Contents

1 Introduzione-warm up example

Il problema spesso è che una buona crittografia è applicata male alla risoluzione di un problema.esempio: paper che discute di una tecnica di sicurezza ed in cui viene matematicamente provata la sicurezza.
Basata sul meccanismo del One Time Pad: ho il mio plain text e voglio criptarlo in modo che non si capisca cosa ci sia scritto. Genero una sequenza random di bit, di lunghezza pari alla lunghezza del testo. Una volta ottenuta la chiave, computo lo XOR fra la chiave ed il plaintext ed ottengo il mio ciphertext.
Il seguente meccanismo è il migliore possibile per fare enrcyption.
Per decriptare applico il procedimento inverso, facendo sempre l'XOR, infatti: $b \oplus 1 = \overline{b} \oplus 1 = b$.
Devo però fare delle assunzioni:

1. Per ogni nuovo messaggi, devo usare una diversa chiave. Questo perché, se ripetessi la chiave avrei il peggior meccanismo di enrcyption: se ho due messaggi M_1 ed M_2, ed ottengo $C_1 = M_1 \oplus K$ e $C_2 = M_2 \oplus K$, ora facendo $C_1 \oplus C_2 = (M_1 \oplus K) \oplus (M_2 \oplus K) = M_1 \oplus M_2$ (in quanto $K \oplus K$ si elide).Conoscendo uno dei due messaggi posso ricavare l'altro.

2. La chiave deve essere lunga quanto il plaintext

3. La chiave deve essere veramente random, e non pseudorandom.

L'algoritmo introdotto sopra è chiamato Vernam Cipher, ed è il miglior meccanismo di enrcyption possibile.
Il problema nel metterlo in pratica è che la chiave deve essere nota sia a chi produce il messaggio, sia a chi lo riceve, quindi va trasmessa su un canale sicuro. Se le dimensioni della chiave cominciano a diventare considerevoli, ad esempio per

2GB di plaintext devo avere 2GB di chiave, il costo d'invio al receiver diventa oneroso.
Quando si parla di sicurezza, non bisogna chiedersi come rendere il sistema sicuro, ma da cosa devo difendermi, di cosa è capace l'attaccante.
OTP protegge la confidenzialità, ma non garantisce l'integrità.
Le 3 proprietà che posso/voglio garantire sono:

- Confidenzialità: proteggo i dati da persone esterne,che non possono leggere il contenuto senza una chiave
- Integrità: voglio che i miei dati rimangano inalterati
- Aviability: mi proteggo, ad esempio da DDoS

Nel caso di OTP, l'integrità non è garantita: se un avversario prende il mio messaggio e lo cambia,ad esempio flippando alcuni byte (Man in the middle attack) non riesco a rendermene conto; l'avversario ha agito sul testo cifrato, senza interessarsi del,contenuto.

2 Encryption

Servizio di sicurezza che vuole proteggere la confidenzialità dei dati, non protegge però l'integrità. Si parte dal plaintext $\longrightarrow$ encryption $\longrightarrow$ cipher text $\longrightarrow$ invio $\longrightarrow$ decryption $\longrightarrow$ plaintext.
Servono delle chiavi per potere de/criptare, per ora mi concentro sul meccanismo della symmetric key: sia sender che receiver usano la stessa chiave. Il cipher sarà il mio algoritmo per criptare e decriptare:
C = ENC(K,P) D = DEC(K,C).

2.1 Esempi storici di cipher

Un primo esempio può essere quello di sostituire le lettere del plaintext con altre lettere, in maniera reversibile ovvero se a $\longrightarrow$ b, non posso avere c $\longrightarrow$ b.
Posso decriptare in maniera veloce? Vedo la frequenza delle lettere di una lingua, ad esempio l'italiano, e saprò quali lettere compaiono più spesso in un plaintext, inoltre posso avere delle parole con delle ripetizioni interne.
Procedo per tentativi, nel momento in cui deduco una lettera, la associo ad una più o meno probabile.
Posso inoltre ricavare la chiave usata per criptare.
Il metodo è storico (me pare addirittura lo usavano i romani sotto Cesare), quindi fortemente sconsigliato.

2.2 RFID mutual authentication

Vernam cipher è il miglior meccanismo, ma ha delle forti implicazioni. Considero una situazione reale:
ho un TAG, ed un reader presso cui devo autenticarmi. Il TAG ha lo scopo

di provare al reader che l'utente è davvero reale, ma anche il reader dovrebbe dimostrare di essere sicuro; voglio quindi che l'autenticazione sia mutua.
Ho un segreto S, scritto ad esempio in una mia carta d'autenticazione e quando mi approccio al reader mi devo identificare. Ho due probelmi:

- La trasmissione avviene su un canale wireless, se poi la trasmissione è in chiaro un attacker può captare e rubare S
- Se il reader è falso, ora conosce il mio segreto S

Vorrei poter autenticarmi senza mostrare il segreto S esplicitamente, uso uno schema:
il TAG ha un segreto S,statico, ed una chiave temporanea k. Invece di trasmettere S, invio $k \oplus S$ al reader, che avrà un database in cui ha il segreto salvato e la chiave k. Il reader riesegue quindi lo XOR e vede se il risultato coincide con quello che gli ho inviato io. La prossima chiave sarà generata dal reader a partire da k, quindi con un meccanismo pseudorandom. Provo ad usare un former analyzer, che mi garantisce che il sistema è sicuro (software che prova a crackare il meccanismo di encryption). Sono realmente sicuro?
Ho l'operazione $k \oplus S$, k è pseudorandom e posso avere due situazioni:

1. S è la chiave: sto violando la proprietà 1, in quanto riuso S più volte per messaggi diversi
2. k è la chiave: se faccio $(S \oplus k_i) \oplus (S \oplus k_{i+1}) = k_i \oplus k_{i+1}$. Non ho violato il sistema, ma ho la combinazione delle due chiavi, che sono pseudorandom: ho $x \oplus f(x)$ (PNRG(x)), x dipende da f(x), quindi posso fare un ciclo fino al valore massimo, controllo se $z_i = x_i \oplus \text{PNRG}(x_i)$, alla fine ricaverò k_i.

Il microfono non può essere risolto in alcun modo, le assunzioni erano errate, quindi:

- Former analyzers non sono una certezza, bisogna comunque verificare che l'assunzione è corretta
- Random e pseudorandom sono completamente diversi

2.3 Sicurezza di un cipher

Un cipher è sicuro quando:

- protegge la confidenzialità
- nasconde i messaggi
- non può essere violato

Ma questa definizione è una supercazzola (serio, così ha detto il prof a lezione e così scrivo io negli appunti), e anche altre definizioni sono brutte e sbagliate. Un cipher può essere sicuro per un determinato attacco che vuole svelare il

bit segreto	bit random	XOR	probabilità
0	0	0	$\frac{p}{2}$
0	1	1	$\frac{p}{2}$
1	0	1	$\frac{(1-p)}{2}$
1	1	0	$\frac{(1-p)}{2}$

contenuto, ma non sicuro per un altro che vuole vedere solo parte delle coppie plaintext-ciphertext.
Ad esempio un chosen plaintext attack permette di vedere sia plaintext che ciphertext, voglio essere robusto quantomeno a questo tipo di attacco.
Definizione di semantically secure o IND-CPA, ovvero Indistinguishability Under Chosen Plaintext Attack.
esempio: ho due messaggi, M_0 ed M_1, suppongo di poter criptare solo uno dei due.
Permetto all attacker di mandarmi i due messaggi ed io scelgo a caso quale dei due criptare con un coinflip. Mando indietro il messaggio cifrato all'attaccante: in condizioni normali l'attacante può facilmente decriptare il messaggio, se usassi un cipher non semantically secure, ma ora entra in gioco IND-CPA $\Rightarrow$ l'attaccante ha una probabilità del 50% di ottenere il messaggio corretto, ovvero deve scegliere a caso fra i due.I sistema sarà semantically secure se l'avversario non può risolvere questa situazione: ha a disposizione un oracolo, che gli fornisce l'encryption dei due messagi, quindi se uso un meccanismo di encryption sostitutivo (vedi esempio de Giulio Cesare) $\Rightarrow$ GAME OVER. Ora uso una chiave random (esempio Vernam Cipher): allo stesso plaintext corrispondono ciphertext diversi, quindi l'oracolo non può fornire il risultato esatto all'attaccante. L'unico modo che ha per vincere è di tirare ad indovinare, quindi con un coinflip.
L'encryption deve essere random, perché se una sotto stringa si ripete non deve corrispondere allo stesso ciphertext. Lo XOR è random:
bit segreto $\oplus$ bit random
bit segreto: 0 = p, 1 = 1-p
bit random: $0 = \frac{1}{2}$, $1 = \frac{1}{2}$
Avrò quindi:
Quindi il Vernam ciper è perfettamente random: l'avversario vede solo il cipher text, quindi può indovinare 0 o 1 con probabilità: $\frac{p}{2} + \frac{(1-p)}{2} = \frac{1}{2}$. Vernam cipher è però teorico e nella pratica si usano altri cipher, divisi in categorie:

- stream cipher: un mimic di Vernam cipher, usa un algoritmo pseudorandom usando lo XOR, il più famoso era RC4, oggi si usano Salsa20 e ChaCha20.
- Block cipher: il più usato è AES, usano una tecnica diversa
- Block cipher in stream mode: AES-CTR, il block cipher genera una chiave pseudorandom e poi usa uno stream cipher.

2.4 Stream cipher

L'obbiettivo è quello di approssimare One Time Pad: invece di usare una chiave random, uso una chiave di 128 bit come seed per uno stream di bit pseudorandom, che sarà il keystream.
Usa poi lo XOR, la chiave è più corta e viene incrementata con il keystream: l'algoritmo pseudorandom è progettato ad hoc, non è il classico pseudorandom. La differenza cruciale con OTP è che la chiave è generata a partire da una chiave k piccola, quindi posso trasmettere k al receiver facilmente. Ma se k è sempre la stessa ho un problema, ovvero encrypto sempre con la stessa key di base. Se una sottostringa si ripete, avrò ciphertext diverso (la periodicità del sistema pseudorandom deve essere molto lunga), ma per lo stesso messaggio ho lo stesso ciphertext, in quanto l'algoritmo pseudorandom deterministico. Vorrei comunicare la chiave una volta per tutte senza doverla cambiare (come avviene in Wi Fi access point), ho una chiave k piccola ed un keystream lungo, ma non sono IND-CPA.

2.4.1 Initialization vector

Ho un plaintext che voglio cifrare, mando un messaggio alla mia NIC in modo che lo encrypti con un algoritmo di tipo stream cipher. La NIC ha una chiave k a lungo termine e quando riceve il messaggio genera una quantità dinamica,che è l'initialization vector (IV); questa quantità può essere truly random. Il seed sarà generato giustapponendo la chiave k all'IV, che mi fornirà il keystream, ovviamente l'IV deve essere diverso per ogni messaggio. Come comunico all'altro end l'IV? Lo mando in chiaro con il messaggio, se lo strem cipher è buono non posso determinare il messaggio a partire dall'initialization vector. Ora il receiver può riprodurre il keystream: fa lo XOR e decripta il messaggio ricevuto; l'ipotesi fondamentale è che il PRNG sia buono.
Ho la prova di essere semantically secure se l'IV non si ripete.

2.4.2 Case study: WEP 802.11

Wired Equivalent Privacy, standardizzato nel 1997-1999 dagli stessi progettisti di Wi-Fi. Aveva 3 obbiettivi:

- confidenzialità: proteggere i pacchetti da qualcuno di esterno alla rete, uso dell'algoritmo stream cipher RC4 (poi scoperto vulnerabile, ma è n'altra storia).
- integrità: il pacchetto non doveva essere modificato lungo il tragitto.
- : autenticazione: voglio che qualcuno possa entrare nella rete solo tramite delle credenziali.

2.4.3 RC4

Algoritmo PRNG specifico, usato per generare il keystream. Oggi è considerato debole, ma comunque WEP avrebbe avuto gli stessi problemi anche se fosse

stato buono.
ENC(KEY,MSG) = MSG $\oplus$ RC4(KEY,IV)
L'IV va generato per ogni frame e deve essere diverso per ognuno di essi, inoltre lo stream cipher deve essere sincronizzato in un canale che ha perdita. L'IV viene trasmesso in chiaro, se lo stream cipher è buono è buono non ho problemi. WEP è sicuro se l'IV non si ripete, altrimenti userei la stessa chiave e non avrei semantic security.
In Wi-Fi è "semplice" attaccare con Chosen Plaintext Attack o Known Plaintext Attack, anche se non conosco i messaggi ma li vedo in XOR posso ricavare qualcosa, l'IV è quindi cruciale e in WEP furono commessi due errori:

- La taglia era di 24 bit, molto piccola: circa 16.7 milioni di encryption diversi, se assumo 1500 byte di trama, con 7 Mbps di thoughput $\Rightarrow$ riciclo dopo appena 8 ore.
- L'implementazione fu lasciata libera $\Rightarrow$ COSA DA NON FARE MAI, MAI-IIII M A I (MAI PIÙÙÙÙÙÙÙÙÙ NON NOMINARE MIA MADRE CIT*), potrebbero metterci tutti 00..0 se non leggono la specifica.

Inoltre, conviene generare l'IV random o in maniera sequenziale? Se lo genero random, ho il 50% di probabilità di avere un duplicato dopo circa 4000 frame (birthday paradox).Meglio quindi sceglierli in serie, però sono suscettibile ad un attacco: se il router viene spento e riacceso, la sequenza riparte da 0. L'attacker può catturare i messaggi, rebootare di nuovo e fare un Chosen Plaintext Attack, ricreando la sequenza degli IV.
Il reboot dovrebbe prevedere un seed sempre diverso, ma qui il generatore è PRNG.
L'attacker può quindi creare un dizionario:
per ogni IV avrà il corrispondente keystream = RC4(IV,K), così da poter usare la coppia per attaccare (manda un contenuto noto ed una volta ricevuta la risposta ricava MSG $\oplus$ keystream $\oplus$ MSG ed ottiene il keystream).
Se RC4 è buono, non deve essere possibile ricavare una entry del dizionario avendo tutte le restanti. Un altro attacco può consistere nell'aspettare che l'IV si ripeta.

2.4.4 User authentication

Autenticazione: mostrare davvero chi sei. Non va confusa con l'identificazione, con cui fornisco nome cognome etc..., l'autenticazione è la prova che controllo la mia identità digitale.
Non è semplice definire l'autenticazione, molti siti difatti permettono di creare ad esempio mail che non mostrano il mio nome e cognome e quindi questo non mi identifica, ma voglio comunque che l'account sia usato da una sola persona.
Metodi di autenticazione:

- Metodo "base": una password, un pin, chiave segreta etc...
- device fisici: smart card, token digitali, hardware non clonabile.

- biometrics: impronta digitale, retina etc...
- behavioural authentication: registrazione vocale, hand writing etc...

In WEP non fu prevista l'autenticazione di ogni singolo utente. L'obiettivo era quello di riuscire ad autenticare un gruppo di persone che potessero entrare nella rete. L'idea: chi sta nella stessa rete può essere visto dagli altri, quindi usa los etsso meccanismo di enrcyption.
Il grant di accesso era dato solo a chi aveva una password comune, pre-distribuita. Come provare l'autenticazione: non posso inviarla in chiaro (sono in Wi-Fi), quindi in WEP venne introdotto un meccanismo che prevedeva di effettuare delle operazioni sulla password; il risultato non doveva dare informazioni sulla password.
Meccanismo: conosco k, l'access point mi manda una challenge ed io gli fornisco un encryption della challenge e della password.Per ogni nuovo utente devo usare una challenge diversa, può seere una stringa in plaintext, in WEP era di 128 bit. Una volta ricevuta la risposta, l'AP decriptava e se il risultato era la k dava l'accesso.Tecnica symmetric key, buona? Sì, trovo in un libro scritto da gente top nel settore che mi descrive esattamente questa tecnica, se la challenge è random e senza ripetizioni sono al sicuro.
In WEP non è così,anzi l'autenticazione aiuta a violare la confidenzialità: come detto sopra, posso effettuare un Known Plaintext Attack per creare un dizionario IV — keystream = RC4(K,IV). Quello che vedo nel messaggio è chipertext = plaintext $\oplus$ keystream, devo conoscere il plaintext. WEP fornisce la possibilità di un KPA con l'autenticazione: CT $\oplus$ challenge = RC4(k, IV) = keystream. L'approccio è corretto, ma viene riusata la stessa chiave per cifrare la challenge ed i messaggi. La challenge inoltre è in plaintext $\longrightarrow$ nota $\longrightarrow$ Known Plaintext Attack.
Attacker si finge l'access point ed inviando challenge false costruisce il dizionario, una volta ottenuto il keystream (user mi manda challenge $\oplus$ keystream, io ho la challenge, faccio $\oplus$ ed ottengo il keystream) posso usarlo per criptare la challenge successiva e ottenere l'accesso.
L'autenticazione era certificata come robusta, ma l'implementazione non lo era, inoltre l'IV era lasciato all'implementatore $\Rightarrow$ MAI FARLO.
Il fix fu di far scegliere sia la challenge che l'IV all'acces point, ma in ogni caso essendo l'IV corto si sarebbe ripetuto.

2.5 Integrità

Per l'integrità, l'idea fu quella di utilizzare CRC-32, il controllo a lvl2, come integrity check. Non è certo però che funzioni, ma l'attacker vedrà solo il ciphertext, quindi anche se il CRC-32 non è buono è protetto dall'encryption: assunzione errata. Confidenzialità non garantisce integrità. CRC-32 è lineare rispetto allo XOR: se faccio CRC32(A) e CRC(32) di B (con A e B due messaggi diversi) fare CRC32(A $\oplus$ B) = CRC32(A) $\oplus$ CRC32(B). Inoltre, lo XOR era proprio usato per l'encryption $\Rightarrow$ deadly. Ogni messaggio può subire modifiche o injection:

ho un plaintext M di, cui l'attacker vuole flippare 3 bit, ho CRC32(M), ed ho M $\oplus$ RC4(K,IV). Produco un messaggio δ che è uguale ad M, ma con i 3 bit che voglio flippare pari ad 1, computo CRC32(δ), prendo il precedente ciphertext e ne faccio lo XOR con il mio:
keystream $\oplus$ M $\oplus$ δ — keystream$_2$ $\oplus$ CRC(M $\oplus$ δ) (per linearità dello XOR).
Ho un nuovo messaggio valido (nelle ipotesi che δ sia pari ad M), quindi posso eseguire un Man in the middle attack.
Dopo WEP ci fu 802.11 in cui il protocollo è WPA (anche WPA2 con AES), venne inoltre eseguita una patch firmware a RC4:

- IV a 48 bit
- protezione dell'IV
- etc...

Morale: rivolgersi ad un esperto di crittografia.

3 Autenticazione in generale

Le password sono deboli, faccio una panoramica per capire se una password è hard o no. Autenticazione: provo che ho una password, che per ora ritengo analoga ad un segreto (in pratica: un segreto è una stringa random). Se ho 4 bit, ho 2^4 possibilità, quindi la probabilità di indovinare al primo tentativo è $\frac{1}{2^4}$.
Una password è una stringa con meno entropia: se ho una password di N bit, la probabilità di indovinare al primo tentativo è >> di $\frac{1}{2^N}$.
Ho 4 problemi maggiori:

- password overload: gli utenti tendono a riutilizzare le password su più siti
- restricted charset: 1 byte = 8 bit, quindi 256 possibili combinazioni, ma da tastiera ne ho circa 100.
- low entropy: la password non è del tutto random, in quanto va comunque memorizzata.
- predictability: spesso le password sono associate alla vita reale

3.1 Password overload

Nel 2018, in USA, uno studio ha rivelato che ogni persona ha circa 130 account nel web: il 38% degli utenti riutilizza la stessa password su più siti. Se scopro una password di un account, posso usarla per accedere su altri siti $\Rightarrow$ cross site break.
Il 21% degli utenti modifica la propria password, ma le modifiche sono predicibili, inoltre il 46.5% delle password si cracka con 100 tentativi.

3.2 Restricted charset

Se ho un segreto di 8 byte, quindi 64 bit, ogni byte ha 256 diverse possibilità, quindi la probabilità di guess al primo tentativo è $\frac{1}{256^8}$. È un numero elevato? Una macchina "ordinaria" può effettuare 66 milioni guess/secondo, quindi il tempo medio per crackare la password è di circa 4431 anni: 1.8x10^{1}9 tentativi totali, divido per il numero di guess/secondo e divido per 2 (per fare una media), converto in anni. Le password però non hanno 256 possibilità per ogni byte, inoltre alcune usano solo lettere lower case, o al più numeri. Anche se vengono introdotti numeri e lettere upper case/simboli, spesso vengono messi in posizioni predicibili (es: all'inizio, alla fine, nel mezzo).
Sto considerando un attack brute force offline, in quanto proteggere un web server sarebbe possibile, ade sempio bloccando l'accesso dopo il 3° attempt fallito.

3.3 Low entropy

Ci sono dei tool dell'information theory che misurano la randomness. Le password non sono quasi mai random. Come misuro la randomness: Shannon entropy: Entropia H(X) = $-\sum_i p_i log_2(p_i)$, considero p > 0, inoltre il segno meno serve perché essendo p $\leq$ 1, il log mi da un valore negativo.
La quantità viene misurata in bit. esempio: un coinflip di una moneta equiprobabile ha H(X) = $-2 \cdot (\frac{1}{2} \cdot log_2(\frac{1}{2})) = 1$.
Per il dado ho $-6 \cdot (\frac{1}{6} \cdot log_2(\frac{1}{6})) = 2.58$. Per un random byte ho $-256 \cdot (\frac{1}{256} \cdot log_2(\frac{1}{256})) = log_2(256) = 8$, ma questo solo se i bit sono davvero random, altrimenti ho un valore minore di 8.
L'information value di x_i dipende da quanto x_i è inatteso: minore è la probabilità di un certo evento e più sono sorpreso, l'information content è quindi = $\frac{1}{p_i}$.
$-log_2(\frac{1}{p_i})$ è una traslazione della probabilità in bit, ad esempio $\frac{1}{4}$ diventa 2 bit.
L'information content è misurata quindi come $-log_2(\frac{1}{p_i})$.
Definisco l'entropia come l'average dell'information content degli x_i: H(X) = E[IC(X)] = $\sum_i p_i IC_i = -\sum_i p_i log_2(p_i)$.
Entropia: misura quantitativa per vedere quanto un evento random è predicibile, se pari ad 8 ho un byte perfettamente random, se è 0 è deterministico. Se N = 2^b possibili outcome allora b = $log_2(N)$, se l'entropia è pari a b, non posso predire. esempio: una moneta truccata con $\frac{1}{4}$ $\frac{3}{4}$ ho entropia pari a 0.81 < 1, quindi è predicibile.
Conseguenze: quando trasmetto un bit, trasferisco una quantità minore di informazione, posso comprimere di (1-quantità)% un file.
esempio: genero 3 bit random, ho entropia pari a $\frac{1}{3}$, ma se ci sono dipendenze? Ad esempio se solo il primo è un coinflip e gli altri due sono deterministici, ad esempio prendono il valore del primo ho entropia = 1. Non conta quindi la lunghezza della stringa, bensì la randomness.
Nel 1950, Shannon misurò l'entropia di un testo (in inglese), mostrando che il

linguaggio naturale è molto predicibile:
le lettere che comparivano nel testo non erano equiprobabili, quindi l'information content della singola lettera non è 4.71 (ovvero non ho probabilità di $\frac{1}{26} \Rightarrow -log_2(\frac{1}{26})$. Nota la prima,l'entropia della seconda etc... sono in un certo modo predicibili, ogni lettera inglese ha nella migliore condizione 1.3 di information content e 0.6 nella peggiore. Ogni lettera ha un contributo $\simeq 2$, e quindi generando una password avrò un entropia di circa 2 bit invece di 8.
Se ho 10 lettere random:
tempo di crack se puramente random $= 2^{4,7 \cdot 10} = 2^{47}$ attempts, mentre nel caso di password "umana" ho $2^{2 \cdot 10} = 2^{20}$ attempts; perdo un fattore $2^2 7 \simeq 134$ milioni, quindi molto meno robusta.

3.4 Predicibilità-Dictonary attack

In realtà, non serve nemmeno fare un brute force attack, ma si possono usare parole note. Faccio un dictionary attack: scelgo una serie di parole comuni in una lingua e faccio try su queste parole.
Se riesco a recuperare un set pubblico di password dal web quello brutto e cattivo costruisco il mio dizionario, che può anche essere mirato al singolo individuo (so nomi di familiari, date di nascita, gusti etc...).Gli attacchi funzionano sia online che offline, dove la forza dipende dalla potenza dell'hardware e dalla randomness della password.
Alcune statistiche:

- 25% delle password è del tipo 123456..., posso pensare anche ad un password sparying: prendo una passowrd e la provo su più account di diverse persone, in verticale (può essere molto efficace).
- 26% delle password sono di 6 byte, ne vanno usati almeno 16.

4 Autenticazione password-based vs autenticazione challenge-handshake

Dopo aver esaminato le password, vorrei un protocollo che mi permetta di usarle per autenticarmi. Ricordo che l'autenticazione è la prova di conoscere un segreto, senza dover per forza rivelarlo. Ho alcune alternative:

- PAP: mostro la password in chiaro
- CHAP: alcune informazioni leakate
- ZPK: nessuna informazione leakate (crypto forte), molto complessi e poco usati nella realtà

4.1 PAP

Il protocollo di autenticazione più semplice possibile: mando la password in chiaro, one way authentication. L'utente manda la sua password (insieme allo user id)ad un autenticatore, che fa un check nel DB in cui per vedere se ha una entry user id — password.
La password è pre-shared, ma la sto mandando in chiaro e se vengo intercettato è GAME OVER (se il canale permette eavesdropping, se è cablato sono leggermente più sl sicuro).
Inoltre non ho nessuna protezione da reply attack: se vengo intercettato, subito dopo l'attaccante può fingersi me, se non enrypto con un algoritmo semantically secure e se non ci sono limiti nel poter ripetere l'autenticazione; inoltre non permetto mutual authentication.
Il messaggio PAP è suddiviso in campi specifici a seconda del server a cui mi devo autenticare, se ad esempio ho server PPP: i campi sono espressi in ASCII ed ogni campo ha una semantica, me la studio, prendo il pacchetto e scopro tutte le info.

4.2 CHAP

L'authenticator mi manda una challenge, a cui rispondo con un messaggio contente il mio user id + hash(challenge,password,etc).Proof of knowledge: computazione di un segreto/password, mando una f(password) per dimostrare che la conosco. La funzione deve avere due proprietà:

- la computazione non deve rivelare il segreto, quindi non devo poter computare f^{-1}
- la funzione f non deve poter essere replicata da un attaccante.

L'authenticator mi manda una challenge ogni volta nuova, ovver una nonce. Lo user risponde con userID ed una funzione di challenge, key, etc...(parametri opzionali). La funzione deve rispettare le due proprietà, l'authenticator la ricalcola, dopo aver preso dal db la password corrispondente allo userID ricevuto; la funzione deve quindi essere deterministica.
Se la challenge è fresh non sono suscettibile a reply attack, inoltre la password non è inviata in chiaro.
La funzione può essere una hash function crittografica.
In CHAP è l'autenticatore che controlla tutto il processo: potrebbe accadere che un'attacker potrebbe intercettare la mia sessione kickarmi, sostituendosi a me. Per prevenire ciò, in CHAP è possibile far si che l'authenticator rimandi la challenge periodicamente, per accertare l'autenticità dell'utente. Tutto ciò in PAP non è possibile, ma il grande svantaggio di CHAP è che le password devono essere salvate in chiaro nel db.

4.3 Hash function

Funzioni crittografiche di base. Prende qualcosa in input e la riduce in polvere in maniera che sia incomprensibile ed irreversibile. Se ho un messaggio di lunghezza X, Y=H(X) è detto digest ed ha una taglia fissa; H(X) dovrebbe essere abbastanza semplice da poter essere computata su ogni X.
Non è sempre detto che le funzioni hash sia crittografiche, alcuni esempi di funzioni non crittografiche:

- 4 bit parity vector checksum: prendob blocchi da 4 bit e metto un bit di parità sui blocchi. La size del digest (ottenuto giustapponendo i bit di parità) è sempre pari a 4, e la funzione non è invertibile, in quanto l'inversa non è unica
- modula checksum: spezzo in chunk di interi (valori $\in$ [0,..9]) il mio messaggio, li sommo e ne faccio il mod1000.
- call center control: devo autenticarmi con username e password, mi richiedono un pin ma non lo mando tutto, bensì solo specifiche cifre.

Ogni hash function, anche non crittografica, non è invertibile.
Un hash function crittografica prende il testo e lo comprime in un digest di dimensione fissa, ma ha un importante proprietà: anche piccoli cambiamenti producono digest completamente diversi.Deve cercare di approssimare al meglio la generazione di una stringa random.Nel caso di funzioni hash non crypto, un cambiamento minimo è abbastanza prevedibile.
Un attaccante non deve in alcun modo ricreare l'hash digest: nel caso di non-crypto hash, cambiando i bit posso ottenere un messaggio diverso che mi fornisce lo stesso digest $\Rightarrow$ collisione. L'attacker non dovrebbe essere in grado di poter ricreare o modificare il messaggio così da ottenere il digest originale. Devono valere 3 proprietà:

1. Perimage resistance (one-way property): dato y=digest, deve essere computazionalmente difficile trovare X tale che H(X)=Y. Proprietà più forte del non invertibile, la computazione non deve poter essere ricavabile, anche se ho infiniti emssaggi che generano lo stesso digest.
   Corollario: per essere one-way la lunghezza del digest deve essere grande, non devo potervi fare brute force attacko crypto-analysis.
2. Weak collision resistance: dato X, è computazionalmente difficile trovare un X', che sia diverso da X, e tale per cui H(X) = H(X').esempio: sono un giudice di un tribunale, ho un hard disk su cui ci sono delle prove, lo do ad un esperto per analizzarlo. Come posso essere sicuro che le prove non siano inquinate? Computo l'hash dell'hard disk e lo metto ar pizzo (lo scrivo su un pizzino magari), così che se qualcuno inquina le prove gli do la sedia elettrica, perché vale questa proprietà e non può produrre modifiche tali per cui l'hash è lo stesso.

3. Strong collision resistance: ci sono funzioni che sono solide per la proprietà 1 ma non per la 2? Sì, ad esempio se considero Y= f(x) = $g^x modp$: g è dato, p è un numero primo molto grande. Se ad esempio so che 321475 = 3^x, riesco a ricavare x? Sì, ho che x = $\log_3 321475$, ma se aggiungo il *modp* non posso più farlo, non è facile computare l'inversa sotto determinate condizioni. Ma non rispetto la proprietà 2: se ho un X, mi basta sommare k·(p-1) per trovare lo stesso risultato; la funzione sembra difficile, ma non rispetta le proprietà.
Con la strong collision resistance voglio che sia impossibile trovare una qualunque coppia X_1, X_2 che collida.

4.4 Paradosso del compleanno e dimensione del digest

Voglio vedere come rispettare la strong collision resistance. Considero il birthday paradox: ho k=23 persone in una stanza, voglio associare a ciascuno un hash fatto sul loro giorno+mese di nascita. Probabilità che non ci siano collisioni tra uno dei k e gli altri k-1: $(\frac{364}{365})^{22} = 94.1\%$. Ma qual'è la probabilità che non ci siano collisioni tra tutti i k: $1 \cdot (1\text{-}\frac{1}{365})..... \cdot (1\text{-}\frac{22}{365}) \simeq 49.3\%$. Quindi la probabilità di collidere è il complementare, ovvero 1-0.493 = 0.507 = 50.7%. esempio: ho n bit di digest, $N = 2^n$ diversi risultati. Considero k messaggi:
P(no collisioni) = 1-p = $\frac{\frac{N!}{(N-k+1)!}}{N^k} = \frac{N}{N} \cdot \frac{N-1}{N} \cdot \frac{N-2}{N} ..... \cdot \frac{N-k+1}{N} \simeq (1 - \frac{1}{N}) \cdot (1 - \frac{2}{N}) ..... \cdot (1 - \frac{k-1}{N}) = \prod_{i=1}^{k-1}(1 - \frac{i}{N})$.

Nelle ipotesi di N grande, 1-i ≃ -i ⇒ $\frac{-i}{N} \simeq e^{-\frac{i}{N}}$, quindi ho $\simeq \prod_{i=1}^{k-1} e^{-\frac{i}{N}}$, ma questa è uguale alla somma degli esponenti ⇒ $e^{-\sum_{i=1}^{k-1}\frac{i}{N}}$. Ho inoltre che $\sum_{i=1}^{k-1} i$ è la somma di Gauss = $\frac{k\cdot(k-1)}{2}$ e quindi ho $e^{-\frac{k\cdot(k-1)}{2}} \simeq e^{-\frac{k^2}{2N}}$, approssimando k-1 a k. Questa è la probabilità di non avere collisioni: 1-p = $e^{-\frac{k^2}{2N}}$ da cui ln(1-p) = $-\frac{k^2}{2N}$ ⇒ k = $\sqrt[2]{-2N \cdot ln(1-p)}$ ⇒ k = $\sqrt[2]{2N \cdot ln(\frac{1}{1-p})}$.
Quindi all'aumentare del numero di messaggi k, aumenterà la probabilità di collidere. L'obbiettivo è capire quanti messaggi devo raccogliere per avere il 50% di probabilità di collidere:
$\sqrt[2]{N} \cdot \sqrt[2]{ln(\frac{1}{1-\frac{1}{2}})} = \sqrt[2]{N} \cdot \sqrt[2]{2}\ \sqrt[2]{ln2} \simeq 1.177\sqrt[2]{N} \simeq \sqrt[2]{N}$. Siccome $N = 2^n$, avrò k = $1.117 \cdot 2^{\frac{n}{2}} \simeq 2^{\frac{n}{2}}$. Se la RAND fosse una perfetta hash function: con 32 bit avrei 4.5 miliardi possibili output, e devo raccoglierne solo 60k per avere una collisione.
Per md5, con k = $1.8 \cdot 10^{19} = 2^64$ oggi è considerato weak, mentre per SHA256 ho $3.4 \cdot 10^{38}$.

4.5 PAP vs CHAP

Posso chiedermi quale fra i due è il più robusto. Bisogna comunque avere chiaro l'adversary model:

- eavesdropping attack: ascolto chi trasmette
- rubo i dati dal db

In PAP, se qualcuno ascolta il canale è finita, perché la password viene trasmessa in chiaro, quindi c'è la necessità di proteggere il canale di comunicazione (SSL, TLS, EAP/TTLS), ma nel caso in cui ci sia lo steal del DB PAP è molto più robusto, in quanto posso salvare le password non in chiaro. CHAP è invece meglio nel caso del 1° attacco, ma nel secondo no: non posso salvare l'hashing delle password, nel caso di PAP può essere effettuato brute force e la riuscita dipende dall'entropia delle password.
Perché in CHAP devo per forza salvare la password in chiaro: in CHAP la challenge è sempre diversa, non posso salvare H(psw, challenge) e se salvo solo H(psw), non posso ricavare la password perché la funzione non è invertibile.
Provo a modificare CHAP: faccio l'hash della password on the fly, ovvero faccio hash(hash(psw),challenge) e mando all'autenticatore, che ora può salvare l'hash della password. Ma in questo modo, se il db viene crackato, non devo nemmeno fare sforzi: userò l'hash della password per rispondere alla challenge e mi autenticherò.
In conclusione:

- Se l'attacco è sul canale di comunicazione, è meglio usare CHAP
- Se il canale è robusto ma il BD no, meglio usare PAP.

Quando valuto la sicurezza di un sistema devo capire bene cosa l'attaccante può fare e come posso difendermi.
Un modo per poter mitigare CHAP è aggiungere del "sale": authenticator mi manda la challenge più del salt, io prendo il salt e lo combino alla password e ne faccio l'hash, che uso per fare hash con la challenge.Cambiando il salt, anche il risultato cambia, quindi posso creare un DB con UID e l'hash di (psw,salt).Rimane comunque il problema in caso di db stealing, ma risolvo buttando via il db e ricostruendolo; sono inoltre soggetto a brute force e dictionary attack. Ho bisogno di un DB aggiuntivo,che proteggo in maniera più forte, in cui salvare le password in chiaro per poterlo ricostruire in caso di steal.

4.6 Hash Chain

One time password: voglio una password diversa per ogni tentativo di autenticazione, così da essere al sicuro da reply attack. Sembra una cosa triviale: creo uno userDB con una lista di password random, per ognuna metto un flag che mi indica se è stata già usata o no, quando ne ricevo una la segno.
Su na scala reale: se ho molti utenti, il numero di password totali è considerevole, il problema non è tanto nella taglia del DB quanto nel dover cambiare la

struttura del DB. L'idea è quindi di generare un numero random/pseudorandom di password da un seed. Uso una hash function crypto, come SHA256, a cui passo il seme e computo il digest. Parto da un seme P[0] e genero P[1] = H(p[0]) e così via, devo quindi salvare solo P[0] per poterle computare tutte. Uso P[0], poi P[1] etc..., ma il modo è errato: se riesco a leggere P[0] poi posso generare tutte le successive, quindi faccio il contrario: mando P[n], poi P[n-1]..., l'attacker non andare al ritroso nel calcolo (sto usando una crypto hash function), non è possibile invertire la funzione.
L'authenticator computerà da P[0] a P[n], ma su larga scala questo è oneroso: quello che viene fatto è computare offline, ade sempio durante la registrazione, fino a P[n+1] e salva solo questa. Quando riceverà P[n], farà P[n+1] = H([n]) e lo confronterà con il valore di P[n+1] che ha salvato. Il numero di password è finito, quindi dopo un po' sarà necessario rigenerare la chiave, per creare una nuova hash chain.
Posso avere un problema nel momento in cui ricevo l'ok per l'autenticazione, mando il valore successivo e non accade nulla: posso tentare le altre password, se si perde la sincronizzazione tra client e server, so qual'è ultima password correttamente ricevuta lato server (perché ho salvato P[n+1]) e quindi definisco una finestra di tolleranza per cui provo a fare hash per vedere se mi torna il risultato (la finestra va a salire), ovviamente il valore deve essere limitato.
Benefit della OTP:

- Invio in chiaro
- Rilasso la server security: l'authenticator salva solo la password che si aspetta di computare, quindi in caso di db steal non ho informazioni sulla password esatta.
- minore complessità del db

Problemi:

- Dimensionare bene n
- client side è vulnerabile, in caso di key steal è finita.

4.7 2-factor authentication

Non posso fidarmi della password dell'utente, quindi spesso viene inviato anche un codice (via sms, mail etc...), in modo che l'attacker deve trovare entrambe per poter avere successo. Il codice è un one-time authentication token generato su un device differente e ricevuto su un canale differente.
Deve essere human friendly: un codice da 6 a 8 cifre, possibile generarlo con un hash su cui poi viene effettuato un troncamento...
Se assumo che sia client che server sono sicuri, non ho bisogno di usare un hash chain: ho due protocolli possibili, HOTP e TOTP

4.7.1 HOTP-Hash One Time Password

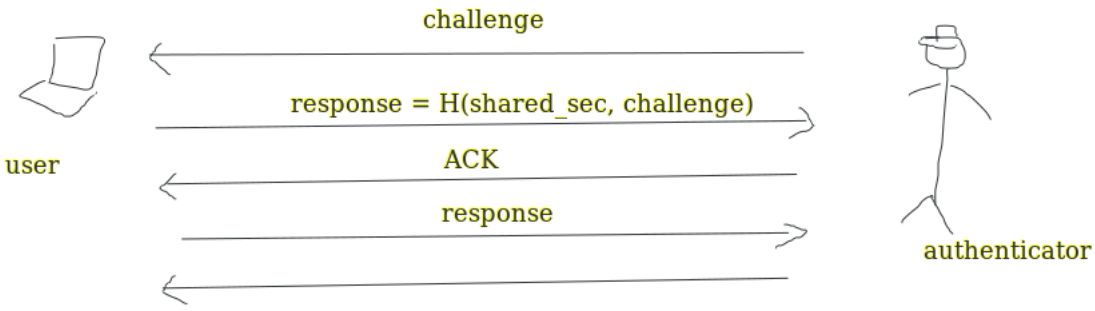
Ho client e server sicuri, c'è il segreto shared su entrambe, non computo P[n] = H(P[n-1]), bensì P[n] = H(segreto, n); ad esempio P[35] = H(secret, 35) e così via.
Anche se ottengo uno dei P[i] non posso ricavare gli altri se la funzione hash è crypto. Inoltre k può essere "infinito", parto da un counter e non devo precomputare nulla. Uso SHA256(k,n) che è un HMAC, e prendo il troncamento del risultato (6-8) digits.

4.7.2 TOPT-Time-based One time password

Sistema più sicuro, in cui non devo generare una OTP, am ha come assunzione che il tempo sia sicuro: ho un time sicuro, parto da un timestamp TS iniziale. TOPT = HOTP(k,T), dove T = $\frac{TS}{X}$, dove X è il range nel quale penso non avvenga un reply attack (solitamente 30 secondi). Questo TOTP ha valore per un certo lasso di tempo, computo il valore ad esempio alle 10:46 ed il server riceve alle 10:46, ma se c'è dello sfasamento il valore non torna. Per incrementare la robustezza posso provare ad usare valori precedenti, tolleranza ad esempio di uno slot temporale.
Se il tempo non è sicuro, un attacker può cambiare il tempo e riusare una OTP. Ad esempio: server NTP ha precisione di O(10 ms), quindi non è sicuro.

4.8 Mutual authentication con CHAP

Assumo che CHAP sia un protocollo sicuro, però è pensato per single authentication. Provo ad adattarlo per mutua autenticazione. Pro tip: non fare mai quello che segue, non si fa. Esce fuori un casino:



Non è una buona idea: uso la stessa chiave per rispondere alla challenge sia dello user che dell'authenticator, ma invece di essere l'utente il primo ad autenticarsi, chiede all'authenticator di farlo (posso farlo se il canale è full duplex). Se però la challenge è la stessa, non è detto che l'authenticator si renda conto che non è cambiata: quando ricevo la risposta posso compiere un reply attack, perché la challenge può essere uguale. Cambio il protocollo in un unico mutuo protocollo: l'authenticator manda il suo nome e la sua challenge, l'utente genera una nuova challenge, manda all'authenticator la nuova challenge e la risposta; ora l'authenticator può verificare che la risposta sia diversa.
Reflection attack: mando la challenge C_1, l'utente mi manda C_2 + hash per C_1. Fingo una perdita di connessione, sospendo la sessione e mando come challenge C_2, l'utente vede C_1 e C_2, ma non sa che deve verificare le due sessioni, quindi mi manda la risposta a C_2 e la nuova challenge C_2. Ora fingo che la sessione è tornata up e mando la risposta a C_2.
Ulteriore patch: ogni nuova sessione rende invalida la precedente, posso comunque compiere un man in the middle/intertwining attack. L'attaccker prende il messaggio, lo manda all'authenticator e riceve l'autenticazione dal server, quindi fa credere all'utente che è connesso col server.
Per fixare: richiedo che venga effettuata una computazione sulla challenge dell'utente e dell'authenticator, ovvero di effettuare crypto binding sulle due

challenge. Lo user fa crypto binding su C_1,C_2 e si fa mandare dall'authenticator l'hash di C_2,C_3, ma così ci sono troppe challenge: l'attaccante invia C_1, mi faccio mandare l'hash di C_1,C_2 posso iniziare una nuova sessione con C_2.
La soluzione prevede che sia l'utente che l'authenticator usino le stesse due challenge, ma scambiando l'ordine con cui viene effettuato l'hashing,ma in questo modo ho progettato un protocollo diverso da CHAP.
Avevo due sessioni indipendenti, l'unico modo per renderle dipendenti è usare crypto dependency sulle due challenge.

5 Nonce

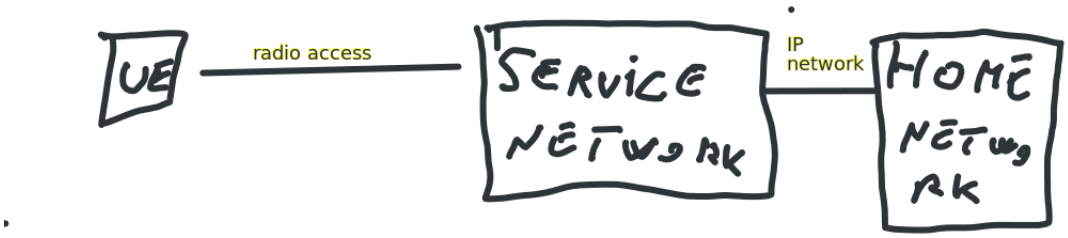
Una nonce è un valore sempre fresco, ovvero ogni volta diverso.
Sono di 3 tipi:

1. Random challenge: $\sqrt[2]{n}$ in termini di randomness
2. # seq : più robusto in termini di predicibilità, n.
3. timestamp: è predicibile, ma devo avere garanzia sul tempo (es: GPS e Galileo, GPS può essere spoofato).

La nonce è un superset di possibili challenge, può prevede l'utilizzo di più metodi

6 Challenge-response authetication in 2G/3G/4G(5G)

Architettura semplificata di un sistema cellulare:



La rete cellulare deve garantire almeno queste 3 parti, la serving network offre un servizio di roaming agli altri operatori(non è detto che sia lo stesso operatore dell'utente). l'autenticazione serve perché in questo modo il dispositivo può accedere alla rete e l'operatore sa chi sta accedendo. Usando un protocollo come PAP, la serving netowork vede la mia password e se sono in roaming in paesi molto ad est non so se esistono regole sulla privacy.
Non posso fidarmi della serving network, quindi uso un protocollo CHAP-like: la rete è complessa, la parte che gestisce l'autenticazione comunica con la home network.
In 3G, il mio device dice alla rete dove sono e chi sono, la serving network contatta la mia home network per ottenere i parametri di configurazione e a questo punto posso autenticarmi usando le credenziali con procedure crypto. In realtà, viene derivata anche la chiave per l'encryption: AKA = Authentication and key agreement, userò anche una chiave successivamente per criptare i messaggi.

6.0.1 Autenticazione in 2G

L'autenticazione è unilaterale: la SN manda alla HN l'IMSI che gli comunica il mio device (l'IMSI è il codice della sim) ed una challenge, ovvero chiede alla HN quale risposta deve aspettarsi dall'autenticazione dell'utente. La HN possiede l'id dell'utente e la password, riceve la challenge e ne fa l'hash usando il segreto, producendo l'SRES, fatta da un authenticator trusted: la HN è il ground truth. Inoltre, fornisce la K_c, ovvero la chiave temporanea che verrà usata dopo l'autenticazione per criptare i messaggi. La SN mi manda la challenge (128 bit random challenge in 2G), io (la mobile station) usa la funzione A_3 (simile ad un hash function) a

cui passa il segreto e la challenge. La risposta è di 32 bit e viene mandata alla SN, che controlla se è uguale alla SRES ottenuta in precedenza dalla HN. Sono protetto, perché la SN non conosce il segreto k_i (identity key) dell'utente.C'è anche la fase di key agreement: (k_i, rand challenge) $\longrightarrow$ k_c usata per criptare, schema di symmetric encryption.
In wi-fi: dispositivi che si collegano allo stesso AP condividono la stessa chiave di accesso, la protezione è solo dall'esterno, per la protezione interna servono ulteriori meccanismi (TLS,SSL,...) Nel modello appena descritto ogni utente ha una chiave diversa e questa chiave cambia per ogni nuova sessione.
La challenge è una nonce, con cui computo k_c mediante la funzione A_8 (anch'essa simile ad un hash function).
Se l'utente si collega da più celle, devo ogni volta ripetere il processo descritto sopra, quindi questo è molto lento. L'idea è quella di fornire alla SN un vettore di ¡challenge, response, k_c¿xN, in modo che la SN abbia N triple e questo porta a diversi vantaggi:

- viene contattata una sola volta l'HN, quindi pagherò il delay della connessione una volta sola
- la challenge random è una nonce che deve essere generata propriamente, se la genera la HN e non la SN sono più al sicuro.

Quindi lo schema di challenge-response:
challenge: RAND — secret: K_i — hash: algoritmo A_3. Gli algoritmi A_3 ed A_8 prendono 128 bit di K_i + 128 bit di RAND e restituiscono rispettivamente 32 bit di SRES e 64 di K_c (anche detto il segreto di Pulcinella). 2^{64} non è computabile, ma 2^{54} sì: la chiave K_c era di 54 bit, estesi a 64 con 10 zeri alla fine.

6.0.2 Scelta degli algoritmi

A_3 ed A_8 girano nel chip della sim. Ma io non ho accesso al chip della sim, quindi non conosco l'implementazione di A_3, ma non ne ho bisogno. Nemmeno la SN deve conoscerla, il risultato sta già nella tripla, è proprio la response, che è l'SRES, quindi ogni operatore può scegliere gli algoritmi che preferisce. Gli operatori scelsero di usare un algoritmo noto, COMP128 che non era open source, ma veniva tenuto nascosto $\Rightarrow$ security by obscurity. CI fu un leak del codice, e questo venne analizzato e violato da crypto guys molt forti in O(min), rendendo vulnerabili milioni di sim.
Morale:

- Security by obscurity non funziona, perché è difficile che se scoprono una vulnerabilità non la dicano.
- Algoritmo pubblico viene validato dai crypto shark che cercano di romperlo per il clout.
- Preferisci sempre l'open source al codice chiuso per la sicurezza

6.1 Autenticazione in 3G/4G

Il problema del 2G sta nel fatto che l'algoritmo per computare K_c era vulnerabile, ma non c'era nemmeno mutual authentication: suscettibile ad over the air attack, ovvero una finta base station a cui l'utente si collega. La base station non prova mai la sua autenticità, e non voglio questo. Uso un algoritmo open source per la mutua autenticazione: la mobile station comunica il suo IMSI alla SN, che lo manda alla HN e riceve una 5-pla ¡rand, XRES, C_k, IK, AUTN¿xN. L'IK è l'integrity key, usata per garantire l'integrità del messaggio. Questo perché l'encryption non la garantisce, quindi serve una chiave diversa dalla K_c per garantirla.L'AUTN serve invece per provare l'autenticità della rete all'utente, è il Network Authentication Token.
La SN mi manda l'AUTN e la rand, provandomi di essere trusted, ma l'AUTN è stato prodotto dalla HN, quindi voglio che sia il device a produrre la challenge che la SN dovrà risolvere. In questo caso non è così, dovrei avere due nonces, uno dall'MS alla SN ed uno dalla SN alla HN.
Funzioni usate:
f_2(K, RAND) = 32 bit di RES
f_3(A_8 equivalent)(K, RAND) = 128 bit di C_k
f_4(K, RAND) = 128 bit di IK
Tutte dipendono solo dal segreto dell'utente e dalla random nonce, le implementazioni sono note.
Sarebbe possibile autenticarsi con un solo messaggi, usando il timestamp:
se la mobile station e la SN usano ad esempio GPS, sotto l'assunzione che il tempo sia sicuro, conoscono perfettamente la nonce usando TS.
Farò H(TS,K), non posso avere reply attack etc... e con questa forma di nonce evito l'uso della challenge.
La mutual authentication in 3G+ (simile anche in 4/5G) prevede che la MS invii una nonce, invece di usare una quantità random, usa un #seq number: in questo modo posso sapere quante volte ho fatto un accesso e qualcuno nella rete mi dice che mi sto autenticando per la #seq + 1 volta. Se mi arriva un messaggio con un #seq vecchio, scopro che è un reply attack. Se il vece il #seq è più grande anche solo di 1, va bene (definisco anche in questo caso una tollerance window): i numeri di sequenza della MS e della HN devono essere sincronizzati, perché la SN si autentica alla HN per ottenere il vettore di credenziali.Quindi, quando mi autentico, uso il #seq come challenge implicita, risparmiando un messaggio: il messaggio che mi fornisce la SN è un'operazione della nonce + k, usando le credenziali fornite dalla HN; in questo modo so che la SN ha le mie credenziali. È un 2-way exchange con 2 messaggi, il problema è che la MS e la HN devono essere sempre approssimativamente sincronizzate; se si perde la sincronia, servono dei meccanismo per ripristinarla.
Format dell'AUTN:
N° sequenza a 48 bit — AMF: 16 bit di auth and key management — 64 bit di MAC-A, derivato come segue: MAC-A = f_1(k, SQN, AMF, RAND), la coppia SQN + RAND corrisponde ad un crypto binding, K ed SQN sono la proof of knowledge di k, l'AMF sono informazioni aggiuntive.

Una volta connesso alla service network, questa mi manda un TMSI, ovvero un identificatore temporaneo allocato dinamicamente quando mi collego; nelle prossime autenticazioni userò questo TMSI (GUTI in 4G). Il TMSI è meglio per la privacy, in quanto se mi collego usando sempre l'IMSI, posso essere tracciato, invece qui uso una quantità che via via cambia.
L'unica vulnerabilità è che la prima volta l'IMSI viene mandato in chiaro, bisogna cercare di usarlo il meno possibile.
Mi autentico ed ogni volta che mi ricollego cambia il TMSI, occorre fare molto lavoro aggiuntivo per evidare che i TMSI siano legati fra loro. Devo inoltre proteggere il sequence number SQN: potrei non trasmetterli ed usarli davvero come informazioni implicite, oppure produrre un encryption con la chiave, ma prima del collegamento non la ho: se però sono autentico ed anche la SN lo è, allora vuol dire che entambi conosciamo k, quindi possiamo derivare qualcosa da k. Faccio l'encryption del SQN con un pattern AK apparentemente random: rendo AK computabile per entrambe gli end, quindi poi la base station potrà risolvere:

- leggi random
- computa f_5(k, random) = AK di 48 bit
- de-anonimizza #seq : #seq $\oplus$ AK $\oplus$ AK = #seq
- leggi il resto del pacchetto
- computa f_4(#seq , k, rand, AMF) = MAC-A
- controlla il MAC-A

7 Pro-tip: come generare password a partire da un segreto master

Ho il seguente problema: devo generare chiavi infinite per infiniti utenti: posso usare un PRNG e mettere i record userID — password in un database e proteggere il DB. Pro tip: uso un master secret S, che tengo al sicuro, e quando devo generare una nuova chiave per l'utente uso un identificativo univoco per l'utente (es: il codice fiscale) e faccio hash(S, UID).

8 Message authentication-Integrity

In 3G e oltre ho l'IK per l'integrità del messaggio, faccio quindi messsage authentication. Ricordo che la confidenzialità è diversa dall'integrità: con la prima,voglio nascondere il contenuto del messaggio mentre con la seconda voglio che il messaggio non sia modificato durante la trasmissione, solo la sorgente legittima dovrebbe poterlo forgiare.
Ho bisogno di un algoritmo che garantisca l'integrità.

8.1 Message Authentication con symmetric key

Il sender ed il receiver che hanno una stessa chiave k, ad ogni messaggio aggiungo un TAG = gen_tag(k, message). Il messaggio ha una size arbitraria, ma il tag deve avere size fissa, quindi userò qualcosa di simile ad una funzione hash. Il receiver riceverà il pacchetto e ricomputerà il tag, confrontandolo con il tag ricevuto.
esempio: message authentication code (MAC) è più debole della firma digitale
La firma digitale non può essere contraffatta, nessuno può modificare un messaggio firmato da me. Nel caso del tag sender e receiver possono modificare il messaggio, chiunque possiede k può farlo. La digital signature ha la non repudiation property o source authentication. Ad esempio, in una chat multicast si usa la stessa IK, quindi chiunque può produrre un messaggio spaccandosi per me, con la firma digitale questo non può accadere.

8.2 Message Authentication con hash functions

Ho mai detto che una hash function può essere usata per msg auth?
Nuovo problema: come usare hash function per msg auth e ci saranno problemi (perché non è quello il purpose per cui è pensata).
Encrytpion non garantisce integrità a meno che si usi un authenticated enrcyption ⇒ AEAD algorithms, Authenticated Encryption Associeted data ovvero un algortimo che fa sia encryption e authentication.
In TLS 1.3 (la nuova) hanno proibito di usare algoritmi che non abbiamo authentication, quindi sono AEAD.
AES-128 o AES-256 è enrcyption only, AES-GSM o AES-CCM sono auth enrcyption.

8.3 Message authentication with simmectric key

Prendi msg m, computa con una chiave k nota ad entrambi gli end(nota ⇒che è pre-shared) C'è una funzione che è usata per generare il tag: riceve una size arbitraria e produce un tag di size fissa e possibilmente piccola (non troppo, per birthday paradox). Trasmetto msg + il tag, message authentication code aggiunge bytes al msg, c'è dell'overhead in quanto non deve aver infromazione(il msg in se è al massimo livello di entropia). Recevier verifica il tag usando la stessa funzione, nota e condivisa, generando il nuovo tag dal msg + chiave k e lo confronta con quello ricevuto.

8.4 Definizione di sicurezza per Message Authentication Code

IND-CPA model definiva la confidenzialità, posso trovarne uno analogo per msg auth?

Sicurezza in integrity vuol dire che l'atk non può essere in grado di creare un nuovo msg o poter modificarne uno; anche se il msg modificato perde di senso è considerata una violazione.
Formalmente, faccio un "gioco" contro l'attaccante:

- attacker model può essere Known Message Attack o Chosen Message Attack, ovvero attacker può chiedere qualsiasi coppia (msg,tag) precedente
- può essere adattivo ovvero che il msg è scelto dopo una analisi della situazione.

Ora, se l'attacker scegli uno nuovo messaggio m, diverso da quelli del passato per cui ha i tag, non deve poter forgiare il nuovo tag per il msg m.
Formalmente, la probabilità di forgiare una coppia valida deve essere un ϵ(prob dell'ordine di 2^{-100}):

- Devo escludere che il tag sia di 1 byte
- Non può tirare a caso il tag del msg con scelta puramente random. Se fosse di 1 byte $\Rightarrow$ avrei $\frac{1}{256}$, che non va bene, il minimo è almeno 96 bit di tag (meglio 256).

Differenza cruciale nella scurezza: IND-CPA l'attacker poteva scegliere se il msg era A o B e aveva esattamente $\frac{1}{2}$ possibilità.
Message integrity protegge dal man in the middle? Sì, genero il msg m, produco il tag=F(K,m). L'atk intercetta il mesaggio e vuole cambiarlo: se F è cryptographically strong:

- K non può essere computata dal msg e dal tag.
- Non posso cambiare il tag in un nuovo tag senza conoscere k, non posso computare tag*=G(K,m*)
- Non posso cambiare m in m' così che F(K,m)=G(K,m')$\Rightarrow$ anticollision property.

Se lo schema è sicuro, allora potrò sempre intercettare un mitm atk.Mitm ha due aspetti:

- networking class: triviale farlo,ARP poisoning, DNS spoofing.
- Il msg è efficace se posso modificare il msg, non solo cambiare il flow dei msg.

Un buon algoritmo deve anche proteggere dalla creazione di un nuovo msg: message spoofing$\Rightarrow$ creo un nuovo messaggio in cui metto un ip fake facendoti credere che è quello con cui vuoi comunicare.
Posso risolverlo con un auth mechanism:

Se ogni msg è autenticato: DNS è autenticato in plaintext, come faccio a sapere che è proprio, es. Google.com?
Devo aggiungere qualcosa che mi garantisce che sia Google ad esempio con un tag.(la versione di DNSSec dovrebbe proteggere da questo, ma questo aggiunge complessità alla rete quindi si continua ad usare DNS.)Posso spoofare msg, ma devo conoscere il tag⇒se algoritmo è buono probabilità è un ϵ.
Questo schema NON protegge da un reply attack:
voglio mandare due messaggi, es due transazioni. Produco due msg identici, ma la F si applica alla chiave⇒la F deve essere deterministica (va ricomputata all'altro end)e quindi i tag saranno gli stessi,MAC non è abbastanza.Ma se i messaggi hanno un contenuto diverso: timestamp, num. seq etc.. potrei dire che non ci sono problemi.Ma non è così: l'applicazione dovrebbe essere disegnata senza avere in mente problemi di sicurezza.Il protocollo deve essere sicuro, non mi deve importare dell'applicazione.
Prevengo reply atk: uso le nonces, devo garantire a livello di protocollo che tutti i messaggi siano diversi, aggiungo una nonce ai msg. Computo il tag sul msg+nonce, posso mandare la nonce in chiaro.

- Se uso seq num: come gestisco i reboot?Devo prestare attenzione. Parto da 0 e vado avanti, però perdo alcuni n° sequenza, come faccio a dire che i pacchetti ricevuti con alcuni buchi in mezzo sono ok? Devo tenere in mente l'utlimo correttamente ricevuto per discriminare reply atk.
- Random number, se truly random sono meglio.Non ho probelma del reboot, ma come controllo se pkt è nuovo? Ho un certo n° bit,quindi dovrei tenere tutt la lista de msg precedentemente ricevuti, costo di memoria e di computazione per il controllo.
- Timestamp migliore possibile, ma il tempo deve essere garantito o ho problemi.

Settare una nonce sembra facile ma non lo è, la maggior parte dei probelmi implementativi è qui.
1° ingrediente:
Hash functions: molto veloci, sono anticollsione se cryptographic. Buoni prodotti:

- SHA-2 family(SHA256, SHA224,SHA384⇒SHA512 troncato, SHA512). Nel passsato SHA1 e MD5, MD5 la più comune e famosa funzione hash,oggi tutte e due rotte.
- Next: SHA-3 family, sempre gli stessi digest ma con approcci differenti.

es: in TLS e Ipsec, SSH non troppo serio si usa SHA256, sha256sum fa hashing di file su Linux.
2° ingrediente:
Includo il segreto nell'hashing del messaggio messaggio. Facile? Ma dove metto l'auth key nel msg?Lo metto dopo il messaggio: H(M||K), o faccio il contrario?

O in altri modi? Ad esempio metterlo sia all'inizio che alla fine etc..Perché me ne preoccupo?
Una funzione hash teorica è una black box, c'è anche definizione per la perfect hash function:
Random Oracle: black box, che preso input X, H(X) = valore truly random, ma che si ripete se X è lo stesso. Ma le due cose non possono coesistere, H(X) deve essere computabile. Nella teoria questo è il modello ideale (come per one time pad) che vorrei avere, ma non posso implementarlo.
Devo vedere nel box:tutte le hash functions (tranne SHA-3, oggi non usate) sono costruite con la costruzione iterative Merkle-Damgard: è difficile trovare f tale che: f(any size)$\longmapsto$ fixed size. Ma è possibile trovare f t.c:
f(fixed size)$\longmapsto$ samller fixed size otuput. Compression function, che possono essere molto buone.
es: sha256
prendo msg di k bit, paddo il messaggio in modo che il risultato(compreso i 64bit di lunghezza del messaggio) sia multiplo di 512 bit: se ad esempio la size del mio file è 1025 bit, metto un bit ad 1 seguito da vari zeri, alla fine del msg mette la size del msg come lunghezza modulo 2^{64}, sono 64 bit (faccio il modulo nel caso in cui lunghezza sia maggiore di 2^{64}, così che sia di size fissa). Ora taglio il msg in chunks di 512 bit: parto con un initialization vector(non crypto) che è noto e fisso, deve poter essere ripetuto. SHA256 prende IV 256bit,diviso in 8 gruppi da 32, è una costante.L'IV fa sì che la funzione di compressione prenda 512(il chunk) + 256(l'IV) = 768 bit di input.Questo perché SHA256 usa aritmetica mod 32 o 64 a seconda dell'architettura.La compression function comprime i 768 bit in 256 bit che è l'hash summary del chunk 1.
Ma ora, se uso questi 256 bit come input per un secondo blocco di compressione, che comprime il chunks 2? SHA256 reitera la stessa funzione di compressione.
La F è il cuore dell'hash function, theorem di Merkle-Damgard dimostra che se F è resistente, ovvero soddisfa le 3 proprietà di una funzione hash⇒l'intera costruzione è sicura.(la F non deve essere lineare)
La chiave è trovare una buona compression function, questa prende un input fisso e ridà un output fisso, a questo punto posso usarla iterativamente; l'ultima iterazione mi darà i 256 bit finali.
In che posizione metto il segreto nell'argomento dell'hash fucntion? Prima del messaggio, o dopo, o in altri modi? La posizione del segreto conta ed è importantissima:
es: msg di 1GB, segreto 128bit, poi ho pad+length.Messaggio è noto, vedo il tag = hash(msg,k), vado da 0 a 2^{128} e faccio H(msg,k_x)=?tag. Brute force atk devo computare fino al massimo 2^{128} hash functions.
SHA256 è white box, so che è costruita iterativamente, il msg è sempre lo stesso: computo i primi blocchi che contengono il messaggio e prenderò l'output precomputato (i 256 bit risultanti), ed ora dovrò computare solo l'ultimo pezzo a partire dal precomputato.Non quindi computare N$\cdot$ 2^{128} blocchi, bensì 2^{128} $\Rightarrow$ riduco la complessità d un fattore N.
Se metto il secret all'inizio, posso rompere la forgiability?Posso forgaire un tag valido per un m$^{'}$ scelto da me, partendo da M,tag=H(s,m). Sì è possibile:

triviale forgiare un messaggio autenticato valido m' != m. Estendo msg, che può anche essere insensato, con una parte di plaintext.
Non posso modificare il msg originale ma non è un problema, inoltre lo faccio senza conoscere il segreto: es. aggiungo una transazione alla fine del messaggio.Aggiungo extra chunks, partendo dal MAC code di prima e genero un MAC extended valido.
Questo è un problema ⇒ ho una funzione forte, ma la costruzione rompe tutto (errore tipico della crypto), quindi non si usa mai una funzione non pensata per quel purpose, anche se i purpose sono simili.
La posizione del segreto CONTA TANTISSIMO.
Come fixare il problema:
Hash Based Message Authentication Code (HMAC), che è stata dimostrata essere sicura come l'hash sottostante.
Ho imparato che una secure hash non basta, quindi HMAC aggiunge un modo sicuro di aggiungere segreto nell'hash, non patcho l' hash in se quindi non dipende da come è fatta l'hash.
1996, paper di Bellare, Canetti e Krawczyk con due versioni: crypto e IETR RFC 2104.
Pluggable hash e usando l'HMAC non aumenti il costo computazionale di molto:
$HMAC_k$(M)=H($K^+ \oplus$opad —H($K^+ \oplus$ipad—M)
Il primo pezzo contiene la chiave, i secondo il messaggio. Sembra che sto facendo come prima, ma in realtà sto usando hash del messaggio tra message e secret alla fine. Quindi faccio hash della chiave seguita da hash di message e chiave,come fare 2 volte hash del msg.
Se il segreto K è < della lunghezza di un blocco fai si che sia di pari lunghezza, paddo con zeri, ottengo così K^+.Questo è il primo chunk di SHA256.
Per la sicurezza della costruzione, i due segreti che uso negli hash devono essere diversi: miglior costruzione è la nested MAC construction : H($secret_1$!! H($secret_2$!! msg)). Ma chiedere di usare due segreti sarebbe stato un disatro, quindi per praticità non era conveniente lasciare all'implementatore la scelta dei due segreti.
Soluzione è che produco due segreti dviersi a partire dallo stesso: in entrambi i due risultati flippo bit diversi riseptto all'originale, sembrano quindi due segreti indipendenti(ma non lo sono) ed hanno una distanza larga in termini di bit.
es : k = 01010101, inner: 01010101$\oplus$01011100=K_o, outer: 01010101$\oplus$00110110=K_i (entrambe le sequenze ripetute come serve).
Parto dal msg, aggiungo all'inizio (prefix) K_i, runno hash function: parto da IV e lo unisco a K_i ed ottengo un secret IV.Hash sugli altri chunks, ed ottengo l'inner hash: ho un classico MAC secret prefix, devo mettergli una pezza: prendo l'oueter key K_o e faccio hash del singolo blocco (inner hash + pad).
Ottengo quindi HMAC, che è dimostrato essere una costruzione sicura.
storiella: 2005 md5 broken, tutti gli HMAC tags usavano md5. Thm ti dice che la costruzione è sicura quanto l'hash sottostante: se l'hash è unsecure ⇒dovrebbe rompersi anche il meccanismo di HMAC. 2006: non era ancora stato trovato un atck pratico ad HMAC di md5.
Assunzioni: modello math dell'hash function:

- pseudorandom output
- anticollision property.

Entrando nei dettagli, Bellare si rende conto che non usa mai la proprietà 2 e capisce che HMAC è più sicuro dell'hash fuction, finché la properietà 1 non è violata.
Paper del 2006 su collision resistance NON necessaria.
Messaggi importanti:

- Confidentiality != integrity
- Message authentication with symmetric key
- Reply atck: MAC non è abbastanza, servono nonces e vanno gestite bene.
- Crypto hash functions
- Come includere key nell'hash fuction? Non è triviale, usa HMAC.

9 Gestione dell'accesso remoto: RADIUS

Tool usato nel backend, Remote Authentication Dial In User Service, obsoleto: oggi migliori protocolli(DIAMETER) ma ci sono un sacco di problemi quindi è utile studiarlo.
Posso accedere alla rete usando diverse tecnologie, tutte eterogenee fra loro e largamente distribuite. Gestire la rete con tutte queste tecnologie ed access point: uso server centralizzato, RADIUS server che è incaricato di gestire username e password dell'utente, così da evitare la distribuzione all'interno della rete.
Anche una questione di sicurezza:(di solito in AP: Linux machine con db interni), non lascio username e psw distribuite in giro per la rete.
Devo cambiare l'authentication model: faccio auth con local technology, RADIUS client che counica con l'utene e col server contatta quest'ultimo ed il server ,manda RADIUS response con un si o no a seconda se l'utente può accdere o meno⇒parte più importante.Il client dice quindi all'utente se può entrare o no(l'utente non sa che sta usando RADIUS).

9.1 RADIUS: AAA protocol

3 servizi di solito eseguiti insieme:

- Authentication
- Authorization: da non confondere con Authentication, qui voglio sapere che l'utente ha il permesso di usare il servizio (perché ha pagato o per altri motivi). Posso avere

- authentication senza authorization
- authorization without authentication ed avrei un servizio privacy preserving.
  Letteratura scientifica è ricca di tecniche per farlo, ma nel mondo pratico non molto usato.
- l'intersezione fra i due.

Serve per management

- Accounting: transmitted bytes (quanti GB sto consumando), billing, minuti di telefonate spese etc..
  Segno cosa stai facendo in termini di una risorsa che stai usando.

9.1.1 RADIUS è client-server protocol

Richiesta parte dal client, non confondere il RADIUS client con l'end user, ovvero il NAS: ho end user - NAS- RADIUS client- Server.
Basato su UDP/IP porta 1812, client port è ephemeral. Sistema centralizzato, logicamente centralizzato: in teoria ho un singolo server ma in pratica è ridondato (sennò è single point of failure)
In RADIUS si può usare roaming: se cambio città rispetto a dove sta il server, es. della mia università, dovrei cambiare account, ma quello che accade è che la mia richiesta viene presa dal RADIUS server della città e la inoltra al RADIUS server della mia univeristà, agendo da proxy server.
Architettura complessa, diversi blocchi:

- Server application
- User db: per ogni username ho almeno authentication info, authentication method e authorization attributes
- Client db: clients che possono comunicare col server.
- Accounting db: se RADIUS usato per accounting, deve essere aggiornato in real time, per questo separato dal db utente.Non necessario se si fa solo authentication.

9.1.2 RADIUS security features

Due feature, 1° è per packet authenticated reply: NAS non ha le mie credenziali, le manda al server, atck interecetta il messaggio e risponde con un "sì", il NAS ora vede che l'utente è autenticato.Non devo pter spoofare il msg⇒deve essere autenticato, ed è quello che è stato fatto: si usa shared secret, approccio CHAP-like,ma:

- solo la reply è autenticata
- l'autenticazione è hash based e non HMAC-based

- funzione hash specifica MD5, quando uso una hash function deve essere future-proof, se metto uno specifico crypto algo in un protocollo è male: qualcuno prima o poi lo romperà. Non è semplice andare poi a modificarlo. Il protocollo è una cosa, l'algoritmo di encryption deve essere messo a parte, così da cambiarlo in caso venga violato.

- Secret non truly random, ma low-entropy shared key

2° servizio: user password enrcyptata: se uso PAP, ho la psw in chiaro. Standardizzazione di un meccanismo. Problemi:

- Custom mechanism, non inventare algoritmi per quanto possibile, ma usane uno già esistente. (Non era rotto, però devo considerarlo come possibile vulnerabilità).

- Shared secret key usata anche per l'authentication ⇒NUN SE FA, anche se non è exploitabile è errato, perché se rompi la chiave rompi più di un servizio.

9.1.3 RADIUS authenticated reply concept

End user credentials $\Longrightarrow$ manda le credenziali al NAS, RADIUS client e server hanno uno shared secret che è $\neq$ dalle credenziali del utente. NAS parsa le informazioni e le traduce nel RADIUS language, include le credenziali in un pacchetto RADIUS che è un pacchetto UDP/IP che ha:

- ID field: mi permette di matchare una richiesta con la risposta.

- Authentication field: nonce di forma strana, è una nonce che mando al server così che il server possa creare un reply message (sì, no go-on se servono più informazioni) e possa autenticare il pacchetto, ovvero il pacchetto deve avere un authentication tag. In message authentication includevo il TAG (che era HMAC di K + content), qui ho una cosa analoga: ho la risposta, il tag si costruisce combinando l'ID, il valore random usato come nonce ed il segreto pre-shared.MAC=H(ID,nonce,secret).
  Il reply può anche avere authorization, esempio poter permettere accesso per un tempo limitato.

NAS si tiene in un local db l'associazione ID-nonce(authentication). Faccio un check e se mi torna ⇒ sono sicuro che il messaggio mi è arrivato dal server e so che non può essere replicato perché l'auth è fresh per ogni nuovo handshake.Ora NAS passa l'informazione all'end user. È una sorta di challenge-response:

- la challenge è il request authenticator

- la risposta include anche,una volta validata, il messaggio di risposta.

Formato del pacchetto:
IP header — UDP header — RADIUS packet:

- byte di codice:
  - 1) sì
  - 2) no
  - . . .
  - 3)access challenge: sta per go-on, non inteso come la classica challenge.
- 1 byte di identifier
- 2 byte di length per il pacchetto
- 16 byte di authenticator che deve essere non replyable ⇒ unique. Sono 128 bit ⇒ $2^1$28 possibili authenticator, se fosse realmente truly random, avrei avuto probabilità di collidere proporzionale al birthday paradox (ordine $2^6$4).
- Attributi sono triplette di (type,length,value), ogni tipo corrisponde ad un determinato tipo (username, password, framed-MTU, Callback-number)

Authenticator field: la parte più importante per la sicurezza.Dovrebbe essere unico ed unpredictable per evitare reply attack. Ha due scopi: nell'access request server per authentication mechanism, nella response è sempre di 16 byte ma viene usato per il TAG. TAG è MD5(Code—ID—Length—RequestAuthenticator—Attributes—Secret): qui code è codice di risposta, length è la lunghezza del pacchetto di risposta, attributes sono le triplette. Request Authentication si ottiene dall'access request. Access-request di solito contiene 2 classi di informazioni, uno dell'utente ed uno dell'access service device:

- Username: NAS ha le credenziali, deve mandarle al server
- Password dell'utente
- Identificatore del RADIUS client, NAS-IP o NAS-identifier
- Identificatore della porta a cui l'utente sta accedendo, la NAS-port (se il NAS ha una porta)

Access-reject: o ho fallito l'autenticazione oppure non ho l'autorizzazione (esempio: non ho pagato)
Access challenge è un go-on message: usato quando server vuole che venga fatto altro: ci sono altri protocolli di autenticazione (esempio: EAP-TLS, EAP-TTLS) in cui devo fare più operazioni, che richiedono più messaggi

9.1.4 PPP CHAP support with RADIUS

In una situazione normale di challenge handshake ho user, server: server mi da challenge,rispondo e lui mi dice si o no.
Nel caso di user — NAS — server:

potrei generare un processo simile, ma se faccio questo devo anche mandare il segnale fisico per far capire che l'utente è attivo: overhead grande, devo "svegliare" l'utente, il NAS deve chiedere la challenge al server e così via.
L'utente si sveglia, il NAS genera la quantità random (mi dovrei fidare dell'access point): utente risponde con hash della password e della challenge usando CHAP. Il NAS crea Access-request RADIUS con Username—Risposta della challenge—Chalenge—Servizio....
Or il server può verificare se il client è autentico e decidere se dargli accesso o no, manda RADIUS Access accept. Nel caso di protocollo CHAP non uso access-challenge message, uso solo Access-request.
Vulnerabilità: messaggio del NAS non è autenticato, l'Access Accept non contiene la tripletta di username o psw, è anche vero che la challenge cambia sempre.NAS non può verificare che la challenge era quella vera. Attacco:
prendo il NAS, mando una challenge "1234" e user manda reply"$\alpha\beta\gamma$" NAS manda il pacchetto al server ed ottiene Access Accept.
L'attacker si finge me: prende il pacchetto che ha generato fingendosi me e sostituisce ai campi dell'auth che il NAS gli ha mandato e la sua risposta alla challenge (che è random,tanto non è importante che sia corretta), a quel punto lo invia al server e non è detto che il server faccia un check per vedere se la challenge che il NAS mi ha dato è fresh o no. Attacco al payload del messaggio: rispondo con una coppia di valori precedenti validi.
Dal 1998 anche le richieste diventano autenticate, ma non era una cosa necessaria.
Problema: posso fixarlo? Potrei pensare di autenticare reply e request, ad esempio fare HASH(request, reply).

9.1.5 Password encryption

Se user manda username e psw con PAP: NAS dovrebbe mandare nel RADIUS packet, ma sono in chiaro. La rete in generale, tra il NAS ed il server RADIUS non può essere trustata ⇒ tecnica per criptare la password: predo la psw nativa, la paddo per riempire un blocco da 16 byte, faccio MD5(secret—RequestAuth), risultato sembra una string pseudoranom, quindi uso una tecnica simile allo stream cipher: il keystream non è prodotto da uno PNRG, ma da un hash function. Psw è messa in XOR con il keystream ottenuto dall'MD5(secret—RequestAuth).
Se la psw è più di 16 caratteri: posso dividerla in due blocchi e paddarla con lo stesso valore, ma il segreto è lo stesso e c'è una sola nonce ⇒ padderei due volte con lo stesso keystream. Computo un keystream differente, usando il cipertext precedente e faccio XOR con i due blocchi in cui divido la password.

9.2 RADIUS Security Weakness

Vulnerabile al message sniffing e modification. Access request non è autenticato, il testo è mandato in chiaro quindi ho problemi di privacy.
Soluzione non c'è, devi coprire con un altro protocollo (ad esempio TLS), per l'autenticazione usato EAP (Exstensible Authentication Protocol): protocollo

che permette di scambiare messaggi di autenticazione, difatti nella specifica si trova EAP-TLS, EAP-AKA, ovvero usi un protocollo di autenticazione con dentro un AEP packet exchange.
Message authenticator: ho un pacchetto di richiesta: contiene code—ID—Length—RandomAuth—triple(T,L,V devo aggiungere un TAG, l'idea è di creare una nuova tripla T,L,V in cui il tipo fosse sepcifico per l'autenticazione. Il valore ora è computato con HMAC-MD5, type è 80 e lunghezza 18.
Reply atck: evito reply attack al pacchetto RADIUS di base, ma posso fare reply di una richiesta. Il pacchetto contiene una nonce e l'auth TAG, ma questo è un pacchetto valido, quindi posso replicarlo. Per evitare reply attack di request message, il server deve verificare che la nonce sia fresh. Può o non essere un problema: separa la practical explanation dalla vulnerabilità, deciderà chi implementa se questo è un problema o no.

9.2.1 Dictionary attack to shared secret

Problema grande di RADIUS, cos'è lo shared secret? Segreto che sceglie il network manager, problema è che è difficile che venga inserita una stringa truly random, ma una stringa a low entropy, ricorda inoltre restricted charset. Spesso un singolo segreto è usato per tutta la rete esempio: Fonera, aggregazione di AP a cui si ci può connettere in roaming. Stesso segreto per 100k+ device ed era triviale (tipo Salute! in spagnolo).
Quindi, usare uno shared secret per ogni client (metodo del segreto unico, che conosco solo io e ne faccio HMAC con un identificatore univoco dell'AP).
Possibile fare dictonary attack offline:
intercetto una coppia (request,response), ho tutte le info per fare brute force o dictionary atck e posso fare precomputation perchè il segreto è alla fine nell'MD5 del response packet.
Se richiesta e risposta su due canali diversi, e posso accedere ad esempio solo la request: non serve la coppia. Siccome il segreto è lo stesso mi basta generare uno userID ed una psw arbitrari, so che avrò una risposta con i campi definiti. Prendo la nonce dal request packet, fare nonce con la psw che ho scelto (Chosen Plaintext atck), quindi ho un keystream e posso fare bruteforce con dictionary atk.
Attacco alla password dell'utente: parto dl nome della vittima che voglio, metto psw arbitraria, ottengo unser password attribute e la psw encryptata che è scelta da me, pulisco encrypted password ed ho un keystream valido. Suppongo di voler fare brute force di un utente: faccio trial di psw, ma è lento e verrei bloccato dopo un certo n° attempts. Ma così ho trovato il modo di bypassare il server: mi metto dietro il NAS e spoofo tutta una serie di request (non è detto che il server faccia check che la nonce sia diversa) e faccio dictonary attack.

9.2.2 Poor PRNG implementations

PRNG è una delle parti più importanti in security.Security in RADIUS richiede un Request Authenticator unico e fresh: ho un req auth di 128bit, 16 byte.

esempio, prendo un random generator, lo chiamo rand(), genera 4 byte:

- Lo chiamo 4 volte.
- Lo chiamo una volta e riempio di 0
- Faccio md5() del risultato della chiamata.

Quale meccanismo uso? PRNG ha un periodo, rand() ha un periodo di 2^{32}: periodo è la lunghezza del ciclo affinché non si ripeta lo stesso pattern (in molti casi, per non crypto PRNG il periodo è 2^{n_bit}). Mando il RADIUS packet e l'auth req. non dovrebbe ripetersi. Se faccio merge di più pacchetti: il periodo si accorcia, perché abbiamo preso più valori. In termini di entropia, 2 e 3 sono quasi equivalenti: sono sicuro se la nonce non si ripete, l'approccio 1 ha 2^{30} come periodo, mentre 2-3 avrebbero la stessa sicurezza.
La maggiora parte delle implementazioni di RADIUS usano non crypto PRNG. Se uso PRNG che è buono dal punto di vista statistico, ma può non essere dal punto di vista della sicurezza:

1. Predictability: non devo poter predirre quale sarà il prossimo valore
2. Periodo: prima o poi il generatore si ripete, non voglio short cycles.
3. Random generator garantisce valori unici o ripetuti: ci sono alcuni random generator in cui è possibile garantire che se genero blocchi di dati, questi sono diversi. es: AES ha blocchi che non sono ripetuti

Riguardo al ciclo:
Linear Congruential Generator: $R_{n+1} = (a \cdot R_n + b)$, nel caso di rand a e b scelti in modo che il ciclo sia di 2^{32}, nel caso di questi generatori se conosci un valore li conosci tutti.
Mersenne Twister: 2^{19937}-1 è ciclo "infinito", ma i valori si ripetono.
Se so che i valori si ripetono, al sicurezza è $2^{\frac{N}{2}}$, altrimenti è 2^N.
Ora che so che RADIUS ura poor PRNG, mi aspetto che auth request ripete: stesso problema di WEP, posso avere più o meno predicibilità e penso ai possibili attacchi. Sono tutti reply attack: monitoro un certo n° attempts in cui ho utenti validi, ogni richiesta avrà una risposta con delle nonce. Creo tabella: Auth request nonce—Access accept packet. Access accept packet mi da il permesso di accedere al sistema. Creo dizionario, dopo un po' entro nella rete, NAS gemera una nuova access request che può contenere un numero che già è apparso, faccio reply di una risposta positiva e rispondo ⇒ ho accesso alla rete.
Stesso alla user psw: è encryptata con MD5(segreto, auth), se auth si ripete il keystream è lo stesso ⇒ two time pad e posso romperlo. Creo dizionario di request auth—user psw $\oplus$ MD5(secret, nonce).Raccolgo il cipher text, quando avrò la ripetizione (di uno stesso keystream con una psw diversa), faccio XOR e ottengo lo XOR fra due password e sfruttando la low entropy le ottengo entrambe.
Posso anche spoofare le password, incrementando il dizionario: aiuto un attacco passivo, uso psw finte che conosco ⇒ ottengo la mia psw in XOR con il

keystream, faccio lo XOR con la psw ed ottengo MD5 e quindi il keystream.
Ora se un utente arriva ed il keystream si ripete ottengo la password a gratis.

9.3 Lezione da RADIUS

Whitebox pentesting: alcuni siti usano nell'URL uno SHA256(emailuser, rand()), se provo a loggarmi, faccio brute force per capire qual'è il valore rand() usato: devo creare 2^{32} hash (se rand ha questa periodicità), con 66M hash/sec, 1 min e ho enumerato tutto i possibili valori, ora so che se entra un altro utente dopo di me, posso usare il mio dizionario per prendere il valore successivo.
Cosa ho imparato da RADIUS:

- Do-it-all-in-one non ripaga: un protocollo applicativo non dovrebbe includere sicurezza.
  Come rendo scuro un sistema? Sviluppo un protocollo apposito, come ad esempio TLS, e lo uso per rendere sicuro un protocollo non sicuro.
- AAA protocol non dovrebbe implementare un meccanismo proprio, inoltre non includere algoritmi nell'algoritmo.

Attualmente: DIAMETER per migliorare RADIUS.