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 meccanismo 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

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	$\frac{(1-p)}{2}$

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 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: $1.8\text{x}10^{19}$ 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 $\text{H(X)} = -\sum_i p_i log_2(p_i)$, considero $\text{p} > 0$, inoltre il segno meno serve perché essendo $\text{p} \leq 1$, il log mi da un valore negativo.
La quantità viene misurata in bit. esempio: un coinflip di una moneta equiprobabile ha $\text{H(X)} = -2 \cdot (\frac{1}{2} \cdot log_2(\frac{1}{2})) = 1$.
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: $\text{H(X)} = \text{E[IC(X)}] = \sum_i p_i IC_i = -\sum_i p_i log_2(p_i)$.
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 $\text{N} = 2^b$ possibili outcome allora $\text{b} = log_2(N)$, 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 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 $2^27 \simeq 134$ 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(P[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

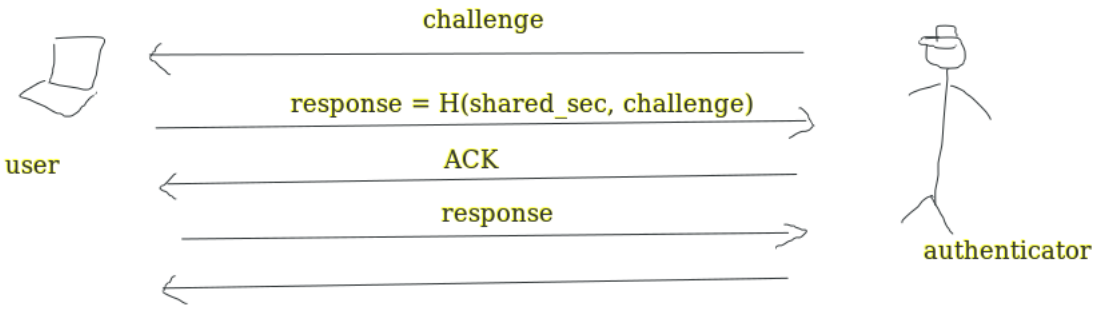
Ho client e server sicuri, c'è il segreto shared su entrambe, non computo P[n] = H(P[n-1]), bensì P[n] = H(segreto, n); ad esempio P[35] = H(secret, 35) e così via.
Anche se ottengo uno dei P[i] non posso ricavare gli altri se la funzione hash è crypto. Inoltre k può essere "infinito", parto da un counter 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'attacker 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

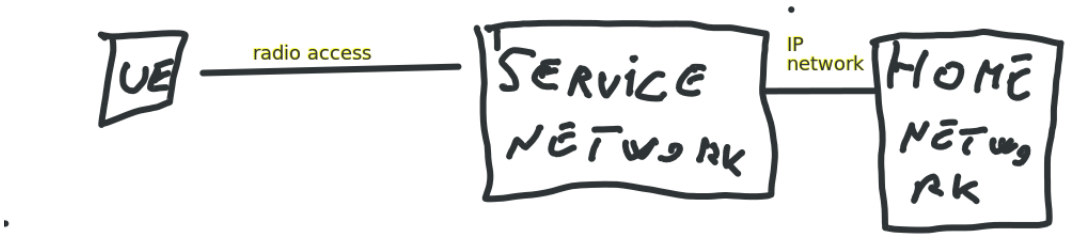
Una nonce è un valore sempre fresco, ovvero ogni volta diverso.
Sono di 3 tipi:

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.
- L'attacco 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 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$01011100=K_i, outer: 01010101$\oplus$00110110=K_o (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^{128} possibili authenticator, se fosse realmente truly random, avrei avuto probabilità di collidere proporzionale al birthday paradox (ordine 2^{64}).
- 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....
Ora 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 nel protocollo.

Attualmente: DIAMETER per migliorare RADIUS.

9.4 AAA evolution: beyond RADIUS

Quando parto da una soluzione, meglio cambiare poco.
RADIUS deployato anni fa, oggi RADIUS è standard de facto per AAA, spesso anche usato in Wi-FI, supporto universale per i device vendors.
Buon protocollo, ma con limitazioni funzionali:

- scalability: quando fu deployato c'era pochi utenti, ma ora sono molti. UDP è unreliable, potrei avere problemi di loss
- diversità nelle tecnologie di accesso: prima dial up, ora Wi-Fi, 3G,4G etc..., devo supportare tutte: type|len|value era non sufficiente, 1 byte di type troppo poco.
  Lista di possibili estensioni: più di 256 combinazioni, servono più byte di type.
- interoperabilità: issue importante, ho un server central, ma non è realmente centralizzato (solo logicamente centralizzato), ho delle repliche ed è distribuito.
  Tutti i server considerati proprietari, quindi difficile avere interoperabilità.

Nota di scalabilità: mando RADIUS request e RADIUS accept una volta per connessione, quindi traffico è irrilevante per la scalabilità. esempio: ho NAS a 48 porte, ogni 20 minuti ho in media una nuova combinazione: $\frac{1}{20}\ minuti^{-1}$ ma $\cdot\ 48 = 2$ call/minuto. Ma se numero di NAS aumenta, tipo a 10000: 400 request/secondo. Dopo access request ho anche accounting request e delivery $\Rightarrow$ traffic può arrivare a diversi Mbps, quindi se scalo può diventare difficile da

gestire, potrei avere problemi di packet loss.
Quando dimensiono un sistema, di solito uso average load, ma non si considerano casi speciali: esempio, ho un crash e il device si reboota. Tutti i device rebootati mandano peek traffic: posso avere molta perdita, RADIUS non scala bene per colpa di UDP, ora serve reliability e quindi TCP o meglio.

9.5 IETF evolution

- Diameter: iniziato nel 1998, ora completato. Attività mosse in DiME(Diameter, Mainteinence, and Extensions WG).
  AP a casa: PPPoE/PPPoA: protocolli per patchare la possibilità di far girare PPP su ethernet o ATM, doveva durare poco, ma attualmente alcuni lo usano ancora.
- RADext: path a RADIUS, usato fino a che Diameter fosse diventato mainstream.

RADIUS: molto lavoro già fatto, pesanemente integrato, standard de facto.
Diameter: protocollo nuovo, più potente e scelta perfetta per nuovi scenari.
Li tengo entrambi: lavoro duplicato ma è conveniente a livello di business, se qualcosa va male in Diameter ho backup che è RADIUS.

9.5.1 DIAMETER

Simile ad """un object-oriented "protocol design" """ (non dirlo alle persone), ho classe base da cui derivo classe derivata. Primo protocollo inventato come OO: DIAMETER non è un AA protocol, ma un protocollo di messaging/signaling generico a lvl applicativo.Definisco il DIAMETER base protocol: ho le primitive per supportare messaging/signaling transport.
Definisco il protocollo per trasportare i messaggi, lo faccio in un altra classe base che è AAA Transport profile(SCTP, TCP-based), deve essere reliable.
Ora derivo le specializzazioni: creo una applicazione DIAMETER differente per ogni uso necessario:

- DIAMETER mobile IPv4 app: per muovere IP
- DIAMETER NAS app: questo è per il purpose di RADIUS.
- DIAMETER credit control app
- DIAMETER EAP app
- DIAMETER SIP app

Nella base: tecniche per keep alive server, load balance etc.., eredito le proprietà e specializzo per lo specifico porpuse dell'applicazione.

9.5.2 DIAMETER improvements

SCTP: perché TCP può non essere buono.
esempio: ho un NAS che deve parlare col server AAA. Quando arriva connessione, devo settare comunicazione, NAS manda request al server. La voglio reliable: soluzione base è setuppare TPC connection per ogni connessione: devo fare il 3-way handshake, mando il pacchetto e poi aspetto ack e mando il FIN. Tutto st'accrocco per un singolo pacchetto.
Non faccio TCP conn per ogni connessione. Uso una singola connesisone TCP per gestire tutti pacchetti: implemento multithreaded server: ho due thread, uno di questi si blocca. Vorrei poter gestire il secondo pacchetto, ma TCP fa consegna ordinata, quindi non posso creare gap nel protocollo: TCP mi da tutto in ordine, è uno dei goal. Ho un flow multiplo embeddato in una singola connessione TCP e quindi in un singolo flow.
Vorrei una connessione reliable che porta stream differenti, e vorrei un protocollo che garantisce che tutti i pacchetti nello stream siano letti nel giusto ordine, ma non di avere ordine fra gli stream: se il primo stream si blocca, vorrei bypassarlo e leggere il secondo. Effetto Head of the line blocking: anche se ho uno switch ad alta capacità, l'effetto impatta sulle performance.
Secondo problema di TCP: ho un NAS, per reliability ho una interfaccia ethernet, ma posso averne una di backup, ad esempio di backup (in altra tecnologia) così da garantire continuità. Ma l'IP address delle due connessioni è diverso e TCP socket usa la 4-pla: quindi se link fallisce devo inizializzare una nuova connessione (MPTCP lo risolve), vorrei supportare multi-homing: manage più IP address.
SCTP (Stream Control Transport Protocol) da queste due garanzie, protocollo migliore per canale di signaling dove trasmetto più flussi di dati.
storiella: perché se funziona così bene uso ancora TCP: chicken/egg problem. Standardizzazione bloccata dopo gli anni 2000, ad esempio anche IPv6. Problema dell'Internet Ossification: 1980/1990, con design spirit di Inernet che era End-to-end principle: nella rete telefonica originale, intelligenza nei device centrali, edge stupidi, in Internet la maggior parte dell'intelligenza ai bordi.
1995-1998-2000: web, avvento di device più intelligente, NAT primo device necessario per far fronte al lack di device alla rete, ma poi firewall, media converter ed altra roba messa sopra. Tutta una serie di middle boxes. Anche device come TCP accelerators, performance enanchements, etc... "intelligent devices". Arriva un nuovo protocollo: SCTP, comincio a connettere siti distanti, ma siccome traffico gira su questi device che lo reputano non noto, viene bloccato.
Servono middle boxes per supportarlo, ma i produttori lo fanno solo se l'evidenza mostra che è usato, ma come cazzo ti mostro che è usato se mi blocchi il traffico.
Ora software networking: middleboxes diventano software e sono controllabili, 5G è rete softwarizzata.

- Reliable transport: uso SCTP, senno TCP se non posso
- Standardizzazione in caso di errore: se server fallisce, come migrare verso

altro server. DIAMETER: standard per scoprire queste situazioni:

  - duplicate detection
  - controllo di ritrasmissioni a livello applicativo
  - fallimento di peer. DIAMETER è protcol p2p, server può startare comunicazione esplicitamente con NAS.

- pacchetti PING-like per testare se il device è attivo o no.
- Estensione dei limiti funzionali: header di RADIUS era corto, quello di DIAMETER è più complesso:
  - length di 3 byte
  - 3 byte comando ma anche ID per la specifica applicazione.
    Id del pacchetto serve per matchare request-response, lo scenario nel mondo reale non è solo point-to-point: ho multi-hop, ogni pacchetto averà un ID specifico. Non so se voglio matchare comunicazione globale o locale, quindi DIAMETER introduce due identificatori:
    * Hop-by-hop ID
    * End-to-end ID
  - altri flag: NAS può rispondere o far partire la comunicazione, sono in p2p: devo identificare se pacchetto è request o response, uso flag R, flag P sta per proxable e permette di specificare se il pacchetto può essere modificato da un proxy, flag E: messaggio di errore, flag T: messaggio potenzialmente ritrasmesso. esempio:
    mando un pacchetto, non ricevo answare e retx. Ricevo risposta: se server era bloccato temporaneamente, potrebbe rispondere di nuovo: uso T flag, così da darti avvertimento.
  - AVPs: i vecchi T|L|V triplets, ora chiamati attribute,value,pairs. Codice , lunghezza e attributo ma anche altro: metto vendor ID che dice che il linguaggio non è di DIAMETER, ma è customizzato.4 byte di AVP code, perché uno era troppo poco.
    Flags:
    * V: vendor specific
    * M: sono NAS e supporto DIAMETER v4.2.1, server DIAMETER v3.9.8, NAS vuole usare un attributo della nuova versione, che server non ha. Ricevo packet, con attributo che non comprendo: se l'attributo è importante, non posso skipparlo. Conviene droppare packet e dire al NAS di non aver capito. M serve per dire di rimandare indietro, perché le info non comprese sono mandatory. Risolvo interoperabilità.
    * P: se c'è encryption o no
  - ho il campo dati

Mangement di intermediate entities: posso usare RADIUS agent per mandare relayed message,ma non c'era supporto al romaning.
Non devo solo fare roaming data, ma anche routing: se mi collego ad un peer distante, quello deve fare route al server del mio paese. 3 cose strandardizzate:

- Nessuno agent intermedio
- Relay agent: sono a Roma, ma vengo dall'univeristà di Parigi, RADIUS server di Roma riceve la request, relay agent guarda al realm, ovvero il dominio di registrazione e nella sua routing table sa di dover mandare la richiesta al server RADIUS di Parigi.
- Proxy agent: simile al relay, ma assumo che può modificare il messaggio: se ho un messaggio con TAG integrità, se uso proxy e modifica ⇒ ho rotto tutto, è un MITM attack (proxy non ha la chiave per modificare il messaggio).

Eduroam: sistema che permette di roammare lungo le università confederate con eduroam e non richiede di avere credenziali specifiche. Cosa fa un utente che sta a Tor Vergata e vuole accedere, ma è di Malta: server parsa l'address e mi manda a Malta.
M se le confederazioni sono molte: che succede se il server di Malta è va giù e viene sostituito con nuovo server che ha cambiato ip? Deve comunicarlo a tutti gli altri, anche se aggiungo un server. C'è management nightmare: se uso relay agents o proxy agents c'è problema: le routing table sono embeddate, soluzione: centralized controller che tiene le routing tables. Tizio di Malta entra da Tor Vergata, va a server di Tor Vergata e server prima di inoltrare richiesta, chiede l'ip di Malta al centralized server: il dato rimane nel RADIUS server di Roma, ma il controllo è demandato al controller,posso usare trick di caching. Separazione fra controllo e dati ⇒ redirect agent: gestisce solo le routing table, ora le routing table non sono embeddate nell relay agent, quindi ho le due operazioni separate.
Questo rende ad esempio possibile number portability

10 Transport Layer Security (secure socket layer)

Analisi approfondita di TLS(/SSL) (il più famoso insieme a , SSH ed IPsec).
Disegnato per sicurezza di altri protocolli, ma c'erano problemi anche qui.2 obbiettivi: analisi dedicata di TLS nei dettagli, capire come fare design di un protocollo di sicurezza long-to-live.

10.1 Introduzione a TLS

Background storico di SSL/TLS:

- 1993, esplosione del web: Mozilla rilascia il primo browser e web diventa servizio reale. Subito dopo ciò, si ci rende conto che la sicurezza poteva diventare un problema cruciale.

- 1995: SSL v2 integrato in netscape 1.1, ma fu rotto quasi subito dopo la release.
- Capiti gli errori di SSL v2, c'è SSL v3. Approccio molto ben fatto in principio che è attulamente la base della versione attuale di TLS (oggi SSL v3.4 -¿ TLS v 1.3). HTTPS = SSL = TLS: SSL era il nome commerciale originale, era proprietario, che fu poi standardizzato da IETF e il nome fu cambiato in TLS, in modo particolare TLS v1.0 = TLS v3.1.
- 3 grandi momenti:
  - TLS v1.1: 2006, problema serio, perché TLS era stato disegnato pensando al web security, ma così stai facendo un'assunzione implicita sul protocollo di trasporto che usi ovvero TCP. Viene fuori che TLS usa le assunzioni di trasporto reliable, quindi se messo su protocollo unreliable non funziona più.
  - DTLS: standardizzato in parallelo a TLS nel 2006, versione di TLS adattata per girare un un transport protocol non reliable: preso TLS con pro e contro e modificato un minimo per farlo girare su un protocollo unreliable.
  - TLS v1.2: la versione usata oggi,anche se ce n'è una nuova. Corretto errore originale di TLS: protocolli vs algoritmi, il protocollo non dovrebbe contenere nessun crypto algorithm hardcoded. Se l'algoritmo viene rotto ⇒ il protocollo è rotto. Devi disaccoppiare per avere un long-time-live protocol.
    Nella parte pseudorandom del protocollo si dipendeva da MD5/SHA-1 (rotti dal 205 in poi). TLS v1.2 disaccoppia e supporta algoritmi per authenticated encryption.
- TLS v1.3: modifiche importanti, solo ciphers AEAD accettati, primo protocollo della famiglia TLS che garantisce la perfect forward secrecy.

10.2 SSL/TLS: layered overview

I due protocolli principali in newtowking runnanno a livelli diversi: IPsec garantisce sicurezza a livello 3 e runna sopra IP, quindi prima del layer trasporto e protegge anche il protocollo IP. Con IP sec ho anche integrity protection per pacchetti IP. Non ho nozione dello specifico layer di trasporto, sono il payload del pacchetto IPsec.
Mentre TLS ha obbiettivi diversi: nato per proteggere il servizio web, idea era prendi HTTP, fallo girare su un protocollo di sicurezza che era intermedio tra lvl 5 e lvl 4: TLS o DTLS. TLS quindi gira sopra il protocollo di trasporto ma non protegge il protocollo di trasporto. TCP rimane non protetto: mentre IP può garantire integrità del pacchetto IP (se ben configurato), TLS non lo fa: il pacchetto TCP può essere modificato. Un attacker può modificare la parte del TCP header, mentre la parte in cui c'è confidenzialità ed integrità è il messaggio HTTP (che è protetto da TLS).

Quello che TLS fa esattamente è proteggere esattamente il payload di TLS, ma non il resto. esempio:
VPN spesso create con IPsec, ma se uso open VPN:
ho il mio indirizzo IP, uso TCP/UDP ed uso TLS, poi di nuovo IP TCP applicativo. Tunneling: l'intera pila poggia su una pipe virtuale.
TLS esempio perfetto di protocollo di livello di sessione (pila OSI).
TLS non è necessariamente limitato al layer di trasporto, usato anche per altri motivi, anche per esempio EAP-TLS: autenticazione basata su TLS e protezione d'integrità su EAP v1.2. EAP-TTLS: creo tunnel su TLS e dentro scambio le credenziali, mentre in AEP-TLS uso TLS per scambiare certificati.

10.2.1 Application support

Socket per HTTP: creo connessione in cui specifico Ip addr, port number. N° porta è l'identificatore del servizio che lancio sulla mia macchina. (HTTPd su porta 80).
Server avere un approccio per livelli, dove ogni livello vede solo quello sopra e quello sotto e non più di uno:
vorrei usare HTTPS, potrei aprire la porta 80, ma ora potrei usare TLS e specificare che TLS usi la porta 123. Approccio corretto per fare incapsulamento sarebbe: TCP specifica che nel pacchetto ha un pacchetto TCP, dentro TLS specifico nell'header che l'application number è 80. TCP $->$ TLS $->$ HTTP. non fu fatto per ragioni storiche: nel 1993 nessuno pensò di usare TLS al di fuori del web. Approccio: ho HTTTP, se voglio usarlo uso porta 80, se invece voglio usarlo tramite TLS usa la porta 443, c'era un numero di porta dedicato. Perché una brutta idea: se uso per un servizio differente? TLS non ha un numero di porta specifico,se voglio usare ad esempio POP3, che usa porta 110 direttamente su TCP ma su TLS non posso usare la stessa: ora ne devo standardizzare una nuova; porta 995 che è SPOP3. Devo standardizzare una nuova porta ogni volta che voglio supportare un nuovo protocollo di livello 5: Duplication of port, non secure vs secure.

10.2.2 Confronto con IPsec

Tutto più clean: ho il classico header IP, ho un field di protocol che dice cosa c'è nel pacchetto: 6 per TCP, 71 per UDP, quindi IPsec standardizza 51 (ci sarà un header aggiuntivo per IPsec).Quindi, se ho IPsec connection e vedo pacchetto IP non so che protocollo sto usando: vedo protocol 51, l'avversario che guarda al mio traffico non sa che protocollo sto usando a lvl 4/5, meglio dal punto di vista della privacy. Ho un servizio detto traffic flow confidentiality: un avversario che guarda il flow di pacchetti non deve poter sapere cosa fai in termini di protocollo o applicazione sto usando (se è criptato); in TLS questa cosa non è vera: ho protezione sul payload ma non sull'header TCP da cui posso carpire le informazioni che mi servono.

10.3 Obbiettivi di TLS

Fa due cose allo stesso tempo

- Nel settare una connessione sicura (secure session in TLS) tra due end devo fare due cose: creare la connessione in se, faccio signal e setup della sessione sicura. Fatta dalla TLS handshake:
  - Quale algoritmo encryption uso, devo essere d'accordo con l'altro end. TLS non specifica o hardcoda un algoritmo specifico. per encryption
  - crypto keys che verranno usate: in TLS v1.2 cambiano in ogni connessione (asymmetric crypto). Qui i segreti sono sharati on the fly con tecniche avanzate.
  - Voglio essere sicuro che sto parlando con il server corretto: authentication.
- La seconda fase prevede il trasferimento dati: ho una chiave, ho definito il crypto algorithm, ho definito algoritmo di integrità, prendo TCP segments e li encrypto per trasferirli all'altro end.

Sarebbe meglio avere questi due due fasi implementate in due protocolli diversi: non dovrei fare le operazioni allo stesso tempo. Setto un'associazione sicura e la uso quando ne ho bisogno: questo viene fatto da IPsec, non proteggo traffico on demand ma è persistent. Approccio di TLS meno flessibile: in ogni connessione devo fare entrambe le fasi.
esempio: applicazione di TLS su IoT: voglio un protocollo come TLS ma il sensor device va a batterie. Se devi far girare un'istanza di TLS per ogni connessione, la batteria va giù. Split di TLS in due pezzi, separata la parte dell'handshake dalla parte dell'invio: creo sessione una volta e trasferisco i dati. In TLS v1.3 ha risolto il problema.

10.4 Protocol stack TLS

TLS gira su TCP/UDP. TLS wrappa i dati in un TLS Record Protocol, formato per scambiare dati. Sopra TLS posso far girare qualsiasi protocollo applicativo TLS non lo sa e non gli interessa, lo sa TCP per via della porta usata), nel TLS RP avrò: dati applicativi, c'è handshake protocol (che usa la stessa struttura del TLS RP), Alert protocol e change Cipher protocol; anche protocolli per gestione errore.

10.5 TLS Record Protocol

TLS v1.3 cambia parecchio. Voglio capire come di fa il design di un protocollo, quindi capire bene gli errori fatti.

10.5.1 Record Protocol operation

Preso dallo standard TLS v1.0, e vale fino a TLS v1.2: application data, taglio in frammenti, comprimo per usare meno banda. Aggiungo integrità (MAC), encrypto e poi mando il pacchetto. 2 grossi errori: da un punto di vista del protocollo sembra tutto ok, ma dal punto di vista della sicurezza ci sono. Gli errori sono usciti fuori dopo, specialmente la compressione fu usata per attacco CRIME (prima compressione e poi encryption è deadly) e nel fatto di fare prima integrity e poi encryption: cambia l'ordine? Irrilevante dal punto di vista del protocollo, am l'ordine conta: problema scoperto nel 2002, fixato, rotto di nuovo poi di nuovo rotto e poi di nuovo rotto e ancora e ancora rotto e rotto e ROTTO: padding oracle attack. Alla fine, TLS rende impossibile fare MAC separato da encryption, ma solo AEAD.

10.6 Compression

Application data sono verbose: ASCII, XML, HTML etc... Quindi in TLS penso alla compressione dei dati. Devo farlo prima dell'encryption: perché non posso fare encryption prima di compressione? Entropia: qualcosa può essere compressa solo se l'entropia è minore di 1 bit. Un buon encryption scheme è pseudo-random string con la stessa entropia $\Rightarrow$ non ha senso comprimere dopo. Lossless compression: introdotta da SSLv2. Considerato in TLS v1.0 ma non specificato: unico algoritmo standardizzato era null compression method. 2004: supporto per la compressione in TLS, nei successivi anni la compressione comincia ad essere supportata da vari browser fino all'uscita del CRIME attack: combinazione di compressione ed encryption è deadly (aiuta a decriptare).
Possibile solo usare HMAC, l'hash function è scelta dall'utente. La chiave è simmetrica ma non pre-shared bensì generata dinamicamente.

10.7 Encryption

Prendo il blocco dati, li comprimo, aggiungo MAC ed ora lo encrypto. Posso usare sia stream cipher che block cipher.
Algoritmo negoziato durante l'handshake, l'algoritmo non può aumentare la taglia di più di 1024 byte (ma non succede quasi mai, i blocchi sono di qualche decina di byte).
Infine ho il plai text compresso, il MAC tag, encrypto tutto ed aggiungo header:

- Content type che informazioni ci sono: handshake messgage, application meassge o alert
- Major version: 3.1 per TLS
- Minor version: 3.1 for TLS
- Compression length: lunghezza della compressione.

Cosa manca? Reply attack? Integrità non garantisce da reply attack senza una nonce all'interno. HMAC è funzione del segreto e del message content, poi ho enrcyption e se uso encryption deterministica o l'ho disattivata (non è mandatory). Ma posso fare un reply attack, manca la nonce: può essere vulnerabile a reply attack.
Perché non c'è un sequence number? Perché è implicito: quando faccio partire una TLS connection e dico che questo è il pacchetto 0, girando su TCP sono sicuro che i pacchetti arriveranno in sequenza ⇒ non serve includere il n° sequenza.
Quando computo HMAC: prendo i dati, includo header di TLS ed un numero di sequenza, che non trasmetti: il mio receiver saprà qual'è perché il traffico passa per TCP.
Ma ora c'è un problema: TLS funziona solo se uso davvero TCP e quindi per UDP si romperebbe la sicurezza. Per questo in DTLS è stato necessario ristandardizzare l'header, aggiungendo 8 bytes di seq num.

10.8 More insights on encryption+authentication

Come combinare encryption e MAC.3 possibilità:

- TLS: MAC then ENCRYPT: prendo i dati in plaintext (compressi ma dal nostro punto di vista è paintext). Aggiungo il MAC computato sui dati e poi uso encryption su MAC+dati.
- IPsec: ENCRYPT then MAC: prendo i dati, li ecnrypta e poi computa il MAC sui dati encrypted.
- SSH: ENCRYPT and MAC: prende i dati e su un lato fa MAC sui dati ed encrypta solo i dati. Argomento fallace è perché criptare il MAC, che è già una trasformazione crypto.

Problema: quale delle costruzioni è la migliore, ovvero assumo che uso un encryption scheme semantically secure (IND-CPA) ed un MAC sicuro (unforgiable), quindi MAC perfetto e dovrei dimostrare che con questi combinati ho ancora le proprietà originali.
Perché la combinazione non è più semantically secure (no IND-CPA): ragiono su un esempio con SSH:
in un plaintext-ciphertext so che posso avere A o B. Do un pezzo di cipher e ti dico di scoprire quale lettera è. Ho la possibilità di scoprirlo con un coin-flip. Ora encrypto A = α ed aggiungo il MAC di A che è x12; poi encrypto B = β ed il suo mac y34.
Non so cosa c'è nel cipher, ma il MAC lo vedo: coin-flip diventa deterministico. Il MAC di A o di B è deterministico, quindi si ripeterà. SSH quindi rompe la semantic security.
La strada giusta è quella di IPsec, quella di TLS è sbagliata.
Padding oracle attacks: iniziano nel 2002 e finiscono nel 2016

10.9 Attacchi a TLS

10.9.1 Background su block ciphers

Differisce dallo stream cipher per quello che riguarda l'operazione di encryption. Prendo i dati e li spezzo in blocchi, applico una pseudorandom permutation che trasforma il blocco in ciphertext. Trasformazione deve essere reversibile: se $PT_1 \neq PT_2$ anche i cipher lo sono.
Ma lo stesso plaintext viene cifrato con allo stesso ciphertext, quindi devo anche avere un modo di combinare il plaintext.
Come non farlo: applico solo sa trasformazione (electronic code book, ECB) $\Rightarrow$ sbagliatissimo: non ho semantic security. Se si ripete il plaintext, si ripete anche il plaintext.
CBC encryption (cipher block chaining): se m[0] = m[2] allora se li cifro senza fare nulla ho lo stesso cyper text. Perché non mischiare i blocchi con altri dati: prendo $PT_1 \oplus IV_1$, $PT_2 \oplus IV_2$ etc... e trasferisco i CT con i rispettivi IV. Genero un random IV ad ogni valore, ma il problema è che renderebbe ciphertext il doppio (IV deve avere la stessa taglia del plaintext). Cipher block chaining: prende IV (32 bit in SHA-256), faccio XOR con il messaggio m[0] ed ora faccio PRP con cui produco il ciphertext.
A questo punto uso c[0] come inizializzazione del prossimo blocco.

10.9.2 CBC padding

Voglio fare encryption usando block cipher: se i dati sono di 8 byte, il campo dati + MAC non è detto che sia di taglia giusta (può non essere multiplo di 2). Quindi se uso CBC, TLS deve paddare i dati: aggiungo qualcosa per riempire il blocco, che poi dovrebbe essere rimosso. Idea di TLS: usa l'ultimo byte dice quanto è la taglia del pad, se è 0 vuol dire che non ce n'è pad. Inoltre, se uso ad esempio 1 byte: lunghezza sarà = 1 ed anche il byte in più sarà un 1. (se ne uso 10: metto 10 byte di pad + una lunghezza = 10 e tutti i byte avranno valore 10).
Anche se la lunghezza del blocco non ha bisogno di padding, ormai mi aspetto che l'ultimo byte sia la length, quindi devo aggiungere comunque i byte di paddin+len: PKCS specification per il padding, posso estendere il padding fino a 255 byte.
Quindi: ho dato è MAC, nell'ultimo byte metto la lunghezza del padding ed aggiungo il pad.
Proprietà: come fa un server a decriptare? Mando un dato + MAC + padding + length, encrypto e mando al server. Il server lo riceve e deve decryptare: se ad esempio usa CBC ed usa AES (16 byte) ed ho 48 byte (o comunque multipli okay)di lunghezza sono ok, altrimenti penso ci sia qualcosa di sbagliato, mando messaggio di errore fatal perché la decryption fallisce.
Se invece la lunghezza è ok decrypto e riottengo il pattern di partenza: ultimo byte mi dice quanti byte di padding ci sono. Se ricevo un pacchetto in cui i byte paddati non coincidono con il byte di length(es: length = 2 e bytes 7 ed 1), so che qualcosa è andata male, la decryption fallisce.Ora so che i byte rimanenti

sono MAC è dati e so quanti byte compongono i singoli campi e posso checkare il pacchetto: se non è ok mando un messaggio che dice che il MAC check è fallito (bad_record_mac).
Perfetto se stessi considerando solo di networking: se c'è un errore devo farlo sapere all'altro end. In security questo non si fa: questo meccanismo è la base di un attacco, posso decryptare il pacchetto usando solo questi error codes ⇒ padding oracle attack.

10.9.3 Padding oracle attack

Sfrutta le scelte sbagliate nel protocollo. Scoperto nel 2002, fa capire perché l'approccio del MAC then ECNRYPT è sbagliato. Abbiamo il padding, necessario quando si usa un block cipher, il blocco deve essere un multiplo del block size e spesso questo non avviene quindi paddo con dei bytes extra. Riservo sempre l'ultimo byte per la lunghezza del padding (quanti bytes extra ci sono).
In crypto quando qualcosa fallisce NON si spiega la ragione: l'attacker può usarlo per decriptare l'intero messaggio.
L'attacco: funziona se viene usato CBC di qualunque tipo (anche il migliore come AES-CBC che oggi è considerato sicuro).
Ricevo un messaggio in ciphertext: IV + c[0]....[n]. Il mio obiettivo è decriptare un singolo blocco, per esempio c[1]. Assumo che l'attacker può fare Chosen Cyphertext Attack: l'attacker può forgiare ogni ciphertext arbitrario e mandarlo al server e vedere la risposta. È un attacco molto realistico: se vedo una connessione e dei messaggi, dopo il msg n° 2 posso creare il msg n° 3 ed inviarlo. Quasi sempre il msg n° 3 che è inventato non sarà corretto, quindi la connessione si interromperà ma posso vedere la risposta.
Voglio decriptare c[1], mando un ciphertext arbitrario e mi aspetto decyption failed o Bad MAC. Ma cosa vuol dire bad MAC: attacker ha selezionato un cyphertext random, es 93142197 e questo è stato decriptato lato server e la decrypt è meaningless es 19123111 ma se ricevo bad_record_mac ho informazioni sul plaintext che ho inviato: la parte finale è 0, o 11 o 222 quindi so che gli ultimi bytes sono un padding corretto (se ricevessi un bad encryption non lo saprei).
Uso un oracolo, che mi dice se il padding è ok o sbagliato:

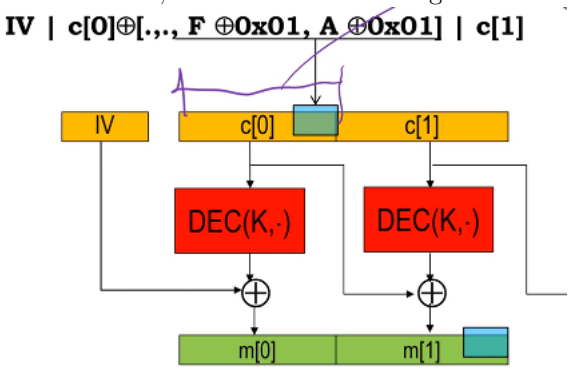
- Padding Ok è bad MAC
- padding wrong è decryption failed

In CBC prendo il plaintext, lo choppo in blocchi, uso IV col primo blocco m[0] e lo metto in XOR. Poi lo passo al PRP ed uso il risultato per m[1].
In decryption faccio l'opposto: ricevo IV, prendo c[0] applico PRP^{-1} e faccio XOR con IV ed ho m[0]. Poi prendo c[1], faccio decrypt e metto in XOR con c[0] ed ottengo m[1] e così via.
I bit di c[1] sono mischiati, quindi se cambio dei bit ho risultati impredicibili. Ma il fatto che uso c[0] nello XOR con c[1] senza passare per non linear transformation è la chiave per l'attacco.

Voglio scoprire un messaggio in plaintext: vedo un messaggio legittimo IV c[0] c[1]....
Levo la parte del messaggio che non mi interessa, faccio si che c[1] contenga l'ultimo byte del pacchetto e lo mando al server. Posso indovinare se l'ultima lettera è A?
Prendo c[0] $\oplus$ A $\oplus$ 0.Il messaggio va al server e farà i check:

- È di taglia giusta, perché ho tolto blocchi interi
- Posso decryptare? Sì
- Suppongo che il contenuto del messaggio fosse: Ciao I am Flavia, sto modificando l'ultimo byte facendo XOR con A e 0. Quindi l'ultimo byte del plaintext è 0 (quello del blocco affianco).
- La del server è bad record MAC: il messaggio è valido perché termina con 0, quindi lo rimuove e checka se la parte del messaggio è hashing corretto. Non lo è, però ho un'informazione e riguardo l'ultima lettera.

Per due lettere, il meccanismo è analogo:



Prendo messaggio originale, lo spezzo così che l'ultimo blocco è quello che attacco, faccio XOR del blocco prima tra l'ultimo byte, il byte che volgio testare ed il byte 0x00..0 così che se il guess è corretto il byte diventa 0 e ricevo bad MAC, altrimenti ricevo decrypt failed. È un test su una singola lettera nel plaintext.
Se lo ripeto più volte: ho indovinato l'ultima lettera, posso testare la penultima: guesso se termina con FA. Prendo c[1] $\oplus$ FA $\oplus$ 01 01.
Sta volta uso 01 01 quindi di nuovo avrò che il padding è ok se il guessing è ok.
Quanto è lungo l'attacco ad un intero blocco: meglio di un brute force attack, in brute force avrei 256^8 se avessi 8 byte di blocco, qui passo a 256x8 = 2^{11}.
Ma dopo il primo tentativo, la connessione si interrompe.
esercizio: ho un cipher f1 aa 11 04 —— 34 35 f1 20 —— 11 01 9c 01 —— ac c3 83 02 —— 65 61 fb 08 —— 91 11 5f 10. Voglio scoprire il 16° byte con padding oracle attack è 0x0f = 00001111.
Per prima cosa devo rimuovere la parte che non mi interessa: il byte che attaco deve essere l'ultimo. Per modificare il byte: ho 11 01 9c 00000001 —— ac c3

83 02 e gli ultimi sono cambiati da PRP^{-1}, quindi devo cambiare l'ultimo byte del blocco precedente così da avere effetto sul byte del blocco affianco.
Ora mi verrà che l'ultimo byte è 00000..0 e quindi potrò vedere il mio guess.
Quindi: se voglio scoprire un byte:

- Choppo il messaggio lasciando il blocco di cui voglio indovinare il byte alla fine
- Faccio XOR del ultimo byte del blocco precedente con la lettera che voglio indovinare e con 0x00.
- Quando verrà decriptato: se il byte è corretto, avrò che l'ultimo byte che sto cerando di indovinare è un 0 e quindi verrà visto come un padding. Verrà scartato ed il server cercherà di checkare l'integrità. Questo fallirà ed avrò bad MAC e quindi saprò ch ci ho preso. Se invece ricevo dab decryption, ho sbagliato il guess.

Se voglio indovinare più byte devo andare linearmente (0, 11, 222, 3333 etc...).
Come faccio a prevenire l'attacco: ho standardizzato due messaggi per due situazioni diverse. Correzione: restituisco una sola risposta, triviale (la maggior parte delle implementazioni corresse mandando sempre bad mac).
Se la decriptionva a buon fine, se check fallisce restituisce bad MAC che però è il vero bad MAC. Sono sicuro che attacker non può capirlo? Uso il tempo come indicazione per capire se il msg di errore è avvenuto a livello di encryption o di vero bad MAC. Side channels: problemi, come in questo caso per colpa del tempo.
LAN, quindi rete eth: la risposta può essere diversa per veri bad MAC o decryption failed. Mandando pochi messaggi è possibile discriminare quali dei due errori ho avuto.
Il programma sarebbe sequenziale, mentre un crypto programmer cercherebbe di fare entrambi i check in parallelo. (se faccio dei check e setto un flag di conseguenza e poi passo ad un compiler con flag -03 che però non capisce che che sto facendo programma per sicurezza).
Tutte le implementazioni corrette:

- TLS 1.2: valida il MAC anche in caso il padding fallisca
- Ma in che caso i dati vengono validati?Se messaggio è formattato bene, so quanto devo paddare, ma dopo decryption non so quanti dati devono essere validati. Se padding è di 2 bytes il msg finisce ad un certo punto, ma al variare del padding ho taglie di messaggi diverse. Decisioni complesse, quindi quanti dati valido? TLS 1.2 valida tutto, considera come max size msg, ovvero anche se padding fallisce considera msg — MAC e ultimo byte
- Kenny Patterson: HMAC, il tempo di computazione non è lineare, ma a scalini. Quando computo HMAC ogni blocco è computato con una f. di compressione. Posso misurare Δt tra i le varie lunghezze di blocchi? Attacco lucky thirteen: c'era ancora time channel

- POODLE: altro attacco, 2014
- 2015: altro subtle time channel
- 2016 attacco alla patch per fixare l'attacco lucky 13.

10.9.4 Lezioni

Problema è sempre quello: prima decryption e poi computazione MAC. Prima MAC e poi ENCRYPT protegge da questi attacchi: non posso più fare CCA, prima checko l'integrità e solo poi vado avanti. Ma se l'integrità non è verificata non ho modo di fare CCA; l'ordine delle operazioni conta.
Quando si ha a che fare con la crittografia bisogna stare attenti: non va in conflitto con security by obscurity, quello dice di usare open algorithm. Ma quando sono a runtime devo stare attento se quando differenzio una risposta, può esserci un oracolo che le differenzia e da delle informazioni all'attacker.
Non implementare crypto da se, attenzione ai side channel attacks.
Usa librerie cryto fatte da gente esperta: openSSL, LIBSODIUM, etc... (stesso vale per l'hardware).
L'attacco è pratico? Posso forgiare un cipher al server e dopo la riposta ho un errore fatale, che fa si che la connessione si rompa. Devo mandare 2048 messaggi, la prossima connessione partirà con una chiave nuova. L'attacco sembra non pratico, ma in qualche caso specifico può diventarlo. Telefono checka le e-mail, probabilmente ogni 5 min (perché si usa IMAP): setta connessione TLS, ed ogni volta la passoword e l'id sono nella stessa posizione. Sono attacker, conosco il formato di IMAP e voglio conoscere la password: so che sta nel blocco 3. Vedo il primo messaggio, provo con il primo messaggio, anche se fallisco dopo 5 min posso riprovare. L'informazione è strutturata sempre nella stessa maniera: posso fare più trial.
Dimostrato nel 2003 che era possibile intercettare e raggruppare le psw degli utenti tramite IMAP (usarono euristiche come dictionary atk etc..)
Quindi la vulnerabilità è estremamente pratica.
Cryptography Doom Principle (Moxie Marklinspike, crittographer che ha standardizzato la sicurezza di WA): se devi fare ogni operazione crittografica prima di verificare il MAC sul messaggio ricevuto, in qualche modo sei condannato.
TLS 1.3 standardizza AEAD, visto che non era possibile cambiare l'ordine delle operazioni (modifiche sw troppo importanti), non più MAC then ENCRYPT.

10.10 Block ciphers

L'obbiettivo è quello di generalizzare un substitution cipher: esempio sostituisco una lettera con un altra (vedi esempio de Giulio Cesare). Definisco un blocco: input è plaintext di taglia nota (in AES per esempio è 128 bit) e l'ouput è una stringa di altri 128 bit, diversa. Rimpiazzo dei 128 bit con altri 128 bit: ho 2^{128} possibile input, il rimpiazzo è guidato da un chiave segreta che è il tipo di sostituzione che vado a fare: k seleziona una delle possibile permutazioni.L'algoritmo di blocco (difficile da disegnare) dovrebbe implementare una

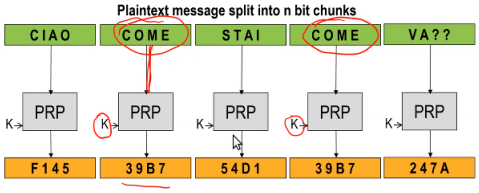
pseudo-random permutation o PRP, chiave dovrebbe selezionare una tra tutte le possibili permutazioni (in pratica, sceglie tra un insieme di queste)

10.10.1 PRP

S è l'insieme di tutti i possibili plaintext, se n = 3, la cardinalità di S è 8. Una permutazione è una biezione Π che associa ad ogni elemento di S un altro elemento di S. Voglio che la trasformazione sia reversibile.
Pseudo-random: il block cipher dovrebbe selezionare uniformemente una delle possibili permutazioni.Se ho 8 simboli, ho un totale di permutazioni=$2^n!$. A seconda del valore della chiave avrò una certa permutazione associata. La permutazione Π_k è associata alla chiave k.Quante permutazioni associate a quante chiavi: se ho 3 bit, $2^3! = 40320$ (posso avere anche la permutazione identica, ma nella pratica non si usa). Con 8 bit, ho 256 simboli differenti che si permutano in 256!: 8.58x$10^5$06. In AES n = 128 bit, numero di simboli è 2^{128}, n° permutazioni è 2^{128}! che è una roba non pensabile.Dovrei avere una chiave di 10^{40} bit mentre in AES sono di 128, 192 o 256. Se considero chiave di 256 chiave ho un numero totale di chiavi di 2^{256} che è il max numero di permutazioni che è molto molto minore del totale delle permutazioni (ricorda che ad ogni chiave è associata una specifica permutazione); probabilità di selezionare la permutazione random è ϵ. Considero quindi un subset dell'insieme delle PRP, ma siamo comunque ok (vedi ad esempio AES).

10.10.2 Problema 1-Plaintext più lungo della taglia del blocco

Ho capito che cos'è il PRP, mi fido che è ben fatto da crypto guys (se vedo alcuni mapping non posso sapere a cosa mappa un certo valore basandomi sugli altri, questo come nozione di sicurezza del PRP).
Plaintext più lungo della taglia del blocco, devo fare qualcosa: posso pensare di dividere il messaggio in chunks di taglia uguale alla taglia del PRP block (128 bit in AES ad esempio) e passare ogni blocco al PRP. Sbagliato: se m[1] ed m[3] sono lo stesso messaggio, il PRP fa si che il ciphertext si ripete, perché la chiave è la stessa



Perdo la semantic security, non IND-CPA secure.L'approccio è chiamato Electronic Code Book (ECB), e non va bene (big red flag).

10.10.3 Problema 2-Stesso plaintext

Ho visto il problema di ECB, ma se il messaggio è più corto? Ho ad esempio 8 lettere di messaggio ed il PRP usa blocchi di 8 byte, quindi uso un singolo

blocco e qui ECB potrebbe funzionare. Ma se lo encrypto di nuovo, riottengo lo stesso ciphertext.
Riuso l'idea dell'inizialitation vector: lo uso per ogni nuova encryption: prendo plaintext lungo quanto un singolo blocco, genero l'IV che deve essere di taglia uguale al plaintext. Li combino con XOR, ottengo sempre 8 byte di risaluto ed ora uso PRP indicizzata dalla chiave k ed ho il mio ciphertext.Posso trasmettere l'IV in chiaro con il plaintext, quindi c'è dell'overhead. A receive side: prendo cipher, faccio PRP^{-1}, faccio XOR con l'IV ed ottengo il messaggio; va tutto bene se la taglia del messaggio è $\leq$ della taglia del blocco.
L'IV non si ripete mai? Ma non basta, deve anche essere impredicibile (alla base del BEAST attack). Ovviamente, se riuso lo stesso IV ed il plaintext è lo stesso l'encryption è lo stesso, inoltre non deve essere predicibile: una nonce non deve essere impredicibile, basta che sia fresca. In questa specifica applicazione non è così.

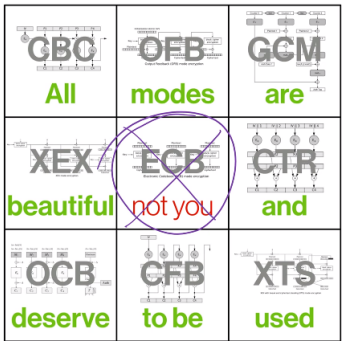
10.10.4 Modes of operation

Non usare mai ECB, usabile solo se :

- Messaggio è minore o uguale ad un blocco
- Messaggio non si ripeterà

Solo in queste condizioni strette si potrebbe usare (ma meglio non usarlo)
Per messaggi ripetuti: si genera un IV, random e della stessa taglia del blocco.
Per messaggi più lunghi: modes of operation: ho due ingredienti:

- Il blocco stesso, cambia il modo di implementazione a seconda dell'algoritmo
- Modo di operare, ho visto solo CBC ma ce ne sono altre. Questo permette di criptare testo di lunghezza arbitraria e mi permette di criptare senza ripetizioni del cipher ne momento in cui ho ripetizioni nel plaintext.



I più usati in pratica: CBC o CTR(Counter Mode, preferita dal prof a CBC)
NIST raccomanda, oltre queste 2, CFB e OFB.
Alcune più sofisticate, che combinano authentication ed encryption
Semantic security: prendo il plaintext, lo taglio in blocchi e genero un IV indipendente e truly random che associo ad ogni blocco, a questo punto combino

in XOR con questi IV e poi produco con il PRP il mio ciphertext. Siccome IV è random, non ho ripetizioni, ma ogni volta che mando il blocco devo spedire pure l'IV, quindi non è soluzione pratica; è il meglio che posso fare.

10.10.5 CBC

Modo più comune è usare CBC:

- Prende blocco di messaggio, aggiungo IV al primo messaggio e faccio XOR con m[0]
- Trusto che PRP è pseduorandom permutation e quindi che c[0] sia come un "random" IV per il blocco 2
- Prendo c[0] come IV per il blocco m[1], quindi lo uso per fare lo XOR e poi passo a PRP.

c[i] = ENC=PRP(K, c[i-1] $\oplus$ m[i]).Overhead è solo un blocco extra, se messaggio ha taglia di m blocchi, il cipher è di m+1 blocchi.
La fase di decryption procede al contrario: prendo IV, metto in XOR con PRP^{-1} di c[0] ed ottengo m[0] che userò in XOR col risultato del PRP^{-1} del c[i] successivo. Encryption consuma tempo perché non è parallelizzabile, mentre la decryption lo è: se voglio decryptare solo un blocco faccio accesso alla RAM e decrypto solo quello (se ho salvato il ciphertext lì).
È il modo più comune di fare block cipher, è sicuro se IV non è predicibile e non si ripete.
È lento per l'encryption ma almneo è veloce in deryption (appropiato per accesso ad encrypted file system o DB). Servono 2 circuiti diversi per encryption/decryption: le due direzioni sono diverse
Inoltre, il plaintext deve essere multiplo del block size: se non lo è, paddo (esiste standard per questo), è necessario anche in altri casi.

10.10.6 Altri modi: CFB e OFB

CFB: prendo un blocco, prendo IV e ne faccio XOR. Applico PRP ed ottengo il ciphertext. Sto prendendo plaintext e applico due trasformazioni che sembrano encryption scheme: nella parte dell'XOR sembra uno stream cipher con keystream noto. Quello che CFB fa è: criptare l'IV, fare XOR col blocco di plaintext ed ottenere il ciphertext. Trasformo il block cipher in qualcosa tipo uno stream cipher. Ho un pad keystream ed ho l'XOR classico dello strema cipher. Ora faccio CBC chaining, uso blocco c[0] a cui applico PRP e faccio XOR col blocco plaintext successivo.
OFB: principio è lo stesso, parto dall'IVe ci applico il PRP ma ora uso direttamente questo risultato. Lo prendo, faccio PRP e lo metto in XOR col plaintext, per il blocco 1 faccio come in CFB. Posso partire da IV ed applicare i vari PRP, sembra molto di più uno stream cipher: parto da un seme e genero i vari keystream per criptare i plaintexts. CBC/CFB non sono parallelizzabili, mentre OFB sì: posso precomputare tutti i keystream.

Decryption in CFB: non devo più invertire il PRP, prendo IV, lo encrypto con PRP e faccio XOR col blocco in ciphertext per ottenere il plaintext.
Lo stesso vale per OFB; in CFB posso fare decryption in parallelo. Perché non usarli: problema dello short cycle:
OFB: IV $\rightarrow$ PRP $\rightarrow$ PRP $\rightarrow$ PRP... esempio: blocco di 3 bit, permutazione è selezionata random, parto dal primo e comincio ad iterare: mi fermo prima di 8, periodicità di 5.
Non tutte le permutazioni sono le stesse: alcune chiavi k potrebbe portare a permutazioni che risulta in short cycle. Se ho un IV sfortunato la periodicità cala. Posso avere anche cicli più piccoli, come 3.
Per usare questi sistemi serve anche conoscere le reali proprietà delle permutazioni, devo progettare AES per non avere short cycle, entrare nei dettagli etc... quindi piango.
Problema di CBC, CFB ed OFB, che in un modo o nell'altro fanno chaining dei blocchi: non ho garanzie che non finirò in uno short cycle a meno di aprire la scatola nera dell'algoritmo di PRP.
Come caso degenere, esempio: se ho 3 blocchi 011 011 011 ed uso CBC con IV 010 ottengo 3 ciphertext identici, quindi male.
Anche nel caso dell'hash chian(OTP) ho questo problema, per questo motivo in crypto si odia il chaining.

10.10.7 CTR

Counter mode encryption, molto semplice:

- Prendo un contatore, nonce completamente predicibile. Lo incremento per ogni nuovo blocco
- Faccio il PRP del counter (il counter deve avere la stessa taglia del blocco usato nel PRP). Ottengo i keystream differenti per costruzione del PRP, che è una biezione (non è più un hash) e se è fatto bene il keystream è scelto random fra le $2^{2^{135}}$.
- Periodicità: periodicità del counter, se vai da 0 a 2^{128} ho un PRNG perfetto che non si ripeterà, quindi non ho short cycles
- Faccio XOR del blocco m[0] con counter ctr ed ottengo c[0], stessa cosa per m[1] $\oplus$ ctr+1 = c[1]etc... Non c'è chaining, ogni blocco è indipendente ed ho chiavi indipendenti, che sono generate a partire da un contatore.
- Se conosco il counter e l'algoritmo, posso parallelizzare: se voglio blocco 200, prendo ctr = 200, uso PRP e computo c[200]. Stesso vale per decryption: voglio m[200], faccio PRP[200] $\oplus$ c[200] = m[200]

Non ho short cycles, incredibilmente efficiente in hardware perché parallelizzabile.
Iniziamento non molto famoso, ovviamente se counter si ripete $\rightarrow$ gameover, anche se lo suo per lo stesso messaggio: deve essere simile ad un IV.

AES-Crt: blocco è di 128 bit (AES), è stato standardizzato che se accetto che la taglia massima di encryption è 2^32 blocchi, ovvero 2^32x16 bytes = 500 GB, prendo gli ultimi 32 bit e li uso come counter ed uso gli altri 96 bit come IV: parto da 0 come crt ma l'IV sarà truly random. Vantaggi:

- Rende block cipher stream
- Combina vantaggi di CFB e OFB (molto efficiente per encryption di file system)
- implementazione efficiente hw e sw
- richiede implementazione solo dell'encryption block (PRP)
- Posso fare random access: se indicizzo blocco col contatore, posso convenientemente accedere (per esempio memoria)
- Se ben usato (IV truly random e non si ripete, usato per al più 2^{32} blocchi) è l'approccio più sicuro: è garantito che per ogni permutazione di AES-CTR non ho problemi di weak permutation, per costruzione non posso avere short cycles.

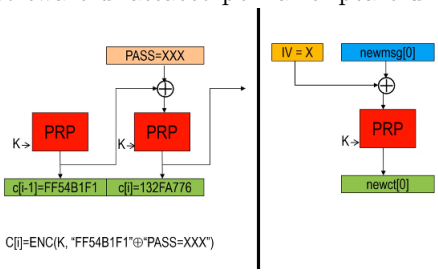
10.11 Vulnerabilità di IV predicibili

Per criptare due messaggi: i due IV devono essere diversi, ma questo non basta: l'IV deve anche essere non predicibile e questo sembra contro-intuitivo.
Suppongo di voler criptare m1|m2: prendo IV, choppo m1 ed m2 in blocchi, uso cipher block chaining così che l'ultimo blocco c di m1 sia usato come IV del primo blocco di m2. Uso IV per criptare m1, assumo che m2 usi l'ultimo ciphertext del messaggio m1. Sembra avere senso: perché la costruzione sopra in cui ho i due messaggi vicini è diversa da questa in cui il messaggio m2 è su una nuova riga? L'intuizione mi dice che se la prima costruzione è sicura (e lo è), allora anche la seconda lo è. Ma io ho scoperto che l'IV deve essere impredicibile: se uso la parte finale c[n] del messaggio 3, posso predire il prossimo IV: nella seconda costruzione sto violando la proprietà 2 di impredicibilità, lo vedo come ultimo ciphertext del messaggio trasmesso precedentemente.
Ho un messaggio legittimo, trasmesso da qualcuno. Posso solo vedere il ciphertext, ma ricevo un vantaggio: nel blocco m[i] del messaggio (assumo blocchi di 8 lettere) ho la stringa pass = XXX (non so cosa ci sia in XXX). Vantaggio 2: ??? non è sequenza arbitraria ma ha solo due possibilità: JOE o UGO. Basta per rompere il sistema? Lo sarebbe se fosse possibile fare dictonary-like attack: posso fare Chosen Plaintext Attack, 3° vantaggio è quindi di criptare un messaggio a scelta. Posso criptare PASS=JOE, guardo il ciphertext e vedere se combacia. Ma l'attacco non è possibile: se encryption scheme è semantically secure, ho un ciphertext diverso per lo stesso plaintext.
Se l'IV può essere predetto, posso rompere questa proprietà. Posso imparare, sapendo che l'IV è predicibile, selezionando un testo di mia scelta così da rivelare quale password è stata trasmessa nel messaggio precedente? Entro nel browser

e faccio criptare a TLS un plaintext di mia scelta: IV è predicibile, posso vedere ct[last] e so che viene usato come IV del prossimo messaggio, che è quello che l'attacker fa criptare: ho X predicibile, che è l'IV. Scelgo il messaggio: XOR è prima del PRP, se ho X $\oplus$ qualcosa come messaggio, prima del PRP posso vedere il qualcosa. Qualcosa = JOE$\oplus$UGO, quindi riapplicando l'XOR ho solo il qualcosa. Nel messaggio prima avevo PASS=XXX $\oplus$il cipher c[i-1]. Se metto nel testo PASS=JOE$\oplus$[c-1], mi rimane solamente il PASS=JOE se il guess era corretto.Quindi:

- Predico X
- Scrivo un messaggio che contiene X $\oplus$ c[i-1] del messaggio precedente $\oplus$ PSWGUESS
- Convinco l'implementazione a criptare il mio messaggio (verrà fatto X $\oplus$ X $\oplus$ c[i-1] $\oplus$ PSWGUESS = c[i-1] $\oplus$ PSWGUESS), se il c[0] risultante è uguale a c[i] (ovvero X), il guess è corretto. Altrimenti, la password è l'altra.

Effetto di overall è che se IV è predicibile posso fare trial and error attack.
Da un punto di vista del sistema: ho il browser web acceso e questo può tx un messaggio con TLS. Il messaggio ha la psw, per attaccare usando il browser uso software di attacco per far criptare un nuovo messaggio scelto da me.



10.11.1 Exploit in TLS-BEAST attack

Sembra un attacco teorico: se ad esempio la psw è 8 byte: 2^{64} try, quindi il CPA si può fare ma è idealizzato. Inoltre devo fare il CPA.
Il fatto che l'IV non dovesse essere predicibile era ben noto dal 1999 (Rogaway, IPsec), inoltre problema di standardizzazione in TLS 1.0: TCP trasmette in ordine, per evitare un IV fresco per ogni messaggio, l'idea è di usare il ciphertext dell'ultimo messaggio come IV del successivo, basandosi anche sul fatto che TCP lo permetteva.
In TLS 1.1+ fu corretto, aggiungendo un IV esplicito per ogni messaggio (mandatory in DTLS). Nel 2011 pochissimi usavano TLS 1.1
Beast attack: si credeva impredicibile, software issue: devo installare Trojan nel PC della vittima e fare injection di messaggi nel browser.C'erano nuove tecnologie come Websockets, HTML5 ed era possibile avere connessione aperta e catturare extra sources per fare injection. Altro motivo per l'impraticabilità era

la complessità: per decriptare un blocco di AES, quindi 128 bit di blocco, devo fare brute force di tutti i 2^{128} possibilità: non devo basarmi solo su psw, perché si usano spesso cookies ($>$ 64 byte e molto entropici).2011: Attacco lineare nella dimensione del cookie (demo su youtube)
Come trasformare brute force attack in tempo lineare: per crackare un blocco dovrei enumerare tutti i possibili valori dei blocchi che sono 2128 tentativi. Ma è possibile fare injection di testo: quando mi connetto ad un server, mando un messaggio. Non conosco l'auth cookie, perché è nello storage sicuro del browser. Ma quello che posso fare è non solo fare injection di un nuovo messaggio, ma anche iniettare del testo di preambolo prima della connessione al server: creo un commento che aggiungo al messaggio (così che il server non lo parsi) e posso misurare la taglia del commento, posso muovere il limite da cui partire: se posso aggiungere testo per cui un carattere noto cada alla fine del blocco, devo fare 256 tentativi per crackarlo. Chosen Boundary attack: selezioni e cambi i limiti del blocco criptato.Ora è possibile attaccare: attacco la prima lettera, poi la seconda (lo forzi o aspetti il prossimo accesso). Complessità dell'attacco, se cookie è di N byte l'attacco è di N·256 tentativi, contro il 256^N.
Quindi, l'IV predicibile in CBC è exploitabile

10.12 CRIME attack

Compression leaks. Stessi autori del BEAST (orami avevano i trojan per muovere i limiti del blocco). TLS usa compression e poi encryption, anche se non posso decrpitare, la taglia della compressione rivela delle informazioni ma in teoria non sono usabili.
Compressione disabilitata dopo questo attacco.Problema non solo di TLS, ogni volta che c'è compressione e poi encryption può esserci un problema. C'era un paper del 2002 che spiegava come la compressione potesse rivelare informazioni sul plaintext.
Idea: posso comprimere e cripater due stringhe: AAAABC $\rightarrow$ 4ABC (se non trasmetto numeri), se non ci sono ripetizioni non posso fare nulla (es ABCDEF). Ora faccio encrypto: ad esempio con stream cipher vedo una stringa di 4 caratteri che non ha leak. Idea: plaintext injection nel BEAST attack, aggiungendo un preambolo per shiftare preambolo. C'è una password in plaintext, non la conosco. So che sarà compressa + criptata ad una taglia es di 6 byte. Ricordo che potevo aggiungere un preambolo: posso ad esempio mettere AAA o BBB o SSS come preambolo, ovvero una sequenza compressa. Guardo al risultato: se comprimo AAASHARON il risultato è 3ASHARON, BBBSHARON è 3BSHARON, ma SSSSHARON sarà 4SHARON. Primi due casi ho 8 byte, ma nel caso 3 ho 7 bytes $\rightarrow$ capisco la prima lettera. Se posso aggiungere un preambolo e forzare l'implementazione ad aggiungere il preambolo, allora posso rivelare una lettere e decriptarla.
Lo stesso problema può accadere se ho un DB, all'interno del cui ho dei dati privati, e che viene poi compresso e criptato. Se posso fare injection di dati nel DB, ho la stessa vulnerabilità; quindi non è solo un problema di TLS. Attacco CRIME funziona anche per block ciphers, cambiano i dettagli a seconda

dell'algoritmo di compressione. Possibilità di fare injection di testo, prima dei dati utenti: in BEAST era padding con commenti inutili, mentre ora è injection di testo scelto. Dettagli di uno specifico meccanismo, in questo caso DEFLATE, ma può essere applicato ad altri compression schemes.
DEFLATE: due tecniche, una bit oriented, un'altra è il Lempel Ziv algorithm: lavora a livello di byte, prende 3+ caratteri, fa un replacement: giuseppe bianchi and marco bianchini, primo passo del parser vede che "bianchi" si ripete (anche spazi si ripetono, quindi vanno compressi).Lascia la prima stringa inalterata, per l'altra aggiunge una coppia(-18,7): -17 dice di andare indietro di 18 caratteri (pointer) e il secondo numero dice quanti byte prendere (in realtà è (-17,8) contando gli spazi). Risultato è: giuseppe bianchi and marco (-18,7)ni.
Ho un testo utente legittimo, voglio indovinare la password:
GET /comment:twid=a HTTP1.1 Cookie: twid=flavia... Aggiungo un preambolo alla richiesta, che sia una parte di commento, in questo caso /comment:twid=a. Conosco format della richiesta di Twitter, quindi so che parte con twid= e cerco di indovinare la 6° lettera, ad esempio la a. Tutti i compression scheme usano una finestra per spostarsi, in quanto diventa difficile andare indietro, ad esempio di 2GB.
Una volta compresso: GT /commenti:twid=a HTTP/1.1 Cookie: (-24,5)flavia. Prende solo il preambolo twid=, ma posso ripetere finché non becco la lettera f, perché mi rendo che la taglia di byte da leggere diventa 6 invece di 5. Ora provo la lettera 2 e così via...
Attacco lineare all taglia del segreto, quindi molto pratico da effettuare. Anche se auth cookie è di 64B ho un O(64x256) nel caso peggiore, quindi attacco molto pericoloso.

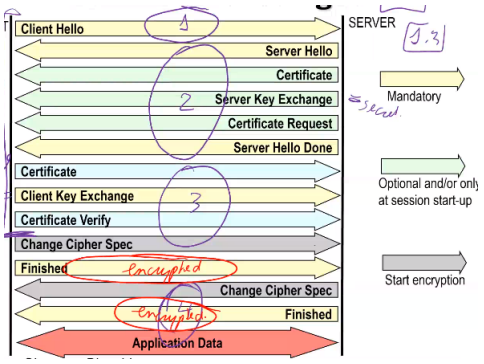
10.13 TLS Handshake Protocol

A partire da TLS v1.2, in TLS v1.3 differisce. Quando si usa l'handshake: connessione ad un server. https://www.fineco.it, ogni votla che mi connetto parte una handshake iniziale che ha come obiettivi:

- Mutually authenticated, ma di solito è unilaterale: quando apro TLS connection la banca mi prova la usa autenticità e una volta che il tunnel TSL viene stabilito (encryption ed integrity) mando username+password. La banca non sa se sono autentico a meno, uso PAP all'interno di TLS. Se posso assegnare all'utente un certificato di sicurezza, posso usare la mutual authentication.
- Non c'è algoritmo di encryption specifico: protocollo è disaccoppiato dal algoritmo crittografico di sicurezza, quindi si negozia l'algoritmo che verrà usato. Ma può essere attaccata la negoziazione: se convinco client e server che si può usare solo RC4? Bisogna proteggere questa fase.
- Servono nonces, scambio delle random quantity
- Serve scambio dei segreti per computare i segreti. Serve quindi asymetric cryptography, non posso mandare in plaintext

Se mi collego, e poi mi ricollego di nuovo il processo ristarta. Ma nella connessione scambio molti messaggi col server, idea di TLS è che ogni connessione deve usare una chiave di encryption e di HMAC diverse anche se le connessioni sono su due server diversi. Questo permette di ridurre il riuso di chiave ed evitare crypto analisys. Ma per usare nuova chiave, l'asymmetric crypto e computazionalmente costosa: symmetric vs asymmetric è 10^4 più veloce. L'idea è che posso riusare del lavoro fatto nel primo handshake per farne uno abbreviato, quello che si fa nel mondo reale. Su altre connessioni TCP uso handshake più leggere, abbreviate: sesison resumption, si riusa parte del lavoro della prima connessione per computare chiavi differenti per la connessione.
Obiettivi dell'handshake:

- Negoziazione sicura dello shared secret, con asymmetric crypto
- Autenticazione opzionale, facendo autenticare sia client che server, in modo da essere robusti a MITM. Non mi proteggo da ARP poisoning, DNS spoofing etc..., ma TLS garantisce che l'atk non può vedere o modificare il contenuto dei messaggi: ho integrity protection e encryption.
  Ma se la banca non è autenticata, atk si finge la banca, prende il traffico lo modifica e lo manda a Fineco ma questo è protetto dall'autenticazione. Tecniche robuste contro attacchi classici e strong, ma se attacco è del governo manco per nulla.
- Negoziazione affidabile: attacker non deve poter fare danni nella fase di negoziazione. Se mi scambio gli algoritmi di encryption ed un atatcker convince entrambi gli host che si può usare un protocollo weak è finita. Va protetto



Tutti i messaggi precedenti alle frecce grige sono in chiaro, aggiungo header di 5 byte ai singoli messaggi e li passo a livello 4 a TCP.
Header TLS:

- Content type
- Major version
- Minor version

- Length

Poi c'è payload del messaggio.
esempio: mi collego ad una banca, prima di inserire in miei dati solo la banca si autentica a me, io mi autenticherò passando username e password. Scambio TLS v1:

- Primo messaggio è mandato dall'utente al sito della banca, che è il Client Hello. Incapsulato su TCP/IP, la parte di TLS:
  - v1.0 (SLL 0301)
  - length
  - handshake type
  - 32 byte di nonce, contiene sia timestamp che random bytes. Ora deprecato il timestamp, non è autenticato quindi non è detto che sia un timestamp vero, quindi ora sono 32 byte du random e non più 28+4.
  - (Session id lenght, session id): può servire se voglio riusare una sessione precedente.
  - Parte della negoziazione: cipher suites e compression): in TLS, la prima cosa da fare è condividere il segreto, si fa con asymmetric crypto, anche detta public key cryptography. Devo scegliere l'algoritmo da usare: in TLS tutto va negoziato (no hardcode nel protocollo), possibilità di negoziare il public key algorithm: esempio TLS_RSA: uso di RSA come algoritmo asimmetrico. C'è anche una seconda parte: TLS_RSA_WITH_ algoritmo da usare per symmetric encryption ed integrity, quindi cosa userò quando comincia l'encryption dei dati. esempio: TLS_RSA_WITH_RC4_128_MD5.
    Oggi la lista di ciphers è estesa (è possibile specificare NULL per la parte dell'algoritmo di encryption e richiedere solo integrity, conferma del fatto che i due servizi sono diversi).
    La parte importante è che il protocollo non sia legato intrinsecamente al crypto algorithm.
  - Versione di Aprile 2012: negoziazione anche del compression algortihm (prima di CRIME), ora hardcoded NULL.

  Nel Client Hello do una lista di possibili algoritmi i supportati), ed il server ne sceglie una. Se il server è vecchio, può scegliere l'algoritmo peggiore (le opzioni sono messe in ordine crescente per livello di sicurezza). Negoziazione: client offre, server sceglie.

- Server Hello:
  - Handshake type

- versione del protocollo: anche la versione di TLS viene negoziata, anche questo è fondamentale. Per questo è presente la versione nel client: appariva in due parti, nel record protocol e nell'handshake. Nell'handshake c'è la proposta di quale versione usare, quindi il server può anche sceglierne una diversa se non supportata.
- Nonce del server, 32 byte(4 TS + 28 random).Server risponde 3 ore dopo? No, il timestamp probabilmente è considerato diversamente sui due sistemi, quindi 4 byte di TS erano uno spreco perché non c'era una time reference precisa.
- Ho un session ID ora
- Cipher suite: server risponde con un protocollo diverso da quelli che aveva il client come priorità

Ho appena visto la negoziazione della versione, downgrade attack (arma grande, difficile da fixare): client usa TLS v1.0 (SSL v3.1), server anche supporta TLS v1.0 ed assumo che la versione sia sicura.
Assumo che l'attacker non possa rompere TSL v1.0 ma possa rompere SSL v2.0
Attacker prende Client Hello, è in plaintext e senza integrity perché non ho un segreto shared, quindi non posso criptarlo, messaggio molto vulnerabile.
Vado nella parte del messaggio dove c'è a versione e cambio: cambio due byte, da 0301, metto 0200.
Server vede che versione è più bassa, ed accetta la connessione (oggi non è possibile scendere sotto TLS v1.0, poodle attack: downgrade di TLS + padding oracle). Ora attacker rigira la risposta al client, così che questo creda che il server abbia supporto alla versione di TLS/SSL più vecchia.
Agreement sulla versione di SSL v2.0, convinti che questa è l'unica versione usabile; è un problema fondamentale di tutte le negoziazioni.

10.13.1 Public key cryptography

Esempio di problemi risolti da asymmetric encryption: comunicazione tra l'app ed il server è essere criptata, ma il problema è dov'è la chiave. Se ho una SIM card, ho lì l'encryption key. Ma come risolvo problema del downaload di un applicazione: scarico e mando encrypted traffic, ma dove metto la chiave? Se metto nel codice, male: hacker wannabe 101 cerca pattern nel codice, faccio regex (della chiave) e guardo nel binary code dove il charset è ristretto a base64, quindi ho trovato la chiave. O predo la chiave da una comunicazione separata, ma come faccio: se devo chiederlo al server devo poterla trasferire.
Asymmetric cryptography: differenza con le simmetric key è che la chiave usata per criptare è diversa da quella usata per decriptare. Solitamente: prendo plaintext, applico una trasformazione che deve essere reversibile, ed ho il mio ciphertext. Ho visto stream e block cipher, ma si basano entrambi su una chiave preshared.
1970: schema dove

- La chave usata per encryption è diversa da quella per decrypt.

- Impossibile derivare una chiave dall'altra: se ad esempio conosci quella per encrypt non puoi trovare quella usata per decriptare

Se ho questo schema: user cripta i dati con una chiave, trasferisce ma la destinazione usa una chiave differente.
Problema risolto nel 1977 Rivest, Shamir, Adleman, RSA.
Le chiavi devono poter essere linkate, siccome sono asimmetriche e conosco K_{ENC}, K_{DEC} è nascosta, quindi come corollario posso avere K_{ENC} pubblica.
Come faccio quindi a risolvere il problema di sopra: scarico l'applicazione, che contiene anche K_{ENC}. Tutti vedranno K_{ENC}, è in plaintext. L'utente vede K_{ENC}, tutti lo vedono quindi chiunque può criptare qualcosa: ma nessuno vede la chiave K_{DEC}. Quindi, se uso la K_{ENC} per criptare i dati da mandare la server, nessuno potrà decriptarli.
Public key = K_{ENC}, private key = K_{DEC} (conosciuta solo da 1).
Perché continuo ad usare symmetric key:

- Risolvono due problemi diversi, non posso dire che una è meglio dell'altra
- entrambe possono essere sicuri o non sicuri
- Asymmetric è più flessibile, ma servono protocolli più complessi.
- (Algoritmi noti sono rotti dal quantum computing, symmetric no).
- Asymmetric è da 3-5 OOM (order of magnitude) più computazionale complessa.

Nessuno usa asymmetric encryption per trasferire dati, si fa questo: user ha la K_{ENC}, deve trasferire i dati, prima di fare questo genera una chiave k simmetrica, trasferisce $ENC_{K_{ENC}}(k)$. Solo il server può leggere, quindi ricava la chiave k simmetrica ed usa un algoritmo apposito (AES) per trasferire i dati.
Due possibili approccio fino a TLS1.2 per fare key management:

- Key transport (es RSA, ora non più usata):
  - server manda una key pubblica, che sarà nel certificate message (non può essere trasmessa in plaintext), client può anche salvarla se sa che la riuserà
  - Client genera un segreto random, che critpa usando la chiave pubblica del server e trasportato al server
  - Ora è possibile fare symmetric key encryption.
- Key agreement (DH algorithm), successo nel trasferire lo stesso segreto alle due side:
  - Tutti e due generano una chiave privata e pubblica, es Y, $g^Y modp$ ed X, $g^X modp$,se conosci $g^Y modp$ non puoi ricavare Y (vedi su la lezione sulle cripto hash)

- Client e server si mandando le chiavi pubbliche, l'attacker non riesce a ricavare facilmente X ed Y. Ma non è difficile per gli end: se ho g^X ed Y posso fare $(g^X)^Y$, quindi ho g^{XY} e posso ricavare le chiavi.
- Se l'attacker li moltiplica, ottiene g^{YX}??, No è la somma g^{Y+X}.

Quindi TLS può usare uno o l'altro modo, anche questo viene negoziato.
TLS handshake: Client manda Client Hello, server risponde con Server Hello e qui è stata negoziata la versione del protocollo TLS.
Ora con RSA per esempio: dopo il Client Hello, server manda una chiave pubblica certificata (attacker non può cambiarlo). Ora client applica key transport mechanism: genera segreto k shared e lo trasferisce al server criptato con K_ENC, non può essere modificato (o meglio lo assumo, rotto nel 2018). Server riceve il messaggio ed estrae il segreto shared. C'è altro trick: metto nel messaggio del client anche il codice della SSL version per prevenire il downgrade attack: se l'attacker l'ha modificato, io non mi fido e invece di essere d'accordo rimando la versione che avevo scelto inizialmente invece di quella accordata: ricordo di nuovo al server che potevo usare una versione più recente di SSL, ma lui ha detto di usare una versione più bassa. 2 byte di overhead. Ora l'attacker non può modificare i messaggi: né quello con la chiave pubblica né quello con la chiave shared dal client per il server. Quindi, il server si accorge che qualcosa è andato storto: mi rendo conto che il client mi ha rimandato la vecchia versione proposta e non quella accordata, quindi stoppo la connessione. Metodo fondamentale per risolvere downgarde atttack, nel momento in cui da un certo punto in poi la cripto è up: da quando è up, ripeto quello che ho negoziato. In un protocollo non posso essere sicuro di quello che accordo in plaintext.
Ma riguardo cipher attack? Attacker potrebbe convincermi ad usare altro protocollo di encryption, etc... Sto proteggendo solo la versione, inoltre cosa cambia tra i dure approcci: in DH non posso trasferire informazioni nei messaggi, ho delle quantità fisse, non è un public key crypto system, usa asymmetric crypto ma non è un crypto system così com'è: non posso trasferire qualcosa, come ad esempio la versione di TLS.

10.14 Asymmetric cryptography

Nell'encryption asimmetrica: prendo plaintext, uso chiave chiamata encryption key (algoritmo di derivazione della chiave noto), ottengo ciphertext e mando alla destinazione. Qui si una una chiave per decriptare che è diversa da quella usata per l'encryption, sono ovviamente collegate l'un l'altra ma non si può derivare una conoscendo l'altra (a meno di conoscere altri dettagli). Spesso K_{ENC} = public key e K_{DEC} = private key: chiunque può criptare ma una sola persona può decriptare.
HTTPS/TLS:

- nella fase di handshake, viene mandata una segreto shared.
- Nella seconda fase si usa key derivation functions per derivare dal segreto shared la chiave di encyption e di integrity.

- Refresh della symmetric encryption e message authentication usando sempre lo stesso segreto della prima fase (nel caso di nuova sessione?).

Hybrid encyption, spesso applicata. non voglio mettere su una connessione TLS, ma voglio fare encryption di un dato, esempio un e-mail ed uso una chiave pubblica, ad esempio quella dell'utente a cui voglio inviarla. Ma il messaggio è lungo:

- genero una chiave K simmetrica, random number
- Uso un cipher ordinario, come AES, ed encrypto messaggio usando la chiave K ed AES $\rightarrow$ symmetric encryption.
- Prendo k e la cripto con asymmetric cipher, e lo mando col messaggio. Mando un messaggio in cui nell'header ho asymmetric encryption della chiave, che è la chiave simmetrica usata per fare enctyption, più il campo dati. Sfrutto la velocità e scalabilità dell'asymmetric encryption.

10.14.1 PubKey crypto

Encryption e decryption sono una l'inverso dell'altra: DEC(ENC(M)) = M, quindi posso fare in questo ordine: public key encryption, un crypto system in cui sto criptando i dati in modo che tutti possano criptarli ma sono uno possa decriptarli. Assumo che l'operazione sia commutativa: ENC(DEC(M)) = M, ha senso dal punto di vista algoritmico (ma non semantico). Ha senso per fare la digital signature, è la duale del public key enrcyption: nel caso della digital signature

- Prendo la mia chiave privata, prendo un messaggio M ed applico $D_A(M)$, posso applicarla solo io perché solo io ho la chiave. Chi può invertirla? Tutti, quindi se trasmetto M, $D_A(M)$: chiunque può vedere il messaggio, l'extra information TAG (AUTH) e quindi se applico al TAG la trasformazione inversa, ovvero lo encrypto posso verificare se ottengo il messaggio originale o altro $\rightarrow$ creo integrity ma con asymmetric crypto.

Posso quindi creare due applicazioni diverse: usa è per l'encryption di dati, una per la digital signature. Devo trasmettere un messaggio: applico asymmetric crypto del primo caso. Applico una hash function crypto, comprimo ed ottengo digest e nessuno può ottenere lo stesso hash con un messaggio diverso. Quindi quello che faccio è la cosa seguente: ho messaggio compresso, trasformo usando l'operazione di decryption (che usa la chiave segreta), quindi trasmetto il messaggio più uno short tag. Per verificare se messaggio è vero o no: uso la public key per invertire l'encryption dell'hash computato prima. Quindi faccio hash del messaggio verifico se i due hash combaciano. Un attacker può solo scoprire cosa c'è nel tag invertendo con la chiave pubblica, ma non può in nessun modo ricavare un messaggio m' che mi generi lo stesso hash. L'hash del messaggio ha due obiettivi:

- Restringere il messaggio così da applicare trasformazioni su un singolo "blocco".
- Sicurezza: se non uso un hash, il sistema è vulnerabile, non solo per ottimizzare le perfomance, avrei un problema di malleability

Message integrity con authentication alla sorgente: in MAC, ho una sorgente, messaggio M ed il tag(K,M), ma se K è shared non posso autenticare la sorgente, non sono sicuro che l'abbia generato la sorgente, può averlo fatto anche la destinazione. Non c'è la non repudiation property (source authentication). Nella digital signature il concept è lo stesso, ma la chiave K usata nel TAG è privata ed è della sorgente.
Attacker può comunque prendere messaggio, sostituire la sua HASH + enc del messaggio e spacciarsi per la sorgente e rendere il messaggio valido.

10.14.2 Basic Algorithms

Pionieri: Diffie-Hellman key agreement, problema dell'asymmetric cryptography, vera soluzione 1977 da Riverst, Shamir, Adelman, RSA cryptosystem. Problema di usare l'approccio di DH in crypto systems fu risolto nel 1985 da El Gamal.
Problema generale: trovare un problema asimmetrico difficile in una direzione e facile nell'altra. esempio: hash function è un problema asimmetrico (da hash a digest facile, da digest ad hash è undefined).
Diffie ed Hellman trovarono problema: computazione del logaritmo discreto, ma non poterono costruire un crypto system su questo. RSA: non trovarono un modo di adattare il problema, ma trovarono altro problema:
ho p e q primi molto grandi, è facile fare il prodotto di p e q. Ma se ho n = p·q, è difficile trovare p e q, problema di fattorizzazione, approccio per trovarli è brute force (tento tutte le possibilità).
Ho un primo p ed un "numero" g (generator, poi scoprirò). Prendi x: $g^x mod p$ p è di 1024 bit, x anche può essere grande. Operazione è veloce o no? Se devo elevare $g^{132412394533242543}$, ma c'è trick per renderla più veloce. Devo computare $g^{1437} mod p$. Prendo l'esponente e lo traduco in forma binaria: $1437 = 10110011101_2$.Lo metto in colonna in ordine inverso (dal bit meno significativo al più significativo) e creo square e multiply. La prima linea è di inizializzazione, nel caso ci fosse 0, metterei 1 nel multiply (altrimenti metto g). Nelle seguenti linee: se ho un 1, nella multiply moltiplico il valore di Multiply della riga sopra per il valore della Square della riga corrente, mentre nella Square faccio una operazione, ovvero il quadrato della componente precedente.
Quante operazioni compio: 10 operazioni di Square + 6 Multiply, passo dalle 1436 operazioni iniziali a 16. Risultato generale: $log_2(numero)$ + le Multiply: $log_2(numero)$ nel caso peggiore, 0 nel caso migliore, avg: $\frac{log_2(numero)}{2}$ quindi in totale = $1.5 \cdot log_2(numero)$. Calo da complessità esponenziale a complessità logaritmica: lineare con la taglia in bit del numero. Dlog è complesso perché, dato un $y = 3^x mod 104729$, e chiedo di trovare un x tale per cui y = 33490. Se la funzione fosse monotona potrei applicare algoritmi molto efficienti per trovare

bit	Square	Multiply
1	g	g
0	g^2	g
1	g^4	g^5
1	g^8	g^13
1	g^16	g^29
0	g^32	g^29
0	g^64	g^29
1	g^128	g^157
1	g^256	g^413
0	g^512	g^413
1	g^1024	g^1437

il valore di x. Ma in questo caso non è così, la complessità rimane esponenziale. Quindi ho il problema che volevo trovare.
Diffie-Hellman, protocollo di key agreement, permette di settare tra client e server un segreto shared, non trasferendolo da una parte ad un'altra, bensì usando computazioni: asimmetrico nel modo in cui ricavo il segreto shared, scambio dei valori pubblici.

- Alice genera una quantità random, x. Bob genera y, random.
- Alice manda a Bob $g^x modp$, si computa facilmente. Bob manda $g^y modp$
- Alice prende $(g^y)^x modp$ ed ottiene k. Bob fa lo stesso, ed ottiene la stessa chiave k. Ho applicato di nuovo modular exponentiation, veloce.
- L'attacker ha visto lo scambio, ma non può computare la chiave perché deve invertire il dlog. Da g^x e g^y è difficile computare g^{xy}. Bob ed Alice hanno i "segreti" x ed y, quindi per loro è facile fare la computazione.

esempio: p = 29'996'224'275'833, g = 3 (sono parametri). x = 123456789, y = 234567890. Alice computa $g^x modp$, Bob $g^y modp$. Poi, alice farà $(g^y)^x modp$, mentre Bob farà $(g^x)^y modp$, ed avranno la stessa chiave K. In Diffie-Helmann: numeri di almeno 1024 bit.
La sicurezza dell'algoritmo si basa sul fatto che l'attacker deve per forza invertire uno dei due dlog: se conosco di più, ovvero x o y, allora posso riuscire a ricavare k: questa è la proprietà di asimmetria.
Limiti di DH:

- Non implementa un crypto system: risolve un problema, ora che ho la chiave K shared, posso usare AES-128_K(data), quindi ho risolto il problema pratico di trasferire i dati su un canale non sicuro. Ma non risolve la challenge originale: prendi un messaggio M, criptalo con una chiave K_{ENC} e decriptalo con K_{DEC}
- Non posso derivare facilmente una digital signature, non posso usare la chiave SK per la DS (non avrei la non repudiation property)

1977: RSA per risolvere il problema (ora non si usa più perché non ha implementazione efficiente su curve ellittiche).

10.14.3 RSA Algorithm

Algoritmo i Rivest, Shamir e Adelman; patented fino al 2000.
Problema asimmetrico difficile: dato N = p·q, è difficile fattorizzare N; p e q devono essere numeri primi grandi.
Operazioni di encyption e decyption sono semplici, e sono modular exponetiation. Questo può supportare sia l'encryption che la digital signature: posso modellare un crypto system ed avrò i due servizi a seconda dell'ordine delle operazioni.
Principio dietro RSA:
$m^x mod N$, c è un numero, quindi ogni messaggio viene tradotto in un numero, deve avere taglia $\leq$ N.
La funzione è periodica: se considero $3^x mod 10$ = {3,9,7,1,3,9,7,1,....}.Il periodo è lungo 4, se provo a cambiare m, la lunghezza periodo è uguale o minore: ese. $9^x mod 10$ = {9,1,9,1,..}, worst case period = 4.
Teorema di Eulero: computazione del max di $m^x mod N$, è la funzione Φ(N), più formalmente se m è co-primo con N (GCD(m,N) = 1), allora vale che $m^{\Phi(N)} mod N = 1$.
Computazione Φ(p):

- se p è primo, Φ(p) = p-1.
- se N = p·q, con p e q primi, avrò che Φ(p·q) = Φ(p)· Φ(q) = (p-1)·(q-1).
- Numero primo p^k, $\Phi(p^k)$ = (p-1)· p^{k-1}.

Conseguenza della periodicità: $m^x mpod p$ è periodica con periodo Φ(N)s. Prendo ad esempio $mod 11$, quindi $\Phi(11) = 10$.
Mi rendo conto che $9 \cdot 7 mod 11 = 2^6 \cdot 2^7 mod 11 = 2^{13} mod 11$. Quindi posso lavorare sull'esponente facendo mod Φ(N): $2^{13 mod 10} = 8 = 2^{3 mod 10} = 2^3$.
Conseguenza: $m^x = m mod N$ se x = $1 mod \Phi(N)$, in questo esempio x = 1 + kΦ(N).
Quando $3^x mod 10 = 3$? $3^5 mod 10 = 3$, ma l'equazione si risolve ogni volta che x = $1 mod \Phi(N)$.
Costruzione di RSA:

- Genero p e q primi grandi e li tengo segreti.
- Computo N = p·q e lo rendo pubblico, trovare la fattorizzazione è difficile (escludendo quantum computing la complessità cresce esponenzialmente con il numero di bit di N)
- Computo Φ(N) = (p-1)·(q-1), non posso computarlo conoscendo solo N, devo conoscerne la fattorizzazione. Ma ora ho il problema del mio recevier: la sicurezza di RSA risiede nel fatto che nessuno può computare Φ(N).

- Genero una public key e: 1 < e < Φ(N), e deve essere co-primo con Φ(N). (e è sempre dispari perché N è pari)
- Genero chiave privata d tale che e · d = $1 mod \Phi(N)$. Quindi $(m^e)^d mod N$ = m.
- Per risolvere: $m^x = m mod N$ devo risolvere x = $1 mod \Phi(N)$, ora ho $(m^e)^d$ = $m mod N$, devo risolvere $e \cdot d = 1 mod \Phi(N)$. Se conosco Φ(N) il problema è facile,altrimenti no.

Assunzione di sicurezza: dato N deve essere difficile trovare i fattori p e q, inoltre nno deve essere possibile computare Φ(N), e senza Φ(N) è difficile computare d ed e.
d è la chiave di decrypt (privata), mentre la chiave di encrypt (pubblica) è la coppia (N,e).
Perché funziona: se ti do N,e ed un messaggio cryptato $m^e mod N$, è difficile trovare x tale che $(m^e)^x mod N$ = m. Unico modo è fare brute force a meno che non conosci Φ(N). Problema dell'aritmetica $mod N$: le frazioni non si considerano, le soluzioni sono tutte intere.
esempio: p = 11, q = 17 (segreti), computo N = p·q, $111 cdot 17$ = 187. Φ(N) = 10·16 = 160 (segreto). Prendo e = 7, che è pubblico e primo con 160.
Ora, cerco d tale che e·d = 1mod160, in questo caso d = 23.
CHi computa ha la trapdoor (conosce Φ(N) che è quel qualcosa in più), rende pubblico N ed e. Per criptare M: C = $M^7 mod 187$, per fare decyption faccio $(M^7)^2 3 mod 187$.
Come funziona: sono la banca, genero p e q localmente e li moltiplico per avere N. Ora genero e e siccome ho computato io N, conosco Φ(N) e posso generare d. Dico all'utente di usare {N,e} come chiave pubblica, una volta fatta l'operazione mi tengo solo {N,e} d in privato. L'attacker può vedere il ciphertext, ma non può decriptare $m^e mod N$.
NB: messaggio numerico deve essere più corto di N (in termini di digits).
L'attaccante deve cercare tutti i numeri possibili: se riprova con e, ottiene qualcosa di diverso (non è symmetric encr, a meno che non conosca la fattorizzazione p e q), mentre per la banca è semplice perché ha la chiave privata d per decypt. N è grande, p e q anche. e può essere piccolo, mentre d è sempre grande: è utile spesso selezionarli entrambe grandi ma è possibile selezionare e piccolo (es = 3, così l'encryption è veloce). La probabilità che d sia piccolo è infinitesima, di solito si fissa una chiave pubblica e, da cui poi tanto si computerà d, non posso selezionare entrambe.
(Non raccomandato scegliere la chiave e piccola perché posso esserci attacchi, per ottimizzazione si può scegliere piccola, ma si apre a delle vulnerabilità).
Perché RSA funziona, trapdoor funciton: diventa facile computare $(m^e)^d mod N$ = m, con algoritmo di Euclide esteso:
assumo di avere due numeri coprimi, esempio 51 ed 11; GCD[51,11] = 1. Devo trovare a e b ∈ Z tali che 51·a + 11·b = 1.
1x51 + 0x11 = 51
0x51 + 1x11 = 11

Divido 51 per 11, segnando il resto: 4, r = 7. Prendo l'ultima riga, la moltiplico per 4 e la sottraggo a quella sopra:
1x51 - 4x11 = 7.
Ora divido 11 per 7, ottengo 1 con r = 4.
-1x51 + 5x11 = 4.
2x51 - 9x11 = 3.
-3x51 +14x11 = 1.
Ho trovato (a,b) = (-3, 14); complessità logaritmica, efficiente dal punto di vista computazionale.
Applico per computare l'inverso modulare:
ho e = 13 e Φ = 60, devo computare l'inverso di e: so che è co-primo con Φ(N), devo cercare $\Phi \cdot a + e \cdot b = 1$. Applico l'algoritmo di Euclide esteso e trovo (a,b)=(5, -23). La riscrivo isolando 13·(-23) = 1 - 60·5, ma l'uguaglianza vale anche per il modulo: $13 \cdot (-23) mod \Phi(N) = 1 mod(\Phi)(N)$, quindi ho trovato che d = -23 = 60 -23 = 37, quindi ora per decryptare mi basta elevare il ciphertext alla d.
RSA signature: prendo messaggio, faccio hash ed applcia DEC(H(M)). Quindi neòl caso di RSA faccio $H(M)^d mod N$, e mando il messaggio, la chiave pubblica e il TAG. L'altro end prende TAG, lo eleva alla e quindi ottiene $(H(M)^d)^e mod N$ = H(M) e può controllarlo facendo hash del messaggio.

10.15 Digital certificates and public key infrastrucutures

Digital signature: so come generare chiave pubblica e privata. Creo H(M) sul messaggio che voglio inviare, lo faccio perché la condizione per fare aritmetica $mod N$ il messaggio deve avere taglia più piccola di N, così lo ottengo. Trasformo il dato (encyption è termine improprio qui) facendo $H(M)^d mod N$: solo io posso fare questa computazione, ma chiunque può fare la trasformazione inversa $(H(M)^d)^e$ che mi fornirà H(M), ora computo di nuovo H(M) e controllo se torna con quello ricavato.

10.15.1 Problemi

Un attacker cerca di rompere la costruzione: prende q' e p', scelti da lui, computo N', genero chiave pubblica e' e chiave privata d', posso farlo perché conosco $\Phi(N')$. Modifico il messaggio M, faccio hash del così ottenuto M' e lo firmo con la mia chiave segreta d'. È vero che l'utente iniziale ha la chiave privata, ma l'utente finale deve avere la chiave pubblica e: L'utente finale ha già la chiave e salvata, ma come posso avere salvato ad esempio la chiave pubblica per un utente che non conosco? Devo per forza ottenerla dalla rete. Attacker può intercettare la comunicazione, dire di essere l'utente originale e rimpiazzare la chiave pubblica con la sua, e'. Quindi verificando il messaggio, questo è autentico ed è un problema.
Altro problema, RSA key transport: voglio connettermi alla banca con TLS, mando Client Hello e la banca deve mandarmi la chiave pubblica, scelgo una quantità random k e la trasferisco usando la chiave pubblica della banca.

Ma quando la banca mi manda la chiave pubblica, l'utente può mettersi nel mezzo e sostituirsi alla banca. Può farlo perché la comunicazione è attualmente in chiaro , quindi quando l'utente sceglie la random k, la manda criptata con la chiave dell'attacker, quindi attacker lo decrypta, lo legge e lo riencrypta con la chiave pubblica della banca rimandandolo a quest'ultima.
Ora l'attacker sa che il data exchange sarà criptato con la chiave k.
Problema anche in Diffie-Hellman: i due utenti Alice e Bob scelgono due chiavi e si mandano $g^x modp$ e $g^y modp$. Attacker si mette nel mezzo: sceglie una chiave z random, ferma la trasmissione di $g^x modp$ e genera $g^z modp$. Dall'altra parte, manda anche all'altro end $g^x modp$, quindi lo manda sia a Bob che a Alice.
Alice computerà una chiave $K_1 = g^{xz} modp$, Bob computerà $K_2 = g^{yz} modp$.
Ma l'attacker li ha entrambi: ha intercettato i due messaggi $g^x modp$ e $g^y modp$ quindi può elevarli alla z, che ha generato lui; attacker agisce da proxy.
Quindi ogni applicazione dell'asymmetric cryptography ha il problema del certificato della chiave pubblica: o ho TUTTE le chiavi salvate, oppure se devo recuperare la chiave dalla rete sono vulnerabile ad un MITM.
3 scenari differenti, stesso problema: ho un nome a cui è associata una chiave pubblica, tutti gli attacchi rompono l'associazione e cambiano una delle due parti. Bisogna legare, in termini di cryptographic bind tra il nome e la chiave pubblica.

10.15.2 Digital certificate

Digital certificate: qualcosa che mi permette di legare (in senso molto stretto) una chiave pubblica ad un soggetto, soggetto può essere persona, compagnia, entità legale. Devo fari si che l'associazione sia crypto binded, non rompibile.
Non si può risolvere il problema a meno di aggiungere un trick extra: non ho la crittografia attivata: i messaggi sono in chiaro, devo ancora attivare la crittografia.
Mi riferisco ad una terza parte fidata (certification authority), a cui chiedo : il nome è associato alla chiave pubblica?
Ma se la risposta avviene on-line, c'è il problema del MITM di prima.
Prendo il nome, la chiave pubblica e chiedo di firmare digitalmente il messaggio che contiene sia il nome che la chiave pubblica: es Flavia | 123456 diviene un unico messaggio digitally signed dalla certification authority, l'assunzione immancabile è la fiducia nella certification authority.
Certificato:

- Fase 1: sono una banca e voglio generare chiave pubblica e privata, devo farlo offline, non voglio trasmettere online nulla.Genero in locale e salvo in locale la chiave pubblica.
  A questo punto, offline, chiedo alla certification authority di firmare il nome della banca e la chiave pubblica.
  La CA è sicura che sia la banca a richiedere la digital signature: devi presentarti lì di persona, mostrando documenti etc..., fase complessa con aspetti legali.

Infine, mi arriva il certificato CERT = (Bank_Name, BankPK)$_{CA_sign}$, l'integrità del messaggio è garantito dalla digital signature della CA.

- Fase 2: un customer si collega alla banca, e questa mi manda il certificato, che contiene il nome e la pubblic key. In TLS avrò client hello, server hello e poi il certificato che contiene nome e public key che sono cryptographically bounded. Ora il customer deve assicurarsi che il CA sia trusted, nel PC c'è lista trustata: controlla nella lista e se è trustata deve controllare la correttezza della signature. Come faccio per fidarmi della digital signature: Browser - settings - security - manage certificates, si apre un box che contiene la lista di certificati fidati. Quindi cosa vuol dire fidarsi di una CA: la CA firma il messaggio che contiene (nome,PK) della banca, quindi c'è un messaggio e questo messaggio ha un HASH, e se uso RSA questo hash sarà elevata alla private key della CA, la chiave non è dell'entità d i cui si fa il crypto binding ma quella della CA. Quindi per invertire il TAG mi serve la chiave pubblica della CA, ma deve essere pre-installata nel PC, altrimenti sono soggetto ad un MITM. Quindi fidarsi della CA è molto forte: anche la public key della CA è nella forma di un certificato, quindi avrò crypto binding tra il nome della CA e la chiave pubblica. Ma chi firma questo crypto binding? Può essere self-signed, in quanto ci sono root authorities che non possono fare altro se non firmarsi il certificato da sole. In principio non bisognerebbe fidarsi della self-signed certificate a meno che non è firmato da una root authority.
  Trick per sito che vuole un certificato può essere self-signed, ma questo non vuol dire che sia vero.
  spazio utile: ssllabs per testare certificati dei siti.

Ma sono sicuro che sto parlando con la banca? Fin'ora ho solo verificato che la chiave pubblica inclusa nel certificato è associata al nome incluso nel certificato, ma chi mi di dice che il certificato me lo ha mandato la banca? Non è questo il ruolo del certificato: non mi garantisce che sto parlando con quel nome. Un attacco di questo tipo ha senso? Può avere ripercussioni il problema che chi sta dietro il certificato non è l a banca reale?
Quando l'utente manda la chiave pubblica, prima si è riferito al CA che ha fatto il crypto biding della coppia (nome | public key). Ora l'utente può prendere il messaggio e fare replay, ma non può sostituire il nome o la chiave e firmarlo, dovrebbe avere la chiave dell'utente. Quindi il certificato è sicuro: il messaggio è stato prodotto usando la chiave segreta associata al segreto, ovviamente l'attacker non può mandarti un certificato valido perché deve fare verifica offline burocratica.
Attacker non può più sostituire nulla nel crypto binding: l'associazione è forte, quindi risolve problemi della digital signature.
Risolve l'altro problema? Devo provare che l'entità con cui sto parlando possiede la chiave privata associata alla chiave pubblica. Come fare a dimostrare che possiedo la chiave privata associata al certificato? Chiedo all banca di firmare

qualcosa di fresco oppure chiedo alla banca di decryptare qualcosa di fresco. Quindi:

- Banca mi manda certificato con chiave e chiave pubblica, di cui verifico la certezza (è firmato dal CA). Posso ottenere ora la chiave pubblica della banca, e sono sicuro che sia autentica.
- Mando una nonce alla banca e chiedo di firmarlo
- Banca ritorna la nonce firmata con la sua signature.
- Ora applico la chiave pubblica ricavata prima per invertire la nonce che è stata modificata dalla banca per vedere se mi torna.

Ma ora faccio la duale:

- Banca mi manda il certificato
- Io trasformo la nonce usando la chiave pubblica della banca, chiedendo il risultato
- Ora, la banca mi manda il risultato pulito usando la sua chiave privata

Quindi, i certificati non garantiscono che la persona sia reale, ma un protocollo deve includere l'uso di certificati per poter provare l'autenticità dell'entità, in TLS uso i due meccanismi appena visti uno server side ed uno client side.
Approccio di TLS con RSA key transport:

- Ricevo il certificato dalla banca
- Non genero la nonce, bensì la chiave simmetrica che userò dopo in encryption. Mando la chiave criptata con la chiave pubblica.
- Ora posso scambiare dati criptati usando AES-128$_K$: quindi se eri la banca bene, puoi decyptare, altrimenti non puoi.

Difficile però includere diverse root authorities in un singolo PC, idea è di avere fiducia in una catena di certificati: mi voglio connettere alla banca, mi manda certificato firmato da un CA non in lista, quindi posso avere una gerarchia di CA, così vedo il certificato del CA che ha firmato quello della banca e vedo se è trusted.

10.16 Public key certificate

Public key certificate è una struttura dati che fa binding tra una chiave pubblica ed il suo legittimo proprietario. Approccio base: mi affido alle CA, che rilasciano CERT$_I D$ = binding del nome e della public key. esempio: sito web protetto con TLS: voglio proteggere l'URL del sito. Mi fido del certificato che mi viene fornito perché è rilasciato da un CA di cui mi fido. esempio: certificato di Bob: contiene chiave pubblica, CA identity CA$_i d$, CA signature del certificato di Bob.

Devo anche verificare che il server che mi manda questo certificato abbia la private key: mando challenge (nonce) e questa nonce viene firmata con la chiave privata del server.
Ora il protocollo di sicurezza può partire, questo ad esempio avviene in TLS.

10.16.1 Public key infrastructure

Un PKI consiste è il set di tools che serve per creare, distribuire, revocare un certificato per chiavi pubbliche.
Formato tipico per un certificato è X.509, ma per definire un PKI servono molti altri meccanismi: Public key Cryptographic standards, ogni PCKS#n è riferito ad un determinato servizio.
Formato del certificato X.509, definisce tutti i campi ed encoding per un certificato per chiave pubblica:

- version, altri dati: specifica la versione del protocollo (3 è l'ultima) e definisce altre cose:
  - validity period: il certificato è valido per un certo periodo di tempo
  - Serial number del certificato: ogni certificato deve essere unico, identificato con un serial number unico per la CA (è locale quindi per la singola CA)
  - Altre estensioni: posso inserire nel certificato altri parametri, esempio l'uso della chiave nel certificato.
- CA identity: chi è il CA che ha rilasciato il certificato. Rappresentato in modo gerarchico
  - Issuer: è in forma gerarchica, CN è l'ultimo livello, in questo caso è il full name della CA.
  - Subject: CN è URL del sito web che sto certificando (se ad esempio certifico un web server)
- User identity
- User public key: public key, formato dipende dall'algoritmo per cui sto certificando la chiave pubblica. Ad esempio se è RSA avrò i parametri pubblici di RSA, quindi l'esponente ed il modulo N. In DH: avrò quei parametri pubblici.
- CA digital signature: CA prende l'intero certificato e fa hash del certificato e lo firma con la sua private key: è quella legata alla public key della CA. La cosa importante è verificare l'autenticità del certificato: devo esponenziare questo campo con la chiave pubblica della CA e verificare.

Mi collego alla banca: mando HTTP GET in chiaro, ma invece di passare il messaggio direttamente a TCP lo passo a TLS (che è user library in user space) per criptarlo. Quindi parte TLS handshake.
Esponente RSA fissato: è provato che se seguo pattern binario la sicurezza è la stessa ma le prestazioni sono migliori (5 esponenti fissati).

10.16.2 Certificate Signing Request

Una certificate signign request è un messaggio mandato da un applicante ad una CA per avere certificato sulla digital identity.
Posso avere come approccio la generazione di tutte le chiavi fatta dalla CA, quindi anche la mia coppia private,public (cosa che succede nelle VPN), ma non funziona.
Il formato più comune per CSR è PKCS#10. Come funziona:

- L'applicant genera chiave pubblica e privata
- Genero CRS che contiene informazioni che identificano l'applicant, l'X.509 subject feild, le estensioni e la prova che posseggo la chiave privata. Quindi firmo con la mia chiave privata
- CA deve verificare che posso chiedere un certificato per il dominio che richiedo, non è una cosa standard:
  - Manda e-mail all'e-mail trovata come maintainer del domain, c'è un link di verifica
  - CA mi richiede di creare qualcosa sul dominio che mantengo, esempio creare un record .txt per quel dominio

  CA verifica la signature che ho fatto usando la mia chiave privata. Se tutto va bene crea certificato X.509, potrebbe inserire estensioni (a partire dalla versione 3): authority key identifier, subject key identifier, key usage (posso usare la chiave pubblica solo per una specifica azione), alternative names, basic costraint extension (molto importante).

10.16.3 Root certificates

Come certifico le CA? Anche la CA ha un suo certificato, e chi fornisce il certificato è un'altra CA. Struttura gerarchica, la root CA è il livello più alto della certification chain: è una CA che si auto-certifica l'identità. Un root certificate è un certificato in cui subject ed issuer sono lo stesso. Come posso fidarmi di un self signed certificate? Chiunque potrebbe farlo, quindi come faccio a capirlo: root certificates sono built in nel sistema operativo e non possono essere rimossi da utenti senza privilegi.

10.16.4 Certificate chains

In molti scenari reali, il certificato non è rilasciato dalle root CA, che sono poche quindi approccio non scalabile.
Uno o più CA intermedie: ho una catena di certificati, nella catena non ho per forza le stesse entità intermedie. Certificate chain è una lista di certificati che è formata da uno o più CA con una serie di proprietà (solitamente inizia con una end-entity certificate):

- L'issuer di ogni certificato matcha il subject del certificato successivo della lista
- Ogni certificato

A volte web server non mandano root certificate (perché può non essere nel SO). Ma il chaining è pericolo? Ho una catena di certificati validi, posso generarne uno fake: creo un certificato fake ma legittimo e lo firmo con il CA finale (che sarebbe ad esempio il mio server).
Non è possibile: c'è un check, il basic constraint extension. Se nel campo certificate authority c'è NO (false) (e questo c'è nel certificato dell'end user) vuol dire che con la coppia chiave pubblica|privata non posso firmare altri certificati.
Wildcard certificate: Certificato valido per tutti i sotto dominii di tutto un dominio, esempio *.google.com perché in pratica una compagnia ha diversi servizi associati ai vari sub domains.
Periodicamente, le CA devono rilasciare una lista di revoca dei certificati, che gli utenti possono controllare. Come faccio ad ottenerla: inserisco la lista in una estensione specifica, distribution point da cui è possibile scaricarla.

10.16.5 Let's build our own authority

OpenSSL x.509: OpenSSL è toolkit crittografico composto da 3 componenti (librerie scritte in C). openssl ha variante di RSA basato sul teorema cinese del resto.
Alcune applicazioni vogliono un unico Ca certificate bundle con tutti i certificati: quindi metto tutti i file in uno solo, concatenando il root e l'intermediate.
Ora voglio provare ad usare Apache2 per configurare l'uso del certificato: proteggo il server http con i certificati che ho creato.
Apache2 supporta il meccanismo del virtual host: se voglio installare più web server su un unica macchina, avrei bisogno di più ip. Oggi posso avere più siti web su un unico web server con i virtual host.
ServerName è un field importante, serve per abilitare il virtual host.
Mi collego via browser e ricevo un warning: il certificato è valido, il browser/SO non ha la chiave pubblica della CA. Devo specificare esplicitamente https, nei siti noti c'è redirect automatico, voglio averlo anche io: lo metto nel file configurazione di apache2.

10.16.6 HTTPS Downgrade Attack

Siti ibridi, homepage in HTTP e login in HTTPS e questo era una vulnerabilità, homage non protetta è vulnerabile ad un impersonification attack: mi metto nel mezzo tra server e vittima, performo HTTPS con il server ma HTTP con la vittima.
Posso pensare ad un downgrade attack: identifico la vittima:

- Mi metto in mezzo alla vittima ed il default gateway.

- Faccio redirect dei pacchetti che hanno l'IP del sito localmente.
- La vittima si collega a me e replico con una versione del website.
- Faccio fare l'autenticazione

HSTS: Http Strict Transport Security: se provo a collegarmi ad un server con http, il server mi risponde di usare HTTPS, risponde con un cookie che specifica che devo usare https ed ha un tempo di fine (1h). Prima query è in chiaro, quindi da un punto di vista teorico la vulnerabilità rimane.
Alcuni browser web hanno incluso una lista di siti che contiene i siti più noti che supportano HSTS.
Non tutti i client supportano HSTS e c'è comunque vulnerabilità a DNS Spoofing Attack.

10.17 Diffie Helmann protection

Anche qui avevo il problema del MITM: Alice e Bob hanno g, p, e fanno exponentiation della loro chiave privata random x ed y.
Attacco è possibile: attacker selezione z e manda ad entrambi $g^z modp$ e quindi entrambi mandano la loro chiave all'attacker, quindi il data exchange passa per l'attacker nel mezzo (può ricavarsi tutte e due le chiavi per cifrare i messaggi e mandarli ai due end).
Senza dare nulla, l'approccio DH può essere attaccato via MITM ed è detto Anonymous DH.
IETF, protocollo BTNS (versione light di IPsec, Bettern Than Nothing Security), che è il DH Anonymous: so che c'è una vulnerabilità ma è meglio che non sapere nulla. Approccio base di DH è questo, quindi vorrei fixare il protocollo:

- Alice e Bob scelgono x ed y e chiedono alla CA di certificare i valori. Ovvero ottengo crypto binding tra il nome ed il valore di chiave pubblica. Ma x non è una chiave pubblica, bensì una quantità pubblica che corrisponde ad una quantità privata che solo loro hanno.
- Chiedo alla CA di ricevere qualcosa che è [Alice, $g^x modp]_C A$ e Bob, $g^y modp]_C A$.
- Ora attacker può generare z, ma non può sostituirsi ad Alice o Bob, perché non può ottenere il certificato.

Questo è il fixed DH exchange. Ho un altro problema: i valori $g^x modp$ e $g^y modp$ ora sono fissati, suppongo che mi collego alla banca nel 2018. Quando mi collego, avviene scambio dei certificati, il segreto che computo è $g^{xy} modp$. Mi collego nel 2020, ma non posso cambiare quantità x, perché il certificato è valido per una durata di anni (è processo legale che richiede vari step, non puoi farlo in 5 min): ogni volta che mi collego, computo sempre lo stesso segreto, quindi la chiave è sempre la stessa. Questo può o non può essere un problema, ma non

è una best practice: ci sono nazioni che fanno attacchi severi, si salvano il tuo traffico per anni. Log del traffico dei cittadini: è criptato, ma aspetto per un tempo ragionevole e portò rompere la chiave privata dal tuo pc e scoprire il tuo traffico criptato.
Protocollo che non permette questo soddisfa la proprietà perfect forward secrecy.
vorrei non usare quantità fisse: se non faccio nulla, sono vulnerabile (DH anonymous), se certifico, ho fermato il MITM ma uso sempre lo stesso segreto nell'agreement (g^{xy}). Cosa posso fare: Alice genera chiave pubblica standard usata in una digital signature e ha il certificato della CA. Ma ora posso dire che il DH public coefficient ed la mia identità e firmarla da solo. Bob può prendere il certificato, prendere chiave pubblica di Alice, che si aspetta sia certificata dalla CA. Non è un certificato self signed: Bob riceve due quantità da Alice, uno è il certificato vero e l'altro è la DS del coefficiente pubblico di DH.
Così:

- Evito MITM: l'attacker non può computare nessuno dei due certificati.
- Posso generare un segreto fresh ogni volta.

Ephemeral DH: ho due certificati (A, $g^x modp)_S ka$, che è dinamico, viene generato localmente e il Ska cambia sempre; (A, Pka)$_{CA}$. Garantisce perfect forward secrecy? Trump mi fa rompere la CA, può vedere le mie connessioni passate? No, perché anche se qualcuno frega la chiave privata della CA, il segreto $g^x modp$ va perduto perché le x ogni volta cambiano.
Quindi anche se la chiave privata viene scoperta, la sicurezza degli scambi precedenti è sicura.
Non c'è un singolo DH, ma 3 versioni:

- DH, MITM probelm
- Fixed, long term secret
- Ephemeral, la migliore

In TLS:
DH senza nulla è la versione fixed, ma devo specificare l'algoritmo usato per la digital signature dalla CA: può essere RSA, DSS etc...

10.18 Symmetric vs Asymmetric

Nell'handshake, quando viene mandato il certificato:

- In DH non c'è certificato, solo 1 messaggio
- In fixed DH mando dopo il server hello il certificato e poi verifica di client
- In RSA key transport: mando il certificato e niente altro.
- Solo in ephemeral DH uso tutti e due i messaggi: mando certificato della Pk ed nel server key exchange mando g^x firmato da me.