# COMPUTER AND NETWORK SECURITY

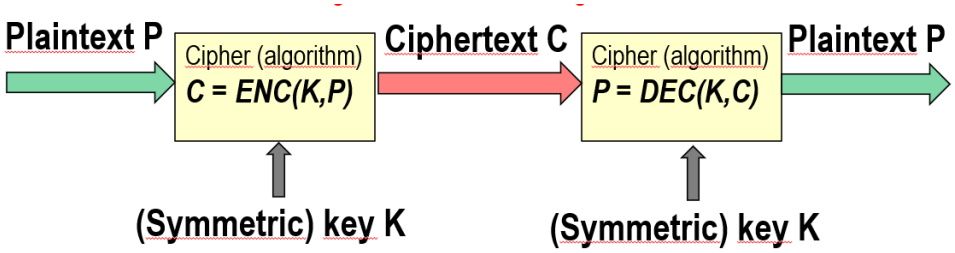
## PARTE 1 - BASICS

## Sommario

# 1. ENCRYPTION

- È una tecnica che serve a proteggere la confidentiality dei dati trasformandoli in qualcosa di incomprensibile.
- La trasformazione deve essere reversibile, ed avviene applicando una chiave al plaintext, dal quale si ottiene il cipher text.
- L'encryption può essere
    - ***Symmetric*** usa stessa chiave per criptazione e decriptazione,
    - ***Asymmetric*** usa diverse chiavi ed è molto più complessa.

Gli algoritmi di encryption e decryption possono essere pubblici, il segreto deve essere la **chiave**!




## 1.1. Tecniche di cifratura

### 1.1.1. Substitution cipher

Usata storicamente, è una tecnica che associa univocamente lettere fra loro (A → B, C→! B).

- Il **problema** di questo metodo è che già solo osservando il testo criptato si può arrivare al messaggio decriptato senza chiave, perché studiando la frequenza di utilizzo certe lettere della lingua di riferimento, nel caso italiano ad esempio studiando le doppie che sono consonanti, posso capire con discreta facilità l'algoritmo adottato.
- Questo è il processo di ***crypto analisys***.

Quindi, mentre il substitution cipher è il peggior metodo possibile, il *one time pad* risulta il migliore.

### 1.1.2. One time pad (o Vernam cipher)

Se ho a disposizione un testo che vedo come sequenza di bit che voglio cifrare, creo un'altra sequenza di bit random lunga quanto il plain text che mi servirà da chiave. La tecnica consiste nel fare lo xor fra plain text e chiave.

Eseguire nuovamente lo xor fra testo cifrato e chiave, permette di riottenere il plain text. L'utilità di questo metodo sta proprio nel fatto che il processo di cifratura è uguale a quello della decriptazione. Infatti:

$$A \oplus 0 = A$$

$$A \oplus 1 = A', A' \oplus 1 = A$$

Per l'applicazione corretta della tecnica servono 3 assunzioni:

1. **Servono tante chiavi quanti messaggi**

Se infatti uso la stessa chiave per due messaggi, facendo lo xor tra i messaggi cifrati ottengo la chiave, cosa non buona. Posto C messaggio cifrato, M plain text e K chiave:
$C1 \oplus C2 = (M1 \oplus K) \oplus (M2 \oplus K) = \ldots = K$

2. **Ogni chiave deve essere lunga quanto il messaggio**
Per rendere la chiave lunga quanto il messaggio, posso ripetere i caratteri della key se sono troppo pochi, o ci sono comunque tecniche per ovviare questo problema.

3. **La chiave deve essere true random**
Bisogna focalizzare la differenza fra pseudo random (esempio la funzione rand() di C), definita pseudo perché il numero in uscita è generato da un algoritmo. True random è quando dei valori vengono definiti da cause fisiche naturali, non da algoritmi.

Quando criptiamo un messaggio, è importante focalizzare il concetto che lo facciamo sempre in base ad un nemico da fronteggiare, mai a caso. Non esiste il concetto di "mettere in sicurezza" e basta, ma solo del mettere in sicurezza da una minaccia.

## 1.2. CIA (principio base della cifratura)

**Confidentiality:** Misura adottata per evitare che le informazioni sensibili arrivino alle persone sbagliate, assicurando che solo chi sia autorizzato possa accedervi.

**Integrity:** Assicura che il messaggio arrivi integro e non modificato da possibili attacchi esterni.

**Availability:** Mantiene aggiornato e bug-free il software che veicola lo scambio del messaggio

Il punto debole di questo metodo è che il messaggio può essere cambiato facilmente da chiunque, anche se è difficile da decriptare. Per esempio, con un attacco di tipo MITM(Man in the Middle) si può flippare un bit della sequenza criptata, cosi da modificare il contenuto del messaggio, senza che nessuno se ne accorga. Si definisce che questo metodo non è **INTEGRATIVO** poiché non viene garantita l'integrità del messaggio nonostante ci sia elevata confidenzialità**.**

### 1.2.1. Esempio:RFID

L' RFID è un sistema composto solitamente da un TAG e un READER (per esempio la carta e il pos). È necessario che tra i due oggetti ci sia una mutua autenticazione in quanto sia il TAG che il READER potrebbero non essere autentici, in modo tale da ricavare informazioni dall'altro per decifrare il sistema.

**Soluzione SBAGLIATA**
Il TAG è composto da
- Un secret statico ***S***
- Una chiave temporanea ***K***

Il secret viene cifrato tramite l'operazione di xor con la suddetta chiave associata all'utente.

Quando esso viene a contatto con il READER, quest'ultimo ottiene il segreto cifrato, e sarà in grado di decifrarlo qualora fosse genuino poiché avrebbe un database interno con le chiavi degli utenti a cui fare riferimento. In questo sistema, ogni chiave è generata ***randomicamente*** a partire da quella precedente (pseudo random). Il READER procede quindi, una volta decifrato il segreto, ad associare una nuova chiave temporanea al TAG in questione, aggiornando il suo database.

Risulta quindi evidente come il secret non venga trasmesso in clear, ma venga invece ***criptato*** usando una pseudo-random key.

Ciò denota dei **problemi**:

Sia S = random quantity e K = pseudo-random, allora

- Se S (che è random) = plain text → la chiave è K, che però è pseudo-random → **violo proprietà 3** (la chiave non è random).
- Se K = plain text → S è la key, che però è random → **violo proprietà 1** (posso avere la stessa chiave associata a due messaggi differenti).

Studiando infatti ogni combinazione di bit per cercare di capire il plaintext, basta eseguire lo xor tra la sequenza casuale con quella ottenuta applicando l'algoritmo pseudo random a partire dalla sequenza di base stessa. Così la chiave temporanea può essere individuata, compromettendo il sistema.

Affinché un sistema possa ritenersi sicuro deve almeno soddisfare una base, ovvero:

## 1.3. IND-CPA: Indistinguishability under Chosen Plaintext Attack

Un attacker ha a disposizione due messaggi M0 ed M1 e li manda ad un esterno; quest'ultimo lancia una moneta e rimanda al primo C=ENC(K,M0 o M1)

- L'attacker avrà il 50% di probabilità di indovinare il messaggio corretto, se C rappresenta M0 oppure M1.

Posta l'esistenza di un ***oracolo***, che conosce l'associazione ***chiave – testo cifrato***, supponiamo che l'attacker mandi all'oracolo C entrando a conoscenza quindi dell'associazione fra i due. Una volta avvenuto ciò potrà sempre decifrare i messaggi successivi.
Si evince quindi quanto il **cambiamento della chiave sia importante**, in quanto farebbe in modo che l'attacker non abbia possibilità invece di capire di quale messaggio si trattava, poiché si vedrà arrivare ogni volta messaggi diversi.

**Indistinguishability**

- Un avversario non deve essere in grado di ricavare alcuna informazione su un plain text a partire dal cipher text, anche se egli può avere accesso ad un oracolo.
- Inoltre, fornendo due plain text all'avversario di stessa lunghezza e gli viene dato anche un cipher text che è uno di quei due messaggi, lui non deve essere in grado di capire quale dei due sia. E nonostante questo lui ha ancora il 50% di indovinare a caso!

**IND-CPA conseguenze:**

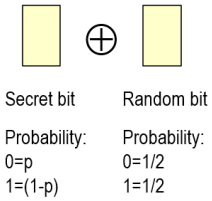
- L'***encryption*** deve essere ***randomizzata,*** poiché uno stesso messaggio deve sempre essere criptato in diversi cipher text, che deve essere indistinguibile da uno randomico.
- Se la sequenza si ripete essa deve essere ***criptata*** con una chiave differente.

## 1.4. Perfect secrecy

Utilizzando la funzione XOR applicata su una stringa random, essa nasconde qualsiasi informazione del plain text. La probabilità di decifrare il plain text dopo aver osservato il cipher text è la stessa dell'osservazione a priori.

Si osservi che se si somma la probabilità di ottenere lo 0 o 1, otterremo che p/2 + (1-p)/2 = 1/2!!!

**Concludendo, il Vernam cipher non è realistico.**




| Secret bit | Random bit | XOR result bit | |
|---|---|---|---|
| 0 | 0 | 0 | p/2 |
| 0 | 1 | 1 | p/2 |
| 1 | 0 | 1 | (1-p)/2 |
| 1 | 1 | 0 | (1-p)/2 |

# 2. STREAM CIPHER

Tipi di Stream Ciphers

- **Stream Cipher**
- **Block Cipher**
- **Block Cipher used in stream mode**

**Stream Cipher:**

È una categoria che cerca di imitare il Vernam Cipher. L'idea è quella di utilizzare una **chiave statica** che viene unita ad un vettore di bit chiamato **Initialization Vector** (così da rendere lunghezza chiave = lunghezza testo); Quest'ultimo vettore viene generato al volo casualmente ad ogni scambio di messaggi in maniera true random. Viene poi applicato un algoritmo Pseudo Random Number Generator (RC4-Salsa20/ChaCha20) all'unione tra chiave ed IV. Si ottiene così il **keystream,** il quale verrà applicato tramite xor al plaintext, cifrando il messaggio. Il keystream verrà inserito nel messaggio e trasmesso visibilmente. Questo perché si presuppone che il receiver possegga la key.

$$\Rightarrow ENC(KEY,MSG) = MSG \oplus PRNG(IV, KEY)$$

**Perché serve l'initialization vector:**

Gli IV sono generati per ogni messaggio in modo dinamico, quindi grazie a loro fortifico il mio sistema di criptaggio completando la chiave statica di partenza per adattarla al plaintext.

## 2.1. Esempio negativo: WEP (Wired Equivalent Privacy 1997-1999)

Era il sistema di sicurezza adottato dai primi WiFi, prima dell'uscita delle chiavi WPA/WPA2. Questo esempio è la dimostrazione che un sistema di criptaggio forte da solo non basta, bisogna anche saperlo usare. Trattandosi di uno stream cipher, generava il keystream sfruttando un algoritmo pseudo-random chiamato RC4 (oggi obsoleto). Se si sceglie di suddividere in pacchetti il MSG è necessario per ognuno di essi generare un nuovo keystream per riconoscerne il punto di inizio in caso ci sia dispersione di pacchetti con il WI-FI.

$$\Rightarrow ENC(KEY,MSG) = MSG \oplus RC4(IV, KEY)$$

WEP risulta quindi sicuro fino a quando non si ripete l'initialization vector. Se ciò accadesse non avrei più sicurezza semantica in quanto eseguendo M1 $\oplus$ M2 riuscirei a risalire alla chiave, risultando sensibile ad attacchi di tipo CPA o KPA (Chosen Plaintext Attack, Known Plaintext Attack).

ERRORI COMMESSI (Confidentiality):

- La dimensione dell'initialization vector era troppo corta, 24 bit. Ciò purtroppo aumentava terribilmente la probabilità che l'IV si ripetesse in pochi frame. Si può dimostrare come con una connessione a / Mbps, si possa trovare la chiave in meno di otto ore. Come se non bastasse, pensarono di poter risolvere il problema generando una key più lunga (si passò da 40bits a 128bits), scelta di fatto inutile, in quanto il problema risiedeva nell'IV.
- La generazione dell'IV era lasciata a cura di chi implementava il codice, cosa molto pericolosa in quanto esponeva ad un utilizzo inesperto/improprio il sistema. Per esempio, mettere tutti 0 o tutti 1 come keystream annienta la sicurezza in quanto lo xor di quei due stream fa sì che non venga

cifrato lo stream key+IV. Ciò non toglie il fatto poi che basta un semplice reboot del sistema a re-inizializzare a tutti 0 l'IV (ripetizione).

Per attaccare un sistema di questo tipo basta scrivere un dizionario dove ogni IV viene associato al keystream corrispondente; In una condizione di attacco di tipo KPA, conoscendo il plaintext si può facilmente rompere il sistema.

**ERRORI COMMESSI (Authentication):**

Definizione preliminare:

**L'autenticazione** è un servizio essenziale nella sicurezza che consiste provare la propria identità informatica. Essa consiste nel provare che le credenziali di un individuo siano autentiche, da non confondere quindi con l'identificazione (la quale consiste solo nel mostrare le proprie credenziali). Un'autenticazione andata a buon fine è tale se riesco ad essere sicuro che il soggetto che accede al servizio in un dato momento è lo stesso che ha acceduto in passato.

Mezzi di autenticazione:

-Conoscenza di un segreto: Password, PIN, chiave; Dimostro di conoscere un segreto.

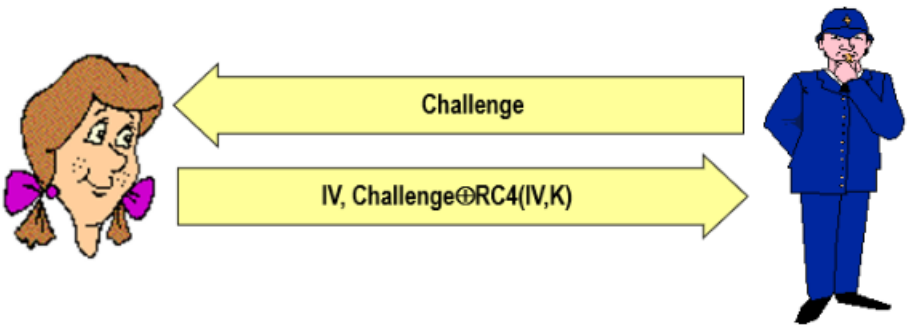
-Possesso esclusivo di un requisito fisico: Dimostro di avere qualcosa (smart card o dispositivi fisici), autenticazione basata su unicità di un hardware (Physically Unclonable Function).

-Autenticazione biometrica: Dimostrare di essere qualcuno sulla base del proprio DNA (impronta digitale e simili).

-Biometrica dinamica: Azioni di un soggetto, riconoscimento della voce, calligrafia o movenze.

L'idea dietro al WEP era non quella di autenticare gli utenti, ma quella di redigere una lista di chi poteva utilizzare la rete. Tutti gli autorizzati nella rete avrebbero potuto comunicare senza ausilio crittografico, mentre chi si trovava fuori la rete avrebbe osservato invece tutto cifrato. Questo si definisce concetto di "***Wired equivalent privacy***", ovvero la conoscenza di un segreto rende un utente fidato.

L'autenticazione nel WEP si basava sul seguente meccanismo: l'access point invia un valore numerico non criptato come plain text (definito challenge). L'utente che vuole accedere cripta il valore ricevuto in un modo da dimostrare la sua autenticità. Per ogni utente che arriva cambia la challenge (sequenza di 128 bit). Dato che la chiave è pre-condivisa, l'access point sarà in grado di riconoscere se il modo in cui l'utente ha criptato la challenge sia corretto o meno. Una volta criptata la challenge quindi l'utente invia all'access point un initialization vector scelto da lui unito alla challenge criptata in xor con l'RC4(IV,Key).



Arriviamo quindi agli errori:

- L'autenticazione WEP è basata sul fatto che la challenge non venga criptata quando inviata dall'access point, ovvero lascia che il plaintext sia visibile. Facendo Ciphertext ⊕ Challenge ottengo proprio la keystream. Il grande errore fu usare la stessa chiave nell'encryption e nella challenge. Il WEP in questo modo finisce per implementare un KPA, in quanto chiunque conosce la challenge e facilmente costruisce un dizionario IV - RC4(IV, KEY), ovvero il modo di violare questo sistema (in attesa che un IV si ripeta). L'autenticazione costituisce un problema per la confidenzialità.
- Solo ascoltando una persona fidata per il sistema che conosce la chiave si può riuscire ad aggirare l'autenticazione. L'errore sta nel fatto che l'initialization Vector è deciso dalla macchina di chi manda la challenge criptata indietro, ovvero da chi vuole entrare nel sistema. Così facendo, se si esegue

  **Challenge ⊕ (Challenge ⊕ RC4(IV, key)) = RC4(IV, key)** ottengo proprio il keystream

  A questo punto, il sistema è sensibile ad attacchi: posto un MITM, se esso ascolta un IV di un messaggio e secondo il suddetto processo si ricava l'RC4 corrispondente, è in grado di entrare nella rete; Se a quel punto chiede una challenge all'access point, può scegliere lui l'IV (ed userà il precedentemente intercettato in quanto così sarà coerente con l'RC4 che ha a disposizione), riuscendo ad entrare nel sistema. Un modo di evitare il problema sarebbe stato comunicare oltre la challenge anche l'IV (evitando che lo potesse scegliere l'user).

**ERRORI COMMESSI (Integrity):**

Il CRC32 è un algoritmo di error detection che può tener conto dei bit di partenza tramite bit di parità (posti alla fine del messaggio);

Se un hacker volesse modificare qualche bit del plain text può riuscirci senza che nessuno se ne accorga. Il messaggio M era seguito da CRC32(M). Quando il messaggio viene mandato vengono criptati tutti i bit. Quindi C (Ciphertext)= M|CRC32(M) ⊕ RC4(IV, Key). Per l'attacco si prende un vettore (delta) di bit contenente 1 nelle posizioni dove si intende flippare i bit nel messaggio originale.

**Chooses δ as long as M**
**Computes c(δ)**
**Computes:** $C' = C \oplus \{\delta, c(\delta)\} =$

$$= \left[RC4(IV,K) \oplus \{M, c(M)\}\right] \oplus \{\delta, c(\delta)\} =$$
$$= RC4(IV,K) \oplus \{M \oplus \delta, c(M) \oplus c(\delta)\} =$$
$$= RC4(IV,K) \oplus \{M', c(M \oplus \delta)\} =$$
$$= RC4(IV,K) \oplus \{M', c(M')\}$$

Importante è concatenare al delta anche il CRC(delta), in quanto va a sommare la nuova quantità di 1 nel vettore esterno facendo in modo che la parità misurata dall'algoritmo di controllo non risulti errata. Il problema del CRC è che è lineare, un algoritmo non lineare non avrebbe dato questi problemi in quanto sarebbe stato più complicato.

# 3. SECRET VS PASSWORD

Concettualmente essi sono la stessa cosa, in pratica c'è una grande differenza:

**Secret**: sequenza di bit costituita da numero random di caratteri.

**Password**: stringa caratterizzata da low entropy, meno sicura di un secret (la probabilità di indovinarla è molto più alta).

La password presenta 4 problemi:

1. Password overload : gli utenti riusano la stessa password per diversi siti.
   - Da una statistica americana si evince che il 38% utenti globali riusano stessa password su diversi siti, il che mette in pericolo la sicurezza. Risulta anche che il 21% degli utenti che volendosi sentire più sicuri modificano la propria password, lo fanno in modo prevedibile, non radicalmente quindi.
2. Restricted charset : non tutti i caratteri sono usati, in quanto sui 256 caratteri totali, sulla tastiera ce ne sono solo 102
   - Un byte sono 256 possibilità. La probabilità di indovinarne 8 di fila è 1/(256^8). Una password di 8 byte significa che la selezione non sia su 256 ma su molte meno.
3. Low entropy : la password va ricordata, quindi considerando l'individuo che la pensa, in realtà non è proprio random.
   - C'è un modo di misurare quanto qualcosa sia random (concetto di randomness). Per farlo bisogna osservare il grado di entropia della password sfruttando il cosiddetto ***Shannon Entropy Theorem (vedi 3.1)***.
4. Predictability : una password può essere scelta sulla base delle proprie esperienze o sulla base delle parole utilizzate da un individuo → cattive abitudini nella scelta delle password.
   - Le password sono soggette ai **Dictionary Attack**: L'idea è molto semplice, si prende un dizionario di parole comuni, sulla base di quelle che si dicono di solito o sui gusti comuni. Ci sono dei database di pubblico dominio di password ottenute da brecce eseguite in sistemi informatici, dove sono riportate anche le password più usate. La tecnica di **Password spraying attack** è proprio l'azione di provare tutte le differenti password più utilizzate.

## 3.1. Shannon entropy (Strumento matematico per misurare l'entropia)

Sia X una variabile discreta randomica (una lettera, un numero o simili). Sappiamo che essa potrebbe essere n caratteri, ognuno con una probabilità diversa (caso più generale non equiprobabile). Si definisce entropia di X il valore

$$H(X) = -\sum_i p_i \log_2 p_i$$

posto p come probabilità del carattere i-esimo. Il meno serve per far tornare il logaritmo positivo considerati i suoi argomenti (numeri minori di uno, che rappresentano una probabilità).

L'entropia si misura in bit, ossia la quantità di informazioni trasportate da X.

## 3.2. Entropy e predictability

**L'entropia** è un modo quantitativo di calcolare quanto un evento randomico sia prevedibile.

Posto $N = 2^b$ possibili risultati (outcomes), sia $b = \log_2 N$

- Entropia = 0 → "minima entropia" - caso deterministico
- Entropia = n → "massima entropia" - numero di eventi equiprobabili
- Nel caso intermedio tra 0 ed n ogni evento non ha la stessa probabilità di accadere.

Definito quindi il concetto, risulta che se i bit sono indipendenti si alza l'entropia, se sono dipendenti si abbassa. Le password pertanto hanno un'entropia più bassa dei secret in quanto le parole di una lingua hanno relativamente bassa entropia (la successione di lettere deve avere un senso).

# 4. APPROCCI PROTOCOLLARI PAP E CHAP

L'autenticazione è una prova di conoscenza, dove si vuole provare la conoscenza di una password/ segreto, non per forza rivelandoli.

## 4.1. Password Authentication Protocol (PAP)

È l'approccio di autenticazione più semplice possibile. Per dimostrare di conoscere la password si trasmette in clear. Questo metodo consiste nel mandare ID, PASSWORD all'authenticator che ha un database con scritte tutte le associazioni utente/password (le password risultano quindi pre-condivise). Esso procede quindi controllando che i valori inviati di id e password corrispondano; In caso affermativo garantirà l'accesso.

Questo protocollo ha però delle limitazioni:

1. Poiché la password e l'ID sono trasmessi in clear, se qualcuno ascolta mette a repentaglio la sicurezza.
2. Non c'è protezione da Replay Attack (ad un eventuale attacker basta ascoltare il messaggio che viene inviato all'authenticator per acquisire credenziali valide per entrare).
3. Non c'è protezione intrinseca sul numero dei tentativi per entrare (più tentativi a disposizione di un attacker gli garantiscono più possibilità di indovinare).

## 4.2. Challenge-Handshake Authentication Protocol (CHAP)

Nell'authentication io devo dimostrare di conoscere la password, ma non è detto che devo dirlo esplicitamente all'autenticatore come nel PAP. Nel CHAP non viene rivelato P, ma H(P) (funzione di computation), ovvero un prodotto di byte ottenuto a partire dalla password
Premesse importanti:

1. Questa funzione non deve essere invertibile, poiché in tal caso ad un attacker basterebbe intercettare il messaggio ed eseguire la funzione inversa.
2. H non deve essere una funzione solo di P, ma deve contenere delle freshness (nonce, challenge). H(P) non deve essere ripetibile.

L'approccio si sviluppa secondo la seguente modalità:

1. L'authenticator manda un ***nonce*** (ovvero un *fresh value*, una *challenge*, quindi una serie di bit). Il lavoro dell'authenticator è fare in modo che la challenge non si ripeta mai.
2. L'utente risponde con UID (User ID), Response=H(Challenge, password, altre cose in caso).
3. L'authenticator prende l'UID e vede nel database quale password è associata a quell'user; Conoscendo anche la challenge che aveva mandato quindi può riapplicare la funzione H da solo (usando la challenge e la password che conosce) per ottenere R. Se vede che le R dell'utente e

quella calcolata da lui corrispondono allora l'utente viene autorizzato ad entrare. In questo modo non deve invertire la funzione, la quale per definizione non deve essere invertibile.

Il vantaggio quindi del CHAP è proprio che non trasmette mai la password in clear. La funzione non invertibile che viene usata di solito è una ***Hash function***.

**Pro:**

1. CHAP protegge dai ***replay attacks***
   - Ma bisogna assicurarsi che challenge non si ripeta mai.
2. È avviato dall'autenticator
   - Un attacker può incontrare difficoltà a convincerlo a rimandare la challenge più volte.
3. ***Challenges ripetute***
   - L'autenticazione può essere ripetuta durante il tempo di connessione (a differenza di PAP dove viene eseguita solo una volta all'avvio). La frequenza e la tempistica delle challenge è poi gestita dall'autenticatore.
   - Permettono di limitare il tempo di esposizione a ogni singolo attacco.

**Contro:**

1. Il secret deve essere disponibile in formato ***plaintext***.
   - Non è possibile utilizzare database di password crittografate in modo irreversibile.

## 4.3. HASH FUNCTION

Funzione che riceve input di dimensione arbitraria (lunghezza qualunque di bit X) e che restituisce una dimensione precisa dei bit di partenza Y=H(X), detto ***digest***.

Per definizione ogni hash, criptata o non criptata, non è invertibile.
Una *hash function crittografata* gode della proprietà che anche solo un minimo cambiamento nel messaggio di partenza cambia completamente il risultato finale (a differenza della non crittografata).
L'obiettivo quindi è che la *crypto hash function* deve **approssimare nel modo più randomico possibile il messaggio di base**.

Nel *non crypto hash* può succedere che due valori di partenza diano una stessa Y di risultato (collisione), situazione che può essere sfruttata da un eventuale attacker. Nelle *crypto hash* l'attacker non ha modo di trovare due messaggi che una volta applicata H siano uguali.

### Proprietà delle crypto hash function

- **Preimage resistance (one way)**: dato Y risultato dell'hash function, risulta praticamente impossibile trovare la X di partenza. Dato che la dimensione di X è arbitraria, sono infinite le combinazioni di partenza, quindi è proprio impossibile tornare ad X partendo da Y. Questa proprietà vale molto più di quella di non invertibilità delle *non crypto hash function*. Importante metodo per garantire più resistenza ai ***brute force attempt*** (ovvero provare ogni combinazione una dopo l'altra) è utilizzare dei larghi digest, ovvero larghe H(X).

- **Second preimage resistance (weak collision resistance)**: Dato X, deve risultare estremamente difficile trovare un X' tale che H(X) = H(X'). Le funzioni hash restituite devono essere il più difficile possibile da ripetere su input diversi.

## 4.3.1. Cryptographic hash function

Le crypto hash function godono di 3 proprietà importanti:

1. ***Preimage resistance*** (**one way** – unidirezionale)
   - Dato Y = risultato hash → sarà difficile trovare un altro qualsiasi X tale che H(X) = Y
2. ***Second preimage resistance*** (**debole** resistenza alle collisioni)
   - Dato X → difficile trovare un altro X' tale che H(X) = H(X')
3. ***Collision resistance*** (**forte** resistenza alle collisioni)
   - Difficile trovare due generici X1 e X2 tali che H(X1) = H(X2)
   - È una proprietà più forte rispetto alla 2, poiché il messaggio di partenza X lo posso scegliere.

**Proprietà 1**

One way → molto più di "non invertibile". Infatti, se ho un digest "c6c8258947bffe06ea4…", ci saranno infiniti messaggi che possono generare tale digest, ma io non dovrei essere in grado di trovare nessuno di tali messaggi!

**Proprietà 2**

È una proprietà più forte della 1!

NB: ci sono funzioni che soddisfano la proprietà 1, ma falliscono la 2! Un esempio di ciò è il seguente:

**Modular exponentiation**

***$Y = F(x) = g^x \bmod p$***

- g valore noto
- p numero primo molto grande
- F(x) unidirezionale → soddisfa proprietà 1
- F(x) non soddisfa la proprietà 2

## 4.3.2. Birthday paradox

Ci sono K=23 persone in una classe (22 + io). Prendo una funzione hash che mi restituisce la data di compleanno di un individuo.

1. Qual è la probabilità che nessuno delle altre K-1 = 22 persone sia nata nel mio stesso giorno?

$$\left(1 - \frac{1}{365}\right)^{22} = \left(\frac{364}{365}\right)^{22} = 94.1\%$$

   ➔ Ogni persona ha meno del 6% di probabilità di avere il mio stesso compleanno

2. Qual è la probabilità che non ci siano 2+ persone nate nello stesso giorno?

$$1 \cdot \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{22}{365}\right) = 49.3\%$$

   ➔ In più del 50% dei casi due di noi hanno lo stesso compleanno!!

### 4.3.3. Hash digest size

Deve essere valutato rispetto al precedente paradosso del compleanno.

- 32 bits (RAND) → 4,3 miliardi di output!
  - Il problema però è che ho il 50% di collisione dopo $2^{16}$ ~ 60.000 msg (molto poco!)
- 56 bit (DES) → $2^{56}$ = 72.057.594.037.927.936
  - 50% di collisione dopo $2^{28}$ ~ 250.000.000 msg (ancora pochissimo! Pochi secondi sul mio PC)
- 128 bits (MD5)
  - 50% di collisione dopo $2^{64}$ = $1{,}8 \times 10^{19}$ (debole oggi)
- 256 bit (SHA256)
  - 50% di collisione dopo il $2^{128}$ = $3{,}4 \times 10^{38}$ → OK oggi

## 4.4. Qual è il migliore tra PAP e CHAP ?

Per rispondere alla domanda bisogna considerare i **modelli di attacco**:

- **Attacco al canale di comunicazione** (da parte dell'**eavesdropper**)
  - Se si desidera utilizzare PAP, è necessario proteggere il canale di comunicazione, poiché contro un eavesdropper per il PAP è la fine!
    - Infatti, esso vede il nome e la password che sono stati scritti → GAME OVER
    - Per esempio. accesso online tramite https, EAP-TTLS, ecc.

- **Attacco al database back-end UID-password** (da parte di un **hacker**)
  - L'attaccante penetra nel sistema e ruba l'intero DB password
  - Mitigazione: database delle password con hash!
  - Contro questo tipo di attacco il PAP è più sicuro

**PAP : hashed password database**

PAP è migliore per difendere i database, poiché si può proteggere il DB con delle funzioni hash (cosa che non succede per CHAP).
L'idea è di **memorizzare H(password), anziché la password stessa.**

La **sicurezza è migliorata**: se l'attacker ruba il database, deve ancora invertire la password!

- La sicurezza dipende dalla scelta di una password forte

**CHAP**

L'attacco avviene al backend UID- password database.

In CHAP non si possono applicare hash password, infatti non c'è modo di calcolare H(password, challenge) senza avere la password in clear (viene mandato H(challenge, password) e nel DB non posso tenere H(password) e la challenge cambia ogni volta). Quindi il DB dell'utente deve rimanere in clear.

→ CHAP è peggio di PAP contro un modello di attacco backend!

Quindi, se:

- attacker è eavesdropper → meglio CHAP
- attacker è backend → meglio PAP

Infine, quindi, non c'è un'unica soluzione, ma **tutto dipende da quale attacco si vuole difendere**!

### 4.4.1. Mitigation ("salt" esplicito)

È una tecnica per migliorare la protezione dagli attacchi back-end nel CHAP.
L'access point mi manda un salt (numero) oltre alla challenge.

- Il valore del salt è in clear
- Memorizzare H(salt, password).
- Io gli rimando l'ID, H(H(salt, password),challenge) → viene salvato il salt, che non cambierà fin quando non si hackera il sistema.

Ma i motivi e l'utilizzo sono diversi!

- L'attacker non può riutilizzare la password rubata, ma la forza della password e gli attacchi dictionary sono ancora validi.
- Si rigenera il DB usando nuovo salt dopo una violazione o dopo un certo periodo di tempo.

# 5. ONE TIME PASSWORD (OTP)

È una tecnica che differisce da one time pad. Si basa sull'idea di migliorare il PAP per avere una password diversa ad ogni successful attempt; questo garantirebbe al sistema una resistenza ai replay attacks. L'idea è quella di memorizzare più password associate ad uno stesso utente, il che però appesantirebbe troppo il DB.

Per ovviare al problema dell'appesantimento del DB si fa ausilio di **hash chains**; Si tratta di catene di hash applicate ad una password di partenza. Questa "catena di password" garantisce quindi che la password cambi ogni volta; ogni volta verrà così usata una password diversa per entrare generata a partire da quella prima. Il problema di questo metodo è che un attacker che ascolta una password poi è in grado di calcolarsi le successive. Una soluzione è usare una hash chain inversa, dove si parte dall' n-esima password generata e si torna indietro (in questo modo l'attacker anche se ne intercettasse una, potrebbe solo calcolarsi le successive). Questa soluzione funziona bene, ma estendendola su scala mondiale, ogni singolo utente richiederebbe il calcolo di n hash nel momento di inserimento della password (potenza computazionale non indifferente).

Mandare le password al contrario quindi va bene, ma bisogna risolvere questo problema computazionale. Si pone ora l'obiettivo di riuscirci senza fare n hash, ma facendone una sola.

Durante la registrazione l'authenticator calcola tutte le password offline fino a P[n+1] e si salva P[n+1]. In questo modo quando l'utente vuole accedere fornisce P[N], l'authenticator calcola P[N+1] applicando la funzione hash, e se corrisponde a quella memorizzata allora l'utente entra nel sistema. L'authenticator sovrascrive a quel punto P[N+1] con P[N] nel suo DB. La prossima volta l'utente fornirà P[N-1] e così via. Questa soluzione prevede che ogni volta l'utente si calcoli offline sul posto fino al P[N] desiderato. In questo modo effettivamente ad ogni authentication viene eseguita solo una hash da parte dell'authenticator.

Vantaggi:

- Ad ogni authentication viene eseguita una sola hash
- Il DB memorizza un solo valore per utente (valore già usato per aggiunta)
- Robustezza contro attacchi server-side (impossibile prevedere le password precedenti)
- La password può anche essere trasmessa in chiaro

Svantaggi:

- Serve un grande valore di n per evitare di finire le password a disposizione (servirà una nuova registrazione dopo)
- Vulnerabilità lato client (tiene memorizzato il seed della password)
- Possibilità di desincronizzazione (se per qualche motivo fallisce un accesso, si perde la catena, in quanto l'utente la volta dopo proverà ad entrare con p[n-1] in giù, mentre l'authenticator si aspetta ancora p[n])

## 5.1. 2-Factor authenticator

Questo tipo di autenticazione si ispira al one time password. Si ipotizza che sia client che server siano sicuri.

Ci sono diversi requisiti in questo sistema, come un one time authorization token (generato su un device differente o ricevuto su un canale differente come e-mail, SMS, ecc.). Oltre ciò, il tutto deve essere human friendly (bisogna usare hash che troncano le key a un massimo di 6/8 digit).

Ci sono 2 protocolli: HOTP (hash based OTP) e TOTP (time based OTP).

**HOTP**: Questo protocollo si basa sull'idea di utilizzare un contatore N. Sul server e sul client è memorizzata la secret k di un utente. Si può fare P[N]=H[K,N]. Questo fa in modo che non si tratta più di una catena, in quanto non sto applicando la hash ripetutamente sullo stesso digest di una password di partenza, ma ogni volta su un bit stream differente. Se un hacker conosce P[N]=H[K,N], non può calcolare P[N+1]=H[K,N+1]. La P[N] può essere trovata asincronamente senza calcoli ogni volta che si vuole, basta avere la key di partenza. La differenza quindi con la one time password è che non è più pre-calcolata e in ordine inverso.

**TOTP**: Ogni periodo di tempo viene cambiato il token N per calcolare la password, e non ad ogni accesso come nel precedente. Gli attacchi a questo tipo di protocollo si basano sul fatto che l'hacker possa manipolare il tempo della macchina, cosi da decidere lui come farlo scorrere per il cambio di token.

## 5.2. CHAP: mutual authentication

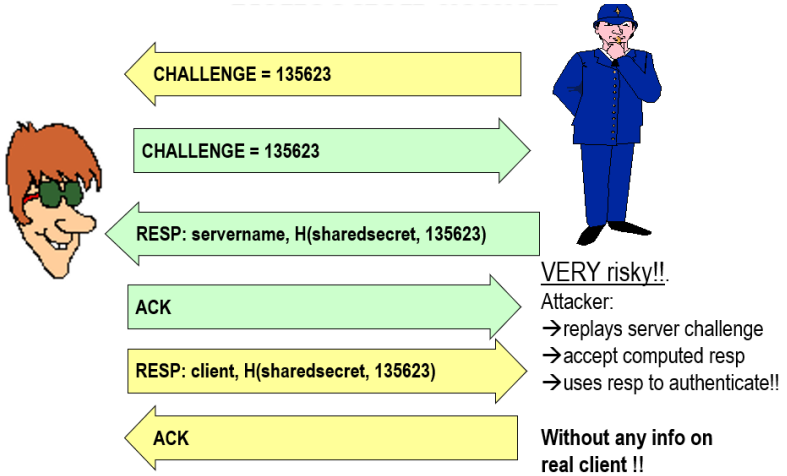
L'idea è quella di provare ad adattare CHAP (ideata per singole autenticazioni) ad una mutual authentication.

Il processo sarebbe:

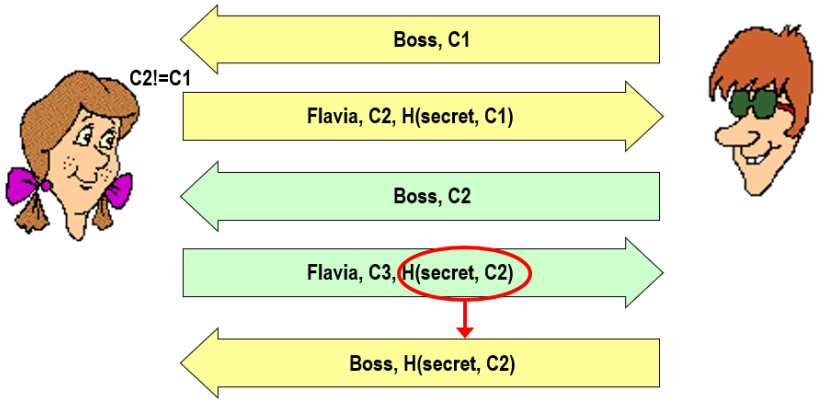
- Arriva la challenge, l'utente risponde con ID,H(sharedsecret, challenge), l'authenticator gli manda L'ACK.
- Ora l'utente rimanda una challenge nuova; l'authenticator risponde con serverName, H(sharedsecret, challenge nuova). L'utente manda l'ack anche ed è fatta.

Siccome non è specificato l'ordine dei messaggi, un hacker può rispondere alla challenge con la stessa challenge, cosi da ricevere come risposta dall'access point la stessa che dovrebbe dare lui per entrare.

Questo può accadere perché le due autenticazioni sono unilaterali e non è specificato né in che ordine vanno mandate, né che non possono essere uguali le challenge.

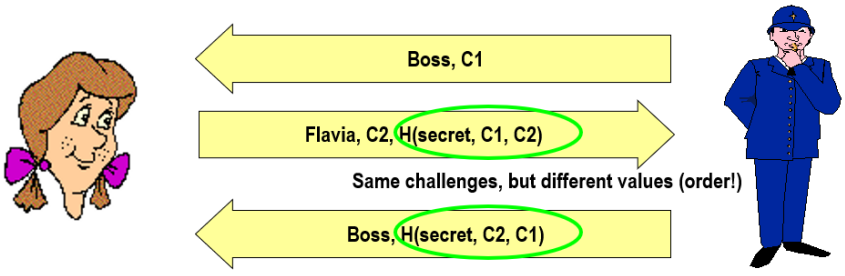



Oltre ciò, il sistema è soggetto anche al reflection attack; l'attacker si finge access point e si fa inviare da un utente ID, C2 e H(secret, C1). L'attacker fa finta di non sentirlo e rimanda la challenge che gli era stata inviata dall'utente. Così aggira i controlli in quanto gli verrebbe rimandato ID, C3, H(secret, C2); Così lui entrerebbe a conoscenza di H(secret,C2) che se lo rimanda riesce ad entrare nel sistema.




## 5.2.1. Come prevenire reflection attack

Se l'authenticator invia una challenge all'utente, e quest'ultimo propone una challenge diversa che include nella funzione hash insieme alla stessa che l'access point gli aveva mandato, rende il sistema sicuro dal reflection attack. L'authenticator rimanderà poi la funzione hash delle challenge in ordine inverso. Un hacker può sentire lo scambio di messaggi e può sentire le diverse challenge, ma senza il pre shared secret non può calcolarsi la funzione hash.




No reflection possible anymore

**Morale**: Mai usare un algoritmo per un obiettivo che non è quello per cui è stato pensato.

Il problema di CHAP (pensato per single side authentication) è che se usiamo 2 side autenticazioni, la loro indipendenza mi espone a rischi il sistema. È per questo che vanno rese dipendenti crittografando insieme le challenge.

## 5.3. Tipi di NONCE (fresh value):

- **Random Challenges** (sensibile al paradosso del compleanno che rende il sistema sensibile alla ripetizione di un valore).
- **Sequence number** (mandare tutti numeri in sequenza 1, 2, 3, … È più facile da predire certo, ma è più robusto delle random challenge, che si ripeteranno prima per il paradosso per il compleanno. Sensibile inoltre alla desincronizzazione).
- **Time stamp** (sensibile all'attacco del cambio del riferimento temporale).

## 5.4. Esempio pratico: 2G

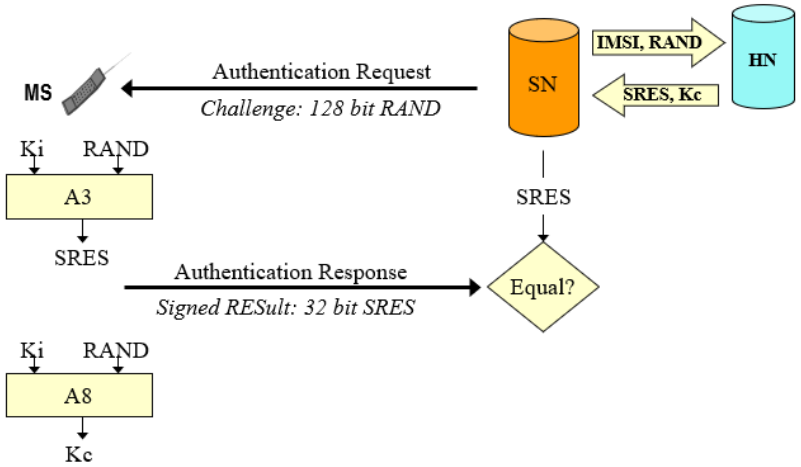
La comunicazione si basa su tre entità principali:

- **MS:** mobile sim (user)
- **SN:** Serving network (Visitor location register)
- **HN:** Home network

Quando avviene la connessione si susseguono i seguenti passaggi:

- L'user che vuole connettersi si identifica.
- SN manda l'IMSI (subscriber indicator, identificatore dell'utente) e RAND (una challenge) all'home network.
- HN risponde con SRES (ovvero H(RAND,Ki) tramite funzione hash A3) e Kc (encryption key che verrà poi usata nella sessione, **generata dinamicamente** che cambia ogni volta, generata con una funzione hash A8 a partire sempre da Ki e RAND).
- Ora che SN è entrato a conoscenza delle informazioni necessarie allo svolgimento della sessione, manda una authentication request a MS con una challenge (RAND) di 128 bit.
- MS crea 32 bit (SRES) a partire dai 128 di challenge che gli sono arrivati facendo ausilio della funzione hash A3 (conosciuta solo dalla sim e dalla HN, fornite dal provider). Questo digest di 32 bit è ottenuto a partire da Ki e RAND.
- SN controlla se gli SRES corrispondono, garantendo l'accesso in caso affermativo.

Punto di forza di tutto è che il Ki (identity key - chiave dell'utente) resta in HN e non esce dal dominio verso SN (grazie ad A8).

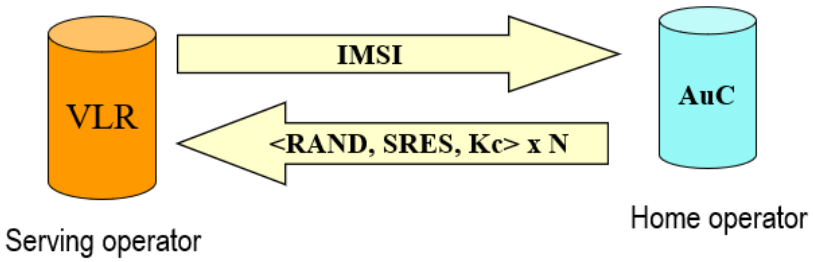


### 5.4.1. Concetto intermedio: TRIPLETS

Importante è il concetto di utilizzare vettori di dati. Si prendano le seguenti due entità situate in parti diverse del mondo.

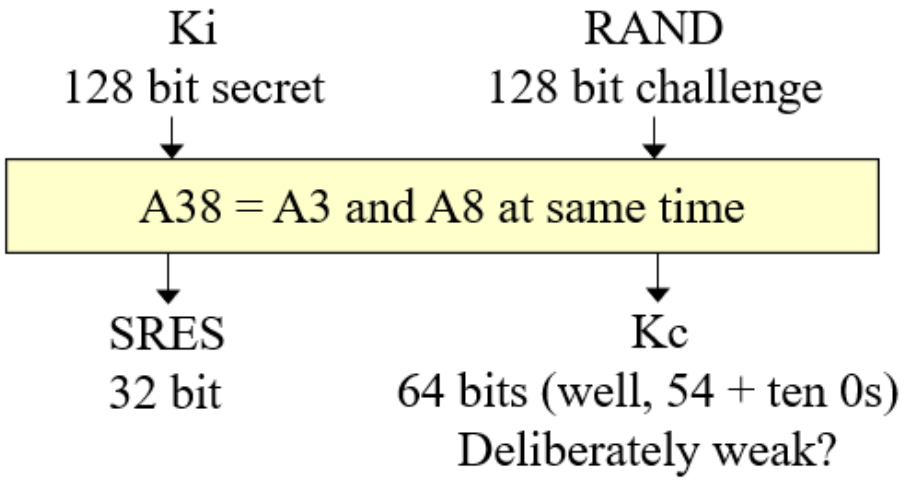
- **VLR:** serving operator
- **AuC:** home operator

Questa volta, VLR invia solo l'IMSI (e non la RAND). AuC risponde con un insieme di vettori di tipo [challenge,sres,Kc]; La risposta è formata da N triplette di questo tipo. Il vantaggio di ciò è la performance, perché vengono mantenute delle triplette in più per le prossime connessioni. Altro vantaggio è che questo mette tutta la sicurezza tutta nelle mani di AuC, poiché è quest'ultima che gestisce le nonce ; Si evita così che qualunque hacker possa provare ad attaccare le nonce cercando di crearle o manipolarle.




### 5.4.2. Problemi del 2G

Dopo l'applicazione di A38 (A3 e A8 allo stesso tempo), viene dato Ki e RAND (entrambi 128 bit), e si ottengono SRES (32 bit) e Kc (64 bit). Il problema fu che in realtà erano 54 bit perché alla fine c'erano sempre 10 zeri. Si perdevano in questo modo 2 alla 10 combinazioni di sicurezza.




Ogni operatore doveva scegliere come creare la sua funzione di authentication: poteva realizzare la sua specifica, oppure utilizzarne una già creata da qualcun altro (in quel periodo COMP128). Questo era il concetto di s**ecurity by obscurity** che si credeva erroneamente fosse sicuro, ovvero l'idea che se un algoritmo viene realizzato privatamente nessuno ha idea di come funzioni e non possa violarlo.

In realtà COMP128 è stato in futuro violato. Il punto è che se un algoritmo non viene realizzato da un esperto è probabile che venga violato.

**Morale:** Security by obscurity è un concetto fallace, la cosa giusta da fare è lasciar fare algoritmi a chi ne capisce (ovvero gli esperti di crittografia).

In pratica piuttosto che fare un algoritmo privato e chiuso autonomamente (il che apre diverse possibilità di fare errori che verranno sfruttati per buttarlo giù), risulta una scelta migliore farlo fare a chi è competente, che non fa errori e lo lascia pubblico.

Altro problema del 2G era l'assenza di mutual authentication, quindi era possibile prendere fingersi una base station fittizia (possibile grazie a software defined radios).

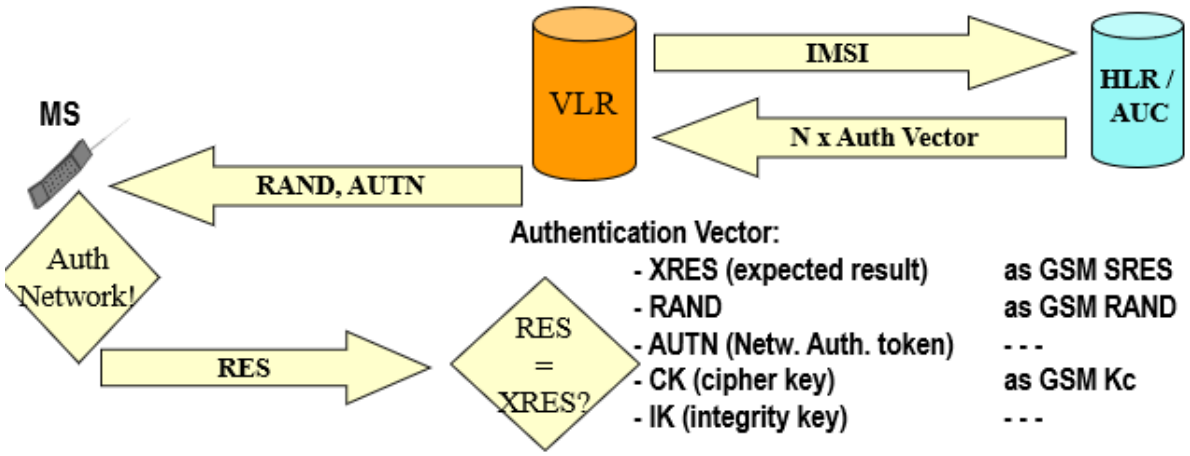
Nel 3G, 4G e 5G infatti verrà usato mutual authentication protocol.

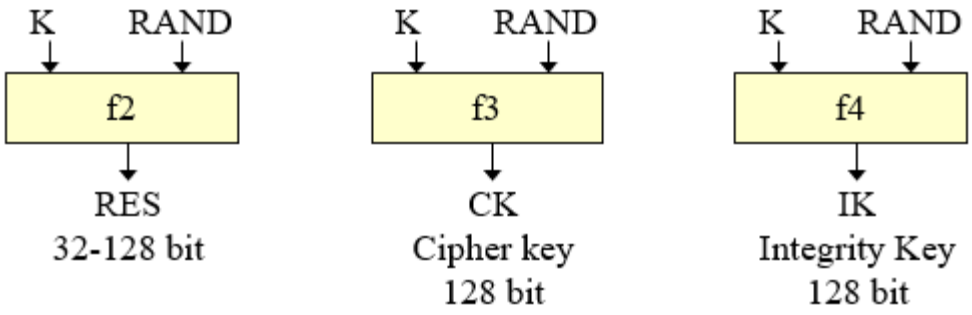
## 5.5. 3G,4G,5G (omettendo in ognuno sempre più dettagli)

Viene usata mutual authentication e gli algoritmi sono pubblici (contrario del security by obscurity).

In questi nuovi sistemi compaiono nuovamente MS, VLR e AUC. Il procedimento è:

- Un MS si connette.
- VLR manda IMSI all'home network.
- HLR manda al VLR delle quintuple che contengono:
  - RAND (challenge)
  - XRES (risultato atteso della CHAP type authentication)
  - CK(cipher key)
  - IK (integrity key)
  - AUTN (Network authentication token - serve a provare che la rete sia autentica per la mutual authentication)
- VLR rimanda quindi a MS RAND, AUTN.
- MS risponde con RES per provare che è autentico una volta che riceve AUTN (ovvero sa che la rete è autentica).






Si noti che VLR manda il token che assicura l'autenticità della rete sulla base di una challenge che lui stesso ha generato (dal punto di vista di MS). Secondo le regole viste finora non si può provare di essere autentici a meno che non sia l'opponent a creare la challenge. In questo caso come è possibile?

### 5.5.1. Esempio: Authentication protocol con 1 single message

La nonce può essere una challenge, una sequenza o un time stamp. Si prende per questo esempio un time stamp, ipotizzando che MS e VLR siano in possesso di un GPS e che il tempo non sia attaccabile. Se MS conosce il tempo, in realtà non serve nel time stamp mandare la nonce poiché è già a conoscenza di che ore sono. Considerando che conosce già il riferimento temporale gli basta la risposta di VLR. VLR quindi provvede a rispondere diretto con H(timestamp, secret). Usando quindi un secure time reference, è possibile effettuare mutual authentication con un solo messaggio.

Se volessimo fare la stessa cosa con una sequenza di numeri:

MS manda una nonce a VLR ed esso risponde con AUTH (K,NONCE). C'è un contatore nella sim chiamato SQN-MS che aumenta ad ogni autenticazione; Cosi quando il VLR manda la sua richiesta di identificazione ad MS, deve anche mettere quanto gli risulta che sia il contatore. In questo modo se MS riceve un contatore minore sa che non si deve fidare.

SQN-MS è sincronizzato con SQN-HE quindi, un altro contatore nella home network; Alcuni attacchi si basano proprio sulla desincronizzazione di questi due.

Metodo di funzionamento: la sim dell'utente conosce SQN MS; quando VLS manda il suo SQN che proviene da HN (incluso nell'AUTN), l'utente deve controllare che SQN=SQN-MS+1 (c'è poi un po' di tolleranza per considerare i casi di messaggi persi);

Se corrisponde, può fidarsi della rete.

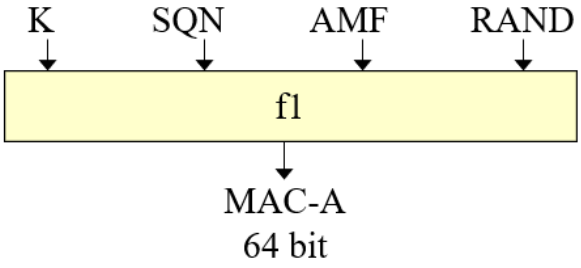
Anche in questo caso quindi con un solo messaggio si assicura network è sicura.

È necessario inoltre introdurre procedure ben strutturate per re-sincronizzare in caso di desincronizzazione HN e MS.

### 5.5.2. Formato dell'AUTN

| SQN<br>Sequence number | AMF<br>Auth & key mgmt field | MAC-A<br>Message Auth code |
|---|---|---|

SQN è il sequence number formato da 48 bit. Attaccati ci sono 16 bit (AMF) che rappresentano quale algoritmo è stato usato e quale finestra di sincronizzazione (signalling info). Infine, ci sono gli ultimi 64 bit (MAC-A) costruiti nel seguente modo:




Si mettono insieme in una funzione di criptaggio f1 SQN (actual proof of knowledge), RAND (challenge), K (chiave) e AMF (info sull'algoritmo).

Tramite gli ultimi 64 bit, MS può testare l'autenticità della network.

### 5.5.3. Sistema sfruttato da 3G e 4G

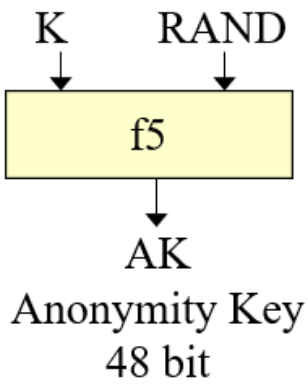
La prima volta che un utente si connette alla VLS, al MS viene inviato un TMSI (temporary MS identifier), dinamicamente allocato alla prima connessione. Alla prossima autenticazione non verrà utilizzato l'IMSI, ma quel TMSI. Quest'ultimo è importante perché cambia ogni volta (risulta quindi non tracciabile).

Siccome vedendo la SQN un eavesdropper potrebbe discriminare e rintracciare l'utente si maschera con una chiave anonima. Si esegue SQN xor AK (anonimity key generata randomicamente).

| SQN xor AK<br>Sequence number | AMF<br>Auth & key mgmt field | MAC-A<br>Message Auth code |
|---|---|---|

L'AK (48 bit) viene generato tramite l'applicazione di una funzione f5 insieme ad un valore randomico. Una volta inviato il pacchetto, per ritrovare la SQN viene rieseguito lo xor con l'AK

Altro passaggio importante è l'esecuzione della funzione f4(SQN,K,RAND,AMF)=MAC-A; verrà controllato che è uguale al MAC A che era stato inviato.



### 5.5.4. Tecnica: generare chiavi a partire da un secret

Come si fa a generare infinite chiavi da un solo secret? "Many secrets in a scalable manner".

Si crea un global secret chiamato S, unico. Ogni volta che arriva un nuovo user crea H(S, a his own unique identifier (e.g. il codice fiscale)). In questo modo si ha la sicurezza che nessun utente nel sistema abbia una stessa password, e analogamente quando ne arriva uno nuovo gli viene affidata la sua Ki senza eseguire controlli su tutte quelle già fornite.

# 6. MESSAGE AUTHENTICATION

Si parte dal presupposto fondamentale che la Confidentiality non garantisce l'integrity:

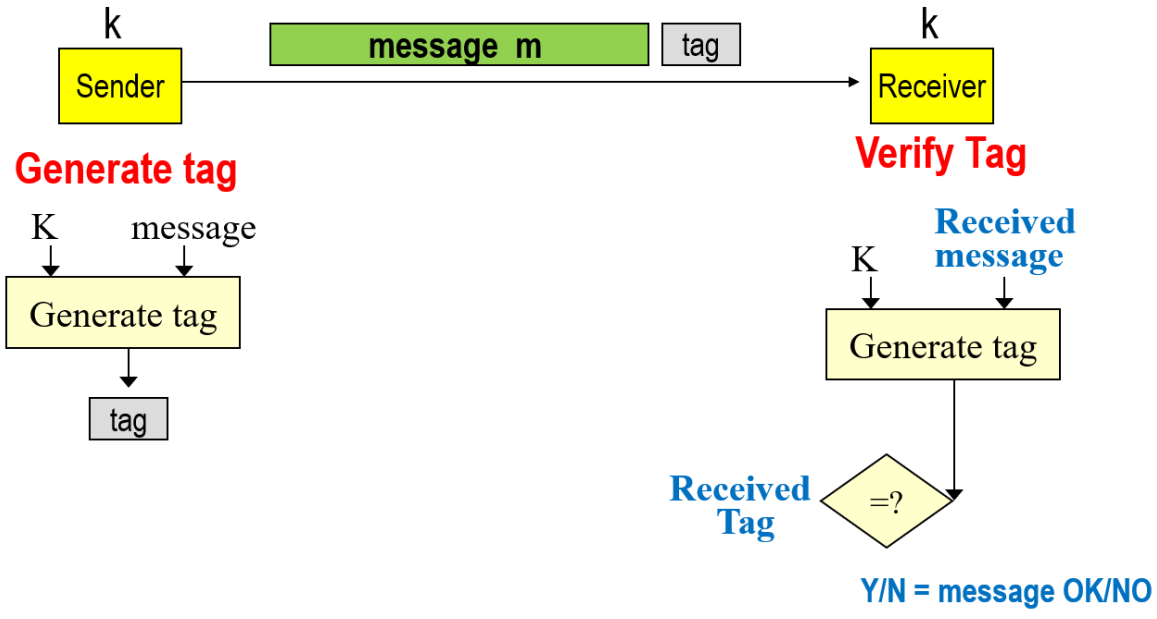
- **Confidentiality** vuol dire message hiding, ovvero la sicurezza che solo il destinatario legge il messaggio.
- **Integrity** vuol dire message authenticity, ovvero la sicurezza che nessuno possa modificare il messaggio.

Il suddetto predicato è vero A MENO CHE non si parla di authenticated encryption tramite AEAD algorithms (***authenticated encryption with associated data***).

## 6.1. Message authentication with symmetric key

Sender e receiver conoscono una chiave k (integrity key). Questa è una chiave diversa da quella per criptare il messaggio già vista.

Si vuole aggiungere un qualcosa al messaggio che serva da check per l'integrità. Si mette un tag dopo il messaggio composto da K e message irreversibilmente uniti in una funzione hash che crea un digest piccolo chiamato tag messo dopo il messaggio stesso. Il receiver è quindi in grado di prendere(che conosce poiché pre-condivisa) e il messaggio per ricrearsi il tag; esso controlla in questo modo che sono uguali.



## 6.2. MAC vs Digital signature

MAC (message authentication code) è una forma più debole di una digital signature. Nella DS nessuno può modificare un messaggio a parte chi l'ha firmato. Con il MAC solo il sender ed il receiver possono farlo; Questo perché entrambi conoscono la chiave.

Nella DS Solo chi firma può modificare integrity messaggio mentre nel MAC chiunque ha la chiave.

Il fatto importante è che la DS garantisce anche non repudiation property , ecco perché è più potente di MAC. La DS richiede diversa crittografia di tipo asimmetrico per essere realizzata.

## 6.3. Message authentication codes

Per l'encryption si era parlato di INDCPA model per la definizione di sicurezza. Per i messaggi si cerca una definizione similare.

Security è unforgeability, ovvero un attacker non deve essere capace di creare un messaggio né di modificarne uno

Formalmente definiamo la sicurezza contro un modello di attacco (Known message attack, chosen message attack, adaptively chosen message attack); dati all'attacker un numero di messaggi passati, vedendone uno nuovo non deve essere in grado di crearsi il nuovo tag. La probabilità di riuscire a forgiare una coppia messaggio/tag autonomamente deve essere negligibile (approssimabile a nulla).

Date queste premesse si esclude l'idea di utilizzare un tag corto, in quanto visto che si vuole lasciare l'attacker a tentativi randomici per azzeccare la coppia, più è lungo il tag e più è tosto da azzeccare.

L'idea di tag minimo per garantire la sicurezza sono 96 bit.

### 6.3.1. Come reagisce il sistema ad un attacco MITM

Un attacker intercetta la coppia messaggio/tag di un sender e vuole modificare m in m'. Se la funzione hash funziona effettivamente bene, l'attacker:

- non è capace di calcolarsi key.
- non deve essere in grado di cambiare il tag tale che tag' = f(k,m') senza conoscere k.
- non deve essere capace nemmeno di cambiare m in m' in modo che f(k,m)==f(k,m'), ovvero anti collision property.

Non è possibile prevenire un man in the middle attack (MITM), ma ci si può facilmente accorgere dell'avvenuto attacco in quanto il tag non corrisponderà. L'attacker ha solo 2 alla meno 100 di probabilità di ricreare il tag giusto dopo aver modificato il messaggio di partenza affinché il receiver non si accorga di nulla.

### 6.3.2. Come reagisce il sistema a replay attacks

Non c'è protezione dai replay attacks, ovvero se vengono mandati due messaggi identici risultano due tag identici.
Esempio: **Message spoofing -** avviene quando il messaggio viene mandato dall'attacker, ma spacciato per un altro mittente senza che il destinatario se ne accorga.

Il replay attack non è più un problema nel caso in cui si è sicuri che i messaggi non si ripetano mai.

**TUTTAVIA:**

Vanno distinti 2 piani differenti, applicazione vera e propria e protocollo (che trasferisce i messaggi dell'applicazione). Bisogna assicurarsi che sia il protocollo a dare protezione e non l'applicazione. Siccome in questo caso (se non si ripetono messaggi) la sicurezza è decisa da dentro l'applicazione, la definizione di sicurezza non va bene. Bisogna risolvere il problema a livello protocollare.

Per riuscirci si usano delle nonce ad ogni pacchetto. Si mette in clear prima del messaggio [nonce, messaggio, tag]. In questo modo il tag cambierebbe, così da risolvere il problema dei messaggi uguali.

Pro e contro di ogni tipo di nonce:

| | |
|---|---|
| Sequence number | Crea il problema del reboot, perché se attacchi e stacchi il sistema si azzera il contatore e si ripetono le nonce. Basta ricordare un solo valore, a differenza dei random number, e il controllo è fatto con una sola operazione |
| Random number | Molto meglio, specialmente se sono true random. Il problema è però che non si può capire se un packet è fresh o meno senza memorizzarli. |
| Time stamp | Il problema di questo è che il tempo deve essere garantito come dato certo |

## 6.4. Message authentication code using hash function

Ingrediente 1: una buona funzione hash (SHA256 è una buona scelta).

Ingrediente 2: includere il segreto nella hash.

A meno che non si possiede una funzione hash function perfetta, ovvero il random oracle, ha importanza la decisione del dove mettere il messaggio nella sequenza di bit che va impasto alla funzione hash.

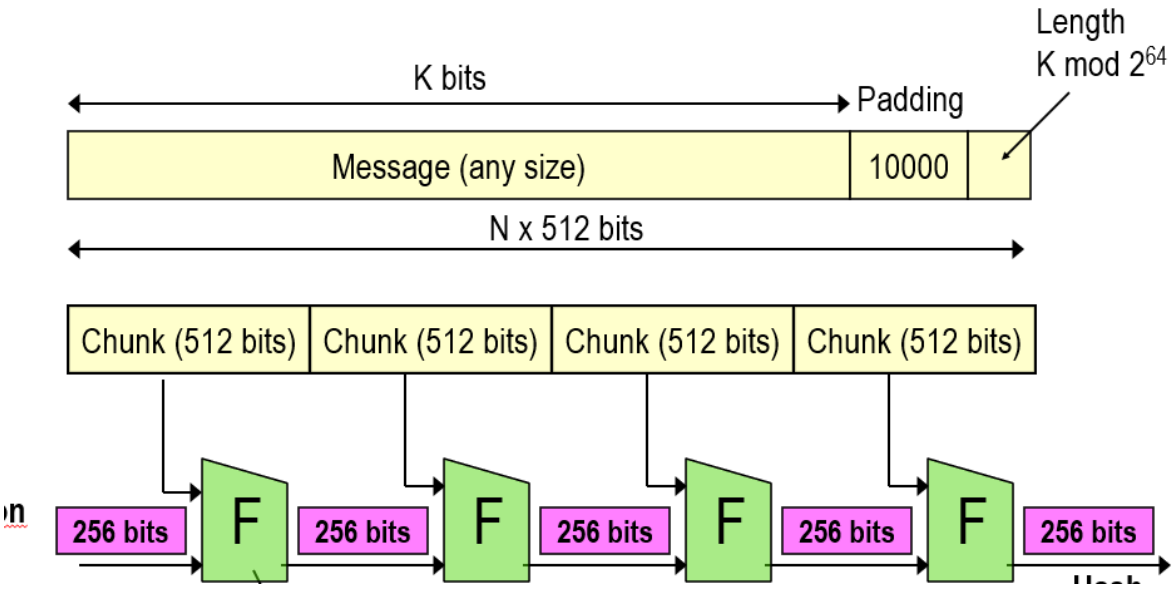
"random oracle" = modello black box tale che dato X input, H(X) è un true random value. Data la proprietà della ripetibilità, a due input uguali corrispondono due output uguali. È impossibile avere una hash function che sia un vero random oracle in quanto è impossibile avere un digest true random value.

### 6.4.1. Come si costruisce una funzione hash:

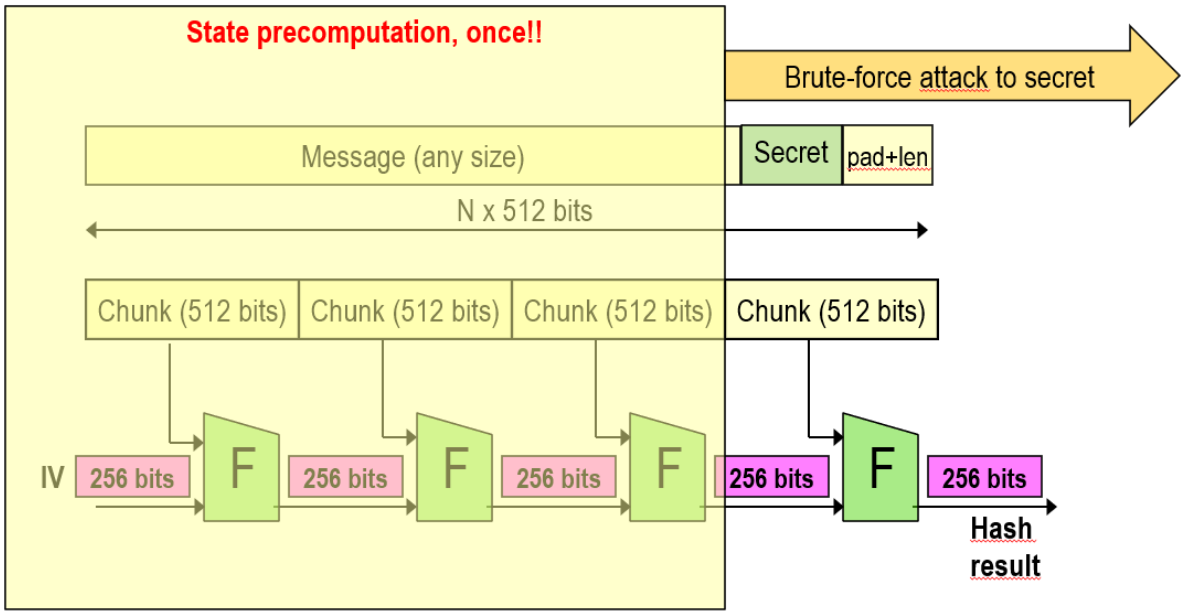
- **Secret come suffisso (sicurezza fiacca)**

Iterative Merkle Damgard construction: è molto difficile trasformare un set di k bit in un fixed size set con una funzione hash, ma i suddetti due crittografi ci sono riusciti.
**SHA256** prende k bit arbitrari, li padda aggiungendo 10000 fino ad avere N x 512 bit; negli ultimi 64 bit viene messo un numero che è la lunghezza del messaggio.

Si ha un messaggio e usando padding e last value si arriva ad avere un multiplo di 512 bit. Essi verranno tagliati in chunks di 512 bit. Viene preso un initialization vector di 256 bit. Questo IV è una quantità fissata e conosciuta (quindi l'IV della hash function non c'entra niente con quello visto nell'encryption). Si unisce questo IV con il primo chunk sommandolo e facendoli diventare 768 bit; Si applica F e si riottengono 256 bit. Poi questi di nuovo vengono riattaccati al secondo chunk e così via iterativamente fino alla fine. F (compression function) è quindi il cuore della funzione specifica usata da Merkle e Damgard nel loro teorema. Se F resiste, allora anche l'iterazione resiste.

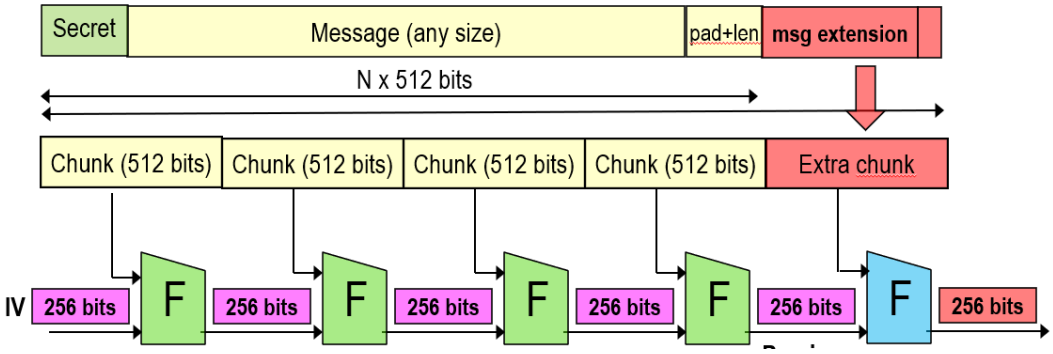



Per un attacker potrebbe essere in grado di ridurre la complessità se si pre-calcolasse tutta la prima parte del messaggio, in modo tale da dover comprimere con F solo l'ultimo blocco. Questo ridurrebbe la complessità dell'attacco in quanto applicherebbe F una sola volta (tutti i chunk del messaggio li avrebbe fatti prima). **Riduce la complessità**, ma resta comunque un brute force attack, quindi dovrebbe poi comunque indovinare la combinazione di bit giusta; tutto ciò costituisce una debolezza del sistema.



- **Secret come prefisso (non sicuro)**

Una volta che l'attacker riesce ad indovinare il pad+len ha violato il sistema in quanto: mettendo il secret all'inizio del messaggio, oltre il pad+len si aggiunge anche un msg extension alla fine fatto da plaintext arbitrario. In questo caso si crea un extra chunk tale che riapplicando F di nuovo si ottiene un messaggio valido non scritto dall'utente.




Dal un punto di vista di sicurezza quindi non funziona.

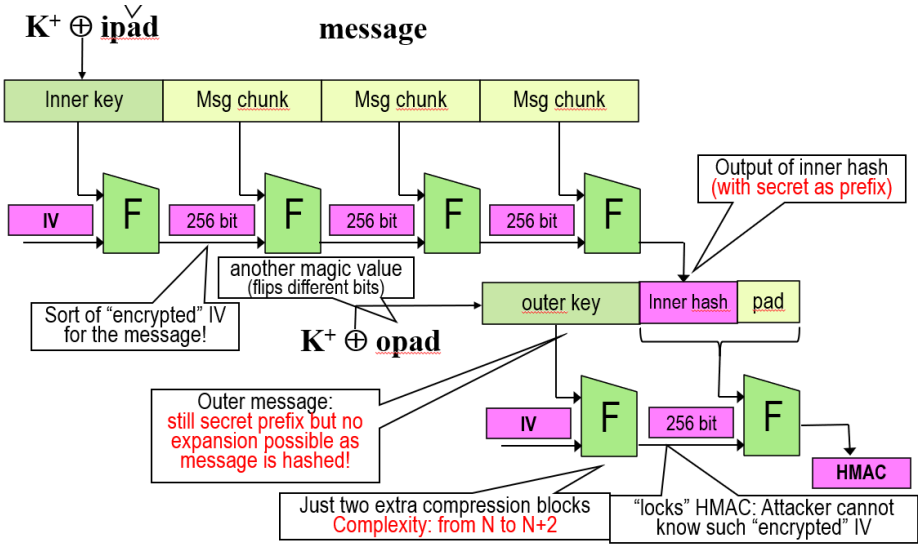
## 6.5. HMAC (Hash based Message Authentication Code)

Costituisce un modo sicuro di aggiungere il secret nell'hash function. Come si costruisce l'HMAC:
Si vuole creare un message authentication code a partire da un message e un secret.
Si applica una funzione hash H scelta tra quelle esistenti (la migliore attualmente è SHA256).

$$HMAC_K(M) = H(K^+ \oplus opad \,||\, H(K^+ \oplus ipad \,||\, M))$$

In parole povere il procedimento è H(secret | | H(secret, msg))

K+ rappresenta la chiave pre-condivisa, estesa alla dimensione di un block size grazie all'acquisizione di una serie di 0 di padding. Si ottengono così 512 bit di chunk che sarà il primo da prendere in considerazione nell'esempio SHA256. Fatto ciò occorre un secondo secret, che tuttavia non si richiede all'utente in quanto scelta pericolosa (l'utente potrebbe usare due secret che lo mettono in pericolo). Per ottenere il primo secret quindi si prende il k+ esteso (come scritto sopra) e si fa lo XOR con opad (outer pad, serie di bit composta da tutti 00110110 ripetuti per arrivare alle dimensioni del chunk). Per ottenere il secondo secret si prende sempre la stessa k+ estesa, ma si esegue lo xor con ipad (inner pad fatto da tutti 01011100 ripetuti fino a diventare 1 chunk di dimensione).



Quindi all'user viene richiesto un solo secret e che viene gestito nel suddetto modo.

Il procedimento quindi consiste nell'attaccare all'inizio del messaggio K+ XOR ipad; si prende un IV conosciuto e si applica F a quest'ultimo e (k XOR ipad) ottenendo 256 bit. Questi bit si mettono in XOR ai msg chunk uno dopo l'altro ogni volta riottenendo 256 bit e riapplicando F.

Alla fine, si ottiene un inner hash (vulnerabile quindi per tutto il discorso precedente all'expansion attack). Si procede poi attaccando (outer key || inner hash || pad) e si esegue la F di IV e outer key ottenendo 256 bit, i quali rifaccio F insieme a inner hash ||pad .

Inner hash richiede N compressioni, outer hash richiede 2 compressioni, quindi la complessità non è aumentata rispetto a prima. In questo modo il sistema risulta sicuro.

In generale, un messaggio è sicuro se soddisfa le **2 proprietà standard delle funzioni hash:**

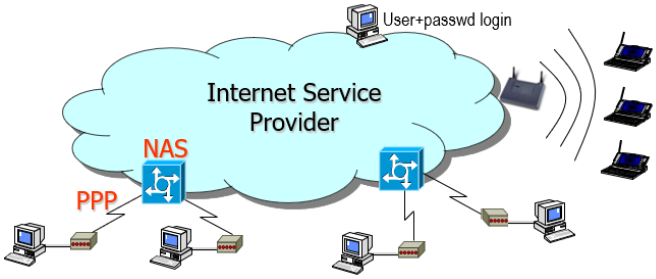
1. Pseudo random output
2. Anti - collision

HMAC è più sicuro della hash function, perché se la hash function è rotta in termini di collision, HMAC construction è ancora attiva (si può ancora usare HMAC con MD5 o SHA-1 anche se sono entrambi danneggiati poiché ci sono algoritmi per calcolare la collisione).

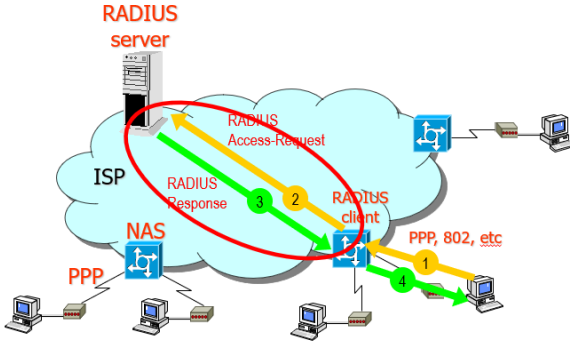
# 7. GESTIONE DELL'ACCESSO REMOTO: RADIUS (Remote Authentication Dial In User Service)

Per gestire reti su larga scala (network larghe e eterogenee) servono tecnologie di accesso eterogenee e distribuite; Esse servono per la gestione di:

- più NAS (client), più servizi di accesso
- non solo autenticazione, ma anche assegnazione della configurazione specifica nel servizio




La soluzione ottima è ottenuta gestendo il DB per il singolo utente



### Funzionamento:

1. Utente invia gli attributi dell'autenticazione al NAS: **non c'è RADIUS qua**
2. Il NAS inoltra la richiesta di accesso al server RADIUS
3. Il server RADIUS risponde OK, NO, Challenge (per alcuni AUTH)
4. Notifica NAS all'utente (se può entrare o no)**: non c'è RADIUS qua**

**RADIUS gestisce il modo in cui i messaggi da client a server vanno e vengono**

## 7.1. RADIUS: AAA protocol

Fornisce funzionalità AAA centralizzate

- **Autenticazione**: *sei davvero chi dici di essere??*
  - Verificare che tu sei veramente chi dici di essere
- **Autorizzazione**: *hai i permessi per accedere a un servizio?*
  - Posso essere completamente autenticato, ma se non pago il servizio, esso non mi darà l'accesso (se non pago SKY non posso accedere al servizio!)
  - Possono essere disgiunte, infatti posso avere autenticazione senza autorizzazione, mentre se ho autorizzazione senza autenticazione il service è privacy preserving (tutela della privacy) (RADIUS supporta entrambe)
- **Accounting** (contabilità): *Cosa stai facendo/usando/pagando attualmente?*
  - Byte trasmessi, fatturazione (billing) ….

## 7.2. RADIUS: protocollo client-server

- Client RADIUS: il NAS (è il NAS che inizia il servizio)
  - Non confonderlo con l'utente finale !!

- Protocollo client-server
  - Basato su UDP / IP
  - Porta server 1812 (porta client temporanea, come al solito in C / S)

- (Logicamente) centralizzato
  - 1 server primario (0+ server secondari - replicati)
    - Gestione dei server replicati è dipendente dall'implementazione
  - Il server può a turno agire come un proxy

### 7.2.1. Proxy operation

- Remote RADIUS server (or other AAA e.g. Cisco TACACS+)
  - Spesso da ISP diversi: l'utilizzo tipico del proxy è il roaming

- RADIUS server
  - Può essere trasparente o non trasparente (ad es. Modificare la risposta per adattarla alle politiche locali)

## 7.3. ARCHITETTURA RADIUS

- Applicazione server RADIUS
- Database utente registrato
  - Per ogni voce (nome_utente), contiene (almeno):
    - Informazioni di autenticazione (segreti)
    - Metodo di autenticazione
    - Attributi di autorizzazione (profilo di accesso per ogni utente)
- Database dei clienti
  - Clienti autorizzati a comunicare con il server
  - ATTENZIONE: NON confondere i client (RADIUS) con gli utenti (finali)
- Database contabile
  - Quando RADIUS utilizzato per la contabilità
  - Utilizzato frequentemente solo per l'autenticazione

## 7.4. RADIUS CARATTERISTICHE DI SICUREZZA

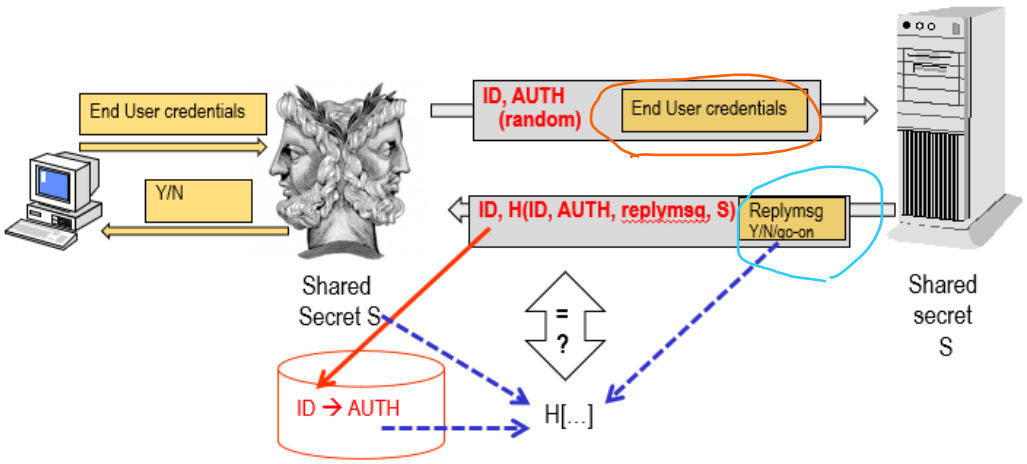
- **Risposta autenticata per pacchetto**
  - Basata su segreto condiviso
  - Nessuna trasmissione di segreti (simile a CHAP)
  - Però:
    - Solo la risposta è autenticata!
    - Basato su hash, non su HMAC
    - Funzione hash molto specifica, MD5
    - Chiave condivisa spesso a bassa entropia
- **Trasmissione crittografata della password utente**
  - Meccanismo personalizzato
  - Stessa chiave condivisa utilizzata per l'autenticazione!

### 7.4.1. RADIUS authenticated reply: concept

(NAS) --> da una parte parla con end user (il quale gli invia le sue credenziali), dall'altra invece con il server. L'user manda le credenziali al NAS; quest'ultimo crea un pacchetto di end user credentials (RADIUS packet) con ID e AUTH (random). A questo punto lo manda al server, il quale risponde con ID,H(ID,AUTH,replymsg, S) ed il messaggio di risposta.
L'ID field serve per riconoscere tramite un numero univoco il pacchetto in questione.

Una sorta di challenge-response:

- challenge è l'autenticatore della richiesta
- risposta include anche (quindi convalida) il messaggio di risposta




Il message replaymessage è l'attuale accept/reject, non contiene la parte sopra (end user credentials), il quale è una parte non protetta!

**Authentication field**

- In Access Request (C --> S)
  - 16 byte generati casualmente
    - imprevedibile e unico per evitare l'attacco replay
- In Response packets (S → C Accept/Reject/Challenge))
  - One-way MD5 hash di:
    - ID della richiesta
    - Autenticatore della richiesta
    - Segreto condiviso
    - Informazioni sulla risposta del pacchetto
      - ◊ Response packet è firmato! In caso contrario la risposta del server potrebbe essere falsificata
- MD5(Code | ID | Length | RequestAuth | Attributes | Secret)

**Access-Reject**

Due ragioni principali:

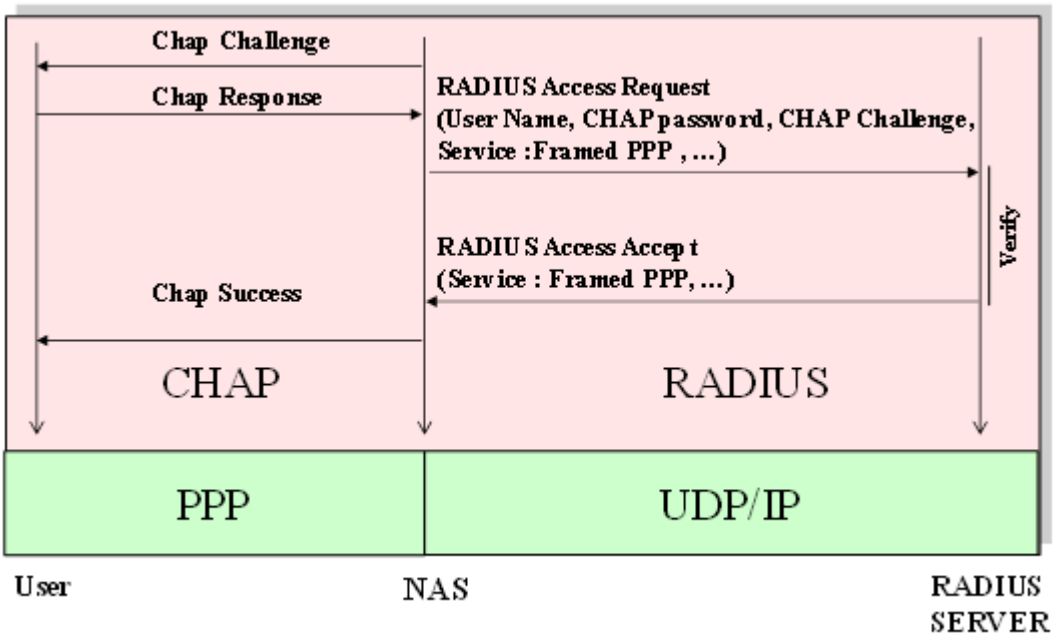
- Autenticazione fallita
- 1+ attributi nella richiesta non sono stati considerati accettabili (autorizzazione non riuscita)

**Access-Challenge**

Usato ogni volta che il server desidera/necessita che 'utente invii un'ulteriore risposta

## PPP CHAP supportato con RADIUS

- CHAP challenge generata localmente dal NAS
  - Non è necessario conoscere la password dell'utente per fare ciò
- CHAP challenge + response inviata al RADIUS server
  - RADIUS server recupera la password utente dal database, calcola e confronta la risposta CHAP




Quindi si noti che per quanto riguarda l'authentication field all'interno di RADIUS, la nonce presente nella richiesta non autenticata è riutilizzata nella response (che serve ad autenticare il messaggio).

## 7.4.2. Password encryption

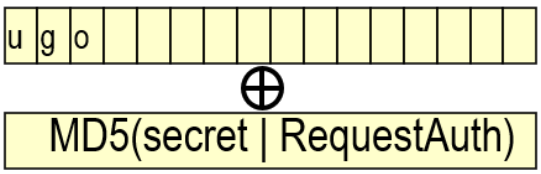
È la seconda tecnica di protezione usata in RADIUS. Lo user manda al NAS username e password. Il NAS lo manda allora al server, ma siccome la rete fra il NAS ed il RADIUS server non è sicura, bisogna criptare il messaggio.

Metodo: si prende la password e si padda fino a 16 byte (blocco). Si mette allora il password block così ottenuto insieme alla nonce dentro l'hash MD5; alla fine si fa lo xor del risultato con la password di nuovo. il risultato di questa operazione viene inserito all'interno del campo attributo, per essere trasportato dal messaggio RADIUS.

Quando la password utente sia più lunga di 16 bytes, viene segmentata in blocchi di 16 bytes ciascuno: P1,P2,..,Pn, dove Pn può essere riempito con zeri per formare un blocco di 16 bytes.

Vengono quindi eseguite le seguenti operazioni per nascondere la password:

B1= MD5 (SS | RA) → C1 = P1 XOR B1
B2= MD5 (SS | C1) → C2 = P2 XOR B2
. .
. .
. .
Bn= MD5 (SS | Cn - 1) → Cn = Pn XOR Bn




Dove SS = segreto condiviso, RA = request authenticator.

I valori C sono quelli che vengono inseriti nell'attributo User-Password (dopo la concatenazione di C1, C2, C3, ..., Cn).

## 7.5. RADIUS debolezze di sicurezza

RADIUS è sensibile a message sniffing, perché non c'è confidentiality, ed è vulnerabile a message modification perché non c'è autenticazione per le richieste di accesso (ognuno può modificarne e crearne una nuova).

L'autenticazione per le richieste di accesso può essere ottenuta tramite:

- **Message authenticator**
  Per fornire la protezione dell'integrità per i messaggi di richiesta di accesso, tra gli attributi del request packet del messaggio viene aggiunto un tag detto "Authenticator" (è un attributo).

  Quando si protegge il messaggio di richiesta di accesso con Message Authenticator, si calcola un hash dell'intero pacchetto usando il segreto condiviso che condivide con il server RADIUS (HMAC_MD5(whole packet) = HMAC-MD5(type | ID | len | RequestAuth | attributes)).

  Il tag verrà posto alla fine del messaggio per l'autenticazione.

- **EAP (Extensible Authentication Protocol)**
  Si tratta di un approccio arbitrario di autenticazione, ovvero di tecniche aggiuntive messe per la sicurezza dei sistemi a cura di chi li realizza.

**Il RADIUS è vulnerabile a dictionary attack al secret:**
Le password scelte dagli utenti sono spesso low entropy, ovvero parole comuni o intuibili; altro problema è che spesso la gente usa la stessa password per tanti siti.
Per risolvere il suddetto problema si prende un qualcosa di unico, come per esempio un dato fisico del device, e si fa in modo che il secret sia HMAC(secret, dato unico segreto); si tratta di una soluzione scalabile per la request authentication.

Come contrastare il dictionary attack?

Un hacker prende l'id di un utente. Si presenta al NAS scegliendo una password a caso e ascolta la richiesta capendo quale random è stato generato. Ascoltando la risposta acquisirà l'MD5 associato. Conoscendo la coppia potrà entrare nel sistema, perché rifacendo xor con la sua password, gli resta MD5 della coppia; Conoscendo la challenge che stava dentro l'MD5, l'unica incognita che gli resta è il secret, che potrà indovinare con un brute force attack (trattandosi di 16 bytes).

Per evitare ciò serve un request authenticator unico. Per realizzarlo servono 16 bytes random. Ci sono 3 modi: attaccare fra loro diverse unità da 4 bytes pseudo-random, prendere 1 unità e paddarla fino a 16, oppure fare MD5 dei 4 byte.

Valutando il ciclo di valori pseudo-random prodotti che mi permette di non avere ripetizioni:

Il primo è la peggior soluzione perché generare 4 sequenze per volta fa in modo che si ricrea una sequenza già vista quattro volte più velocemente.

Il secondo ed il terzo hanno la stessa probabilità di ripetizione, perché si tratta sempre di 4 bit casuali che sono soggetti al paradosso del compleanno. Sono migliori del primo in quanto usano 1 sequenza anziché 4.

### Caratteristiche di un PRNG

Per realizzare un buon pseudo-random number generator si rivolge l'attenzione a predictability e period, due proprietà importanti. Bisogna fare in modo che non si riesca a prevedere una successione di risultati prevedibili e che non si ripeta un risultato.

Un PRNG conosciuto è il linear R(n+1) = (aR(n)+b)mod(n).

### 7.5.1. Perché RADIUS non è sicuro:

- Se un hacker può monitorare dei tentativi di accesso, può costruire un database di coppie auth request nonce – access accept packet (tenta un replay attack). Sarà a quel punto in grado di entrare nella rete quando il NAS genererà un access request nonce già usato. L'hacker potrà allora rispondere al NAS fingendosi server usando la risposta che aveva collezionato nel database. In breve, l'hacker quindi potrà autenticarsi da solo in quanto già a conoscenza della risposta.
- Altro modo è costruire una tabella dove ad ogni user id è associata l'informazione password xor MD5(secret,nonce), permette, alla ripetizione di una nonce, di fare lo xor fra quelle due suddette informazioni al fine di ottenere password1 xor password2. A partire da due password in xor è spesso possibile tramite un'opportuna analisi, risalire ad entrambe le password.

### Cosa insegna il RADIUS:

Un application protocol non deve includere security. Si può provare a sviluppare un proprio sistema di security, ma è meglio non farlo. Bisogna concentrarsi sull'applicazione invece, la sicurezza la può realizzare un sistema esterno apposito.

### 7.5.2. Problemi di RADIUS

I problemi quindi furono delle limitazioni funzionali:

- **Scalability**: RADIUS usava UDP, protocollo non affidabile per migliaia di utenti (conseguentemente RADIUS ha infatti problemi di packet loss). Pensando quindi ad un sistema esteso, in caso di crash di un server, a causa di UDP ci sarebbe packet loss su larga scala, scelta quindi sbagliata. TCP sarebbe molto meglio.
- **Extendibility**: (type | len | value) per il tag non erano sufficienti per l'autenticazione. Questa idea adottata da RADIUS di utilizzare i suddetti byte negli attributi non fu abbastanza per affrontare le nuove tecnologie. Di fatto quindi si creò una limitazione per gli attributi utilizzabili.
- **Interoperability**: L'utilizzo di tanti server (che adottano soluzioni differenti, realizzati da vendor differenti) crea consistenti problemi per l'interoperabilità.

**IETF (Internet engineering task force)**

Si tratta di un'organizzazione che realizzò diverse tecnologie quali Diameter, RADext.

## 7.6. Diameter

Protocollo più potente di RADIUS, più potente e al passo con i tempi che sfruttava nuove tecnologie. È una sorta di object-oriented protocol (definizione di Bianchi). C'è una classe base del protocollo e, grazie al concetto di ereditarietà, se ne ottengono classi derivate.

- Diameter non è un AAA protocol come il RADIUS, ma è un messaging protocol (application layer).
- AAA transport profile comunica come deve funzionare il trasporto dei messaggi (basato su TCP).
- A seconda della necessità vengono create tutte specifiche derivazioni del protocollo base Diameter.
- RADIUS di fatto costituisce una derivazione del generale DIAMETER base protocol.

### 7.6.1. TCP problems

1. **Strict order delivery e HOL**
2. **Mancanza di multi homing**

Quando richiesta, il NAS deve creare una connessione affidabile con l'AAA server. Per garantirla, basta usare una connessione di tipo TCP. Una volta instaurata la connessione di tale tipo, viene utilizzata per tutti i successivi packet del primo. L'AAA server può usare più threads, così che possano essere presi in carico diversi messaggi per volta. Se ora un thread si ferma per un problema, un guasto, TCP diventa svantaggioso in quanto fra le sue proprietà c'è **strict order delivery**, quindi fa in modo che non si possa lavorare sui pacchetti successivi se non ho lavorato i precedenti. Sarà necessaria una nuova ricezione a partire dal messaggio perso. Il suddetto problema prende il nome di HOL (head of the line blocking effect), e si presenta in ogni stream unico TCP per più pacchetti.

Il secondo problema di TCP è il seguente: preso un NAS, se la network fallisce si può garantire service continuity grazie a delle connessioni di ricambio ulteriori. Il problema è che l'ip è differente, e siccome TCP si basa sulle socket che richiedono di pre-specificare l'ip, bisognerebbe creare una nuova socket ogni volta. Quindi se una connessione fallisce, bisogna ricrearne un'altra. Servirebbe multi homing, ovvero la possibilità di avere più connessioni insieme in modo tale che se una fallisce se ne usa un'altra.

### 7.6.2. SCTP stream control transport protocol

Si tratta di un protocollo che permetterebbe di avere multi homing, ma non viene utilizzato per il Chicken and egg problem ed Internet ossification.
Nel 1980 si utilizzava memorizzare "l'intelligenza" (ovvero i dati) degli utenti sui dispositivi, l'edge della network era quindi povero di dati. L'idea fu quella di portare l'intelligenza all'edge. NAT (network address translator) fu necessario per risolvere il problema degli ip corti, al fine di connettere più device di quanti si poteva. NAT fu il primo intermediario che poteva modificare il pacchetto (da cui nacque il discorso sulla sicurezza relativo al firewall e altre cose chiamate **middle boxes**). Quindi perché non si usa l'SCTP?

Per utilizzarlo bisogna mettere a fuoco il fatto che la connessione fra due server passa per diverse middle boxes (intermediari). Se si utilizza un protocollo sconosciuto a quelle middle boxes, esse bloccano il traffico. Affinché il protocollo prenda piede serve che le middle boxes supportino l'SCTP, ma non lo fanno in quanto nessuno lo usa.

In sostanza, nessuno può usare l'STCP perché le middle boxes lo bloccano, e le middle boxes hanno bisogno che il protocollo sia usato per permetterlo. Questo prende il nome di Chicken and egg problem. Quindi si parla di internet ossification poiché la crescita di internet fu bloccata da questa cosa.

### 7.6.3. Diameter improvements rispetto al RADIUS

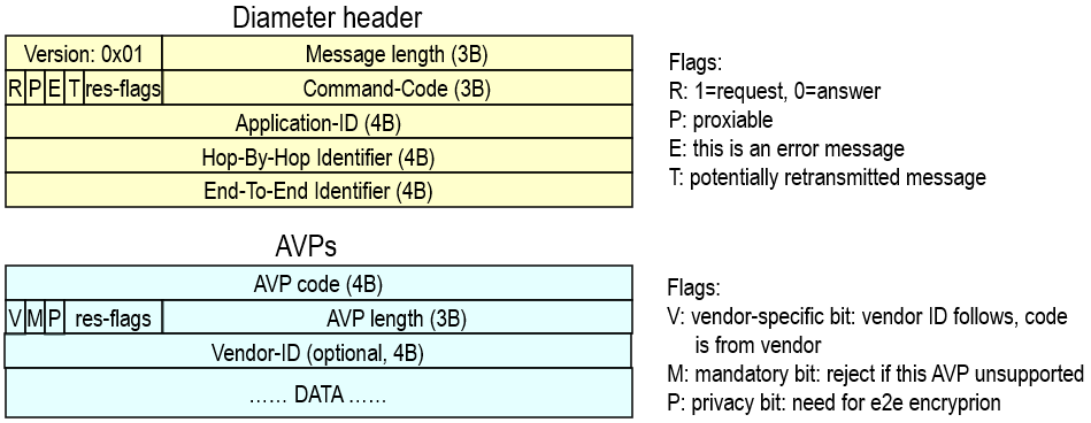
- **Reliable transport** (affidabilità essenziale per discorsi monetari)
- **Standardized error and fail-over control** (all'application level c'è controllo degli errori e una serie di funzionalità di ritrasmissione, gestita tramite l'invio di pacchetti che controllano se un device ha fallito)
- **Peer discovery, configuration, capability detection**
- **Supporto della comunicazione server client**
- **Gestione delle entità intermedie**
- **Estensione dei limiti funzionali del RADIUS**
    - (Code | id | len | req-auth | attributes) era la struttura del pacchetto di RADIUS. In Diameter venne adottato l'utilizzo di un header in quanto non era chiaro l'id assegnato ai pacchetti per via dei limiti funzionali legati alla tecnologia che era cambiata. Ad esempio, ora era possibile comunicare fra più server, fatto che cambiare l'id dei pacchetti ad ogni invio (dato non attendibile). Diameter modificò il tag in un header seguito da uno o più AVPs (Attribute Value Pair).

**Header: (R | P | E | T)**

- R: specifica se si tratta di request o answer
- P: indica se è proxable
- E: indica la presenza di errore
- T: potentially retransmitting message, fa capire se il packet è uno che sto ritrasmettendo e non è nuovo

**AVPs (seguono l'header): (V | M | P)**

- V: indica la presenza o meno dei dati vendor specific
- M: mandatory bit che serve in caso di diverse versioni fra NAS e server, che garantisce che un pacchetto non venga perso. Il server può chiedere di ritrasmettere perché non ha capito, **risolve interoperability**)
- P: dice se il messaggio è criptato o no




## 7.6.4. Le 3 nuove standardizzazioni del Diameter:

- Rely agent
- Proxy agents
- Redirect agents

Contenevano i routing table per connettersi fra i diversi server.

Un rely agent accetta le richieste e le indirizza al server giusto basandosi sulle info contenute nel messaggio (vede per esempio il dominio dell'e-mail e sa dove andare); proxy agent è la stessa cosa ma è anche in grado di modificare il messaggio.

Preso un sistema di server, se uno crolla o cambia qualcosa, comunica il cambiamento a tutti gli altri. Anche quando qualcuno si unisce comunica con tutti. Questo discorso costituisce un incubo per la gestione, perché se ci sono migliaia di server è impossibile gestirli senza i rely agents e proxy agents. Se si lasciano a loro le routing table se ne occupano loro.

L'idea è quella di creare un controller centralizzato che non fa niente a parte mantenere le routing table. Ogni volta il server (che ha il pacchetto data da inoltrare) chiede al controller dove sta il server a cui mandarlo. Cosi tutto il sistema risulta meno complicato in quanto le routing table sono centralizzate sul controller e tutti i server interrogano quello.

Il redirect agent non gestisce i dati ma li controlla solo. Provvede a routing decision sulle richieste in arrivo ma non manda effettivamente la richiesta, manda solo il redirect. Utile quando le routing decision sono centralizzate. Il rely agent parla con il redirect agent.