated). The best session tokens are long sequences of random bytes generated by a secure random number generator, and associated to server-side session data. All data about the current application state should be stored at the server side, and the session token should act only as a key to access it.

An alternative to random session tokens are encrypted session tokens. With encrypted session tokens, the current state of the application is entirely stored at the client side (e.g., with a cookie) in an encrypted form, typically by a symmetric key known only by the server. Although this technique makes the server more lightweight because it outsources all the state information to the clients, it is considered less secure and rarely used nowadays. Indeed, the security of all the web application depends on the strength of the encryption mechanism and on the symmetric key(s) used to encrypt and decrypt. If the encryption mechanism is somehow flawed, for example because it permits an attacker to forge or tamper with session tokens, or if the symmetric key(s) are not well-protected, then all the security is jeopardized. Moreover, this technique does not allow the server to perform data mining on the application states, which could be a valuable source of information.

### E.3.1 Secure token transmission

After session tokens have been created, they must be protected from compromise during their entire lifecycle. Indeed, if an attacker steals somehow the token of an ongoing session of a legitimate user, she can inject requests on the same session, thus impersonating the user. This is called *session hijacking attack*. First of all, tokens should be transmitted only over confidential channels, such as HTTPS. If HTTP cookies are used to transmit tokens, they should be flagged with the “secure” attribute. The “secure” attribute tells the browser that the cookie should be transmitted only over HTTPS. In this way, even if the website contains non-HTTPS pages (usually static resources like help pages) the session token does not risk being compromised.

Session tokens should not be transmitted in URLs, because this is unsafe for the same confidentiality reasons explained before for passwords: URLs appear on screen, on browser history, on server logs, etc. In addition to these reasons, sending session tokens in URLs exposes the users to session fixation attacks. In a *session fixation attack* the attacker induces a legitimate user to follow a particular URL (e.g., by sending an email with a link inside) containing an untrusted session token, which has been previously created by the attacker herself. In this way, the attacker can obtain the same effects of a session hijacking.

### E.3.2 Session termination

The application should implement a logout functionality, with which the users can delete all the server-side session information and invalidate the session token. However, users must not be supposed to explicitly log out at every session. Thus, sessions should also automatically expire after some inactivity period (e.g., 10 minutes).

## E.4 Code injection

One of the most common and devastating attacks in the web is the *SQL injection* attack. Bypassing authentication is not the only possible impact of a SQL injection. Other effects

include escalating privileges, stealing data, adding or modifying data, partially or totally deleting a database.

The aforementioned problem is not specific of SQL, but of all the interpreted languages, for example LDAP, XPath, etc. Such languages are very prevalent in web applications, especially for interfacing back-end components like databases. In a typical web application, user-supplied data is received, manipulated, and then acted on possibly by means of some interpreter. In this case, the interpreter will process instructions of the web programmer mixed to user-supplied data. If this mix between instructions and tainted data is not done properly, an attacker can send crafted input that breaks out of the data context, usually by injecting some syntax that has a meaning in the interpreted language. As a result, part of the attacker's input is interpreted as code (*code injection*) and executed as legitimate code written by the programmer.

#### E.4.1 Tautology

The *tautology* technique consists in injecting a condition which always evaluates to true (e.g., `1=1`) in a WHERE clause in order to bypass it somehow. Imagine the following SQL statement:

```
SELECT * FROM users WHERE pwdHash='$pwdHash' AND name='$name'
```

In this case, the simple commenting technique cannot be applied. However, the attacker can bypass authentication anyway with the tautology technique, by injecting the following input: `$name = admin' OR 'a'='a`. Since the WHERE clause is always true, then the query will return all the tuples of the 'users' table. Thus the attacker can bypass the authentication. Note that here we did not use a trailing comment symbol (--) but a more «elegant» quote balancing technique, which consists in injecting an even number of quotation marks in such a way all string literals are «closed» in the resulting query, thus avoiding syntax errors. The final effect is the same as using the comment symbol, but quote balancing works even in the case the comment symbol is sanitized by the application. The clause commenting and the tautology techniques can also be used to steal data. For example, a webpage that displays some information of the current user can be forced to display information of all the users by making the WHERE clause evaluate always to true.

Moreover they can be used not only with SELECT statements, but only with INSERT, UPDATE and DELETE statements.

#### E.4.2 Finding SQLI bugs

There is a “safe” test to find out (without damaging the database) whether a user-supplied input is injectable and it is incorporated into a SQL statement as a string literal, that is between quotation marks. First, try to submit a single quotation mark as input and see the application response. If the application returns an error or a message divergent than normal, it could be an injection flaw. However, it could be also an error due to a sanitization check that rejects inputs with quotation marks. A second check is to submit two quotation marks and see the application response. If the error disappears and the application returns a “normal” message, then the input is probably injectable and incorporated as a string literal. Indeed, two successive quotation marks are interpreted as an escape sequence that SQL interpreter replaces with a single literal quotation mark.

When an input field is a number, it could be incorporated into a SQL statement as a numeric literal as well as a string literal. In the latter case, the attacker can employ the tests saw before. To “safely” find out whether a user-supplied input is injectable and it is

incorporated as a numeric literal, the attacker can submit a numeric expression like `1+1` and see the application response. If the application returns a “normal” message, equal to the message returned when submitting `2`, then the input is probably injectable and incorporated as a numeric literal.

#### E.4.3 UNION operator

By leveraging the UNION operator, an attacker can deceive the system into returning data from a table different from the one specified by the SELECT statement. The attacker injects a UNION operand followed by another query. The UNION operand merges the tuples resulting from the first and the second query. In order for the attacker to inject a UNION operator without errors, the two operands must have the same number of columns and with the same or compatible type. To discover the number and type of the columns of the injectable query, the attacker can employ the NULL literal constant, which is convertible to all the other types, injecting different number of NULLs until the application returns a “normal” message. The number of injected NULLs will be the number of columns of the injectable query. Note that the NULL values may be not viewable from the browser, since they could be rendered as empty strings on screen. In this case, an inspection of the HTML code could be necessary.

Then, the adversary has to discover the type of such columns. Actually, it is sufficient to discover a column of string type by means of which the attacker can steal any other type of data. To do this the attacker can inject a string in one of the fields, setting the other to NULL and note which input returns a “normal” message.

Finally, the attacker has to know the name of the table and of the table’s columns to steal. To do this, the attacker can use the UNION operator to steal the database metadata from the predefined table ‘information\_schema.columns’.

#### E.4.4 Using inference

Sometimes the application does not return the results of the query nor the errors generated by the database. In these cases, the attacker can extract data indirectly, by inferring it from the application’s behavior. This attack is called *blind SQL injection*. It works by injecting a query that triggers some detectable behavior if and only if the target information satisfies a particular condition. For example, let us suppose a vulnerable login functionality which uses the following query:

```
SELECT * FROM users WHERE username='\$input1' AND pwd='\$input2'
```

Suppose that for the attacker is not enough to login as administrator, but she also wants the administrator’s password, which is stored in the clear in the database (without any hash algorithm). Note that there is no way of stealing data with the UNION query method because the query results are not returned by the application. An attacker can submit the following crafted input:

```
\$input1=admin' AND SUBSTRING(pwd,1,1)='a' --
```

In this way, the attacker can learn whether the first letter of the administrator’s password is ‘a’ by simply checking whether the login succeeded or not. If the login succeeds, then the first letter is ‘a’, otherwise the first letter is not ‘a’. By submitting a large number of such inputs, the attacker can cycle through all the possible values of the first letter, and then through all the letters, eventually discovering the complete password. A SQL exploitation tool like JHijack can help in automating such a process.

Note that the administrator's password should be stored in hashed format inside the database, so discovering the password hash could be not an extremely dangerous leakage. However, this method is general and works for any type of information stored in clear, for example credit card information. Data can be stolen also from different tables by leveraging subqueries.

#### E.4.5 Piggybacked query

With the *piggybacked query* technique, an attacker injects another arbitrary query after the legitimate query. With piggybacked queries, it is possible to modify or add data, for example change the administrator password or add a new adversary-controlled administrator user. It is also possible to delete partially or totally a database, thus causing a denial of service and a loss of valuable information.

Query piggybacking is the simplest and most powerful attack method, nevertheless it is rarely used in practice. This is because the attack is possible only if the victim website performs vulnerable multi-queries on the SQL database, or if the SQL database is configured to accept multi-queries.

#### E.4.6 Countermeasures

Whitelisting inputs and rejecting them if they contain invalid symbols is always a good practice, but it may be insufficient for SQL injection. Indeed, many SQL symbols appear in legitimate field values (e.g., the name «Randy O'Brian» contains the operator «and» and the symbol «'»). If we accept the «'» symbol, then the system will be vulnerable. If we reject it, then the system will reject legitimate names (false positive). A more definitive solution are *prepared statements*, which is a technique by which the query is built in two separate stages: SQL code first and then input parameters. Prepared statements were originally introduced for improving the performance of executing many times the same query with different parameters. As a secondary effect, prepared statements also make SQL injection attacks infeasible.

#### E.4.7 Injection vectors

An attacker can leverage multiple attack vectors to carry out a SQL injection. Apart from POST/GET parameters, other common attack vectors are cookies, and server variables for example HTTP header fields like the browser name and version (*user agent*). Injection attacks through server variables are particularly common in log operations, for example if a website stores in a log database the user agent and the date of each client. The database and other server-side variables can be an injection vector, too. Some applications correctly sanitize data when they first receive and store it, but then they retrieve it again and use it to unsafely build SQL queries. The attacker thus can submit malicious input that is correctly escaped when received and stored in the database, and then retrieved again and injected to SQL queries. This is called *second order injection*. All these infection vectors must be carefully examined and sanitized for SQL injection bugs.

#### E.4.8 LDAP injection

SQL is not the only language subject of injection. Other examples are LDAP filters (*LDAP injection*), shell commands (*OS command injection*), Hibernate queries (*Hibernate injection*), XPath expressions (*XPath injection*), etc. LDAP (Lightweight Directory Access Protocol), which is a protocol for accessing and maintaining a directory service, which is a sort of hierarchical non-relational database that binds names to resources.

LDAP databases are less powerful than relational databases as they cannot process complex transactions. However, they offer high performances when reading data. This makes LDAP very used to maintain databases of users and credential, which are often read and seldom updated. LDAP databases are queried by means of LDAP *search filters*.

An LDAP database simply associates names (e.g., a user name) to resources (e.g., a password hash). It can be queried with an LDAP search operation, which takes a search filter as a parameter. An LDAP search filter is specified in a particular string format, which could be subject to injection (*LDAP injection*) if not properly protected.

## E.5 Cross-site scripting (XSS)

*Cross-site scripting* (XSS) is an attack by which an attacker injects some malicious client-side code (e.g., Javascript code) in a web page provided by a legitimate server. Such malicious code is then executed by the victim client. For example, the attacker could submit a form injecting HTML code like a `<script>` tag containing some malicious Javascript code.

The attack follows this schema: the adversary deceives a victim into following a particular link to a honest server, which contains the injected code as an HTTP GET parameter. The honest server then forms a web page containing the malicious code, which is finally executed by the victim. The malicious code can do various things. Typically, it sends somehow to the attacker the session ID that the victim user has established with the honest server, in such a way the attacker can do operations (e.g., money transferts) on behalf of the victim user (*session hijacking*). Cross-site scripting is particularly dangerous because the client browser trusts the honest server, so it could be configured the execute scripts sent by the server without any protection.

Another, more devastating, technique is the *stored XSS*. In the stored XSS, the attacker makes the server store the injected malicious code, so that the server builds web pages containing it to many clients.

A general countermeasure is escaping all the untrusted inputs before including them in the HTML code.