

Full Crittografia Applicata

Owner	maxbubblegum47
Tags	

Crittografia Applicata

[01/03/2023]

Pagine

- [Random e Pseudorandom for Cryptography \(1\)](#)
- [Symmetric cryptography: derivates and applied schemes \(1\)](#)
- [Authentication Protocols and Cryptographic schemes for password protection \(1\)](#)
- [Crittografia asimmetrica \(1\)](#)
- [Distribuzioni Chiavi Pubbliche \(1\)](#)

Generali

Proposta di libro di testa da seguire per chi è interessato. Non sono previste ore di laboratorio vere e proprie, ma si tutto in maniera trasversale. Si utilizza Moodle in generale. per chiedere ricevimento su richiesta scrivi “[CPA] Chiarimento spiegazione lezione/richiesta di ricevimento”

Materiale

Slides sicuramente e libro di testo: Real-World Cryptography di David Wong. Chiamata anche applied cryptography che cerca di dare una spiegazione della crittografia come applicazioni reale. Per altri argomenti verranno dati anche altri materiali. Per alcuni punti non saranno coperte quindi alcune informazioni da tutto. Ci saranno puntatori di varia natura. Le slides sono in inglese, ma il corso è in italiano. Password delle slides: `kahngu9Eihe4`

Esame

Si fa un appello orale, con due appelli a giugno, uno a luglio, settembre, gennaio e febbraio. Ci si potrebbe cimentare in un approfondimento in sede d'esame. Questo vuol dire che durante lo svolgimento della materia ci saranno dei momenti di leggero approfondimento che possono essere approfonditi in autonomia ed eventualmente portati in sede d'esame. Si potrebbe strutturare la esposizione di un argomento a scelta. Questo approfondimento può essere scelto da una lista proposta dal docente, ma una persona può anche proporre un argomento. Basta che sia tutto univoco. Verranno proposte cose molto variegate, come protocolli in fase di studio oppure aspetti anche molto specifici su aspetti di modifica di schemi standard per rispondere ad attacchi avanzati che ancora non è possibile gestire. Oppure anche cose più business oriented, con materiale e soluzioni proposte da azienda di cui abbiamo materiale e quindi alla fine è tutto studio di servizi esistenti. Ci sarà a partire da un mese dall'inizio delle lezioni tendenzialmente. Una persona può fare retro front? In generale pensiamoci bene, l'unica cosa è che chi prima arriva meglio alloggia, ma non siamo avari per prenotarci argomenti. Cerchiamo di essere in buona fede ecco. Se decidiamo di non fare quel tipo di approfondimento non fa nulla, ma tendiamo di non farla diventare una prassi.

Introduzione

La crittografia applicata è spesso definita come applied cryptography. Si parla di uno stack di primitive, schemi e protocolli sicuri. Vedremo come tanti oggetti lavorano a diversi livelli di dettaglio rispetto a che cosa vogliamo rendere sicuro. Staremo sopra le primitive come livello e vedremo molto bene i protocolli. Cerchiamo di tenere un

livello di astrazione che possiamo tenere vicino al mondo reale. Si cerca di dare linee guida su quali sono le principali chiavi di lettura per studiare questi oggetti e poi avremo una overview di alcuni argomenti che vedremo durante il corso. Verrà discusso della prima parte di partenza, di alcune concetti fondamentali.

La crittografia a livello gergale si confonde con i sistemi di codifica o descrizione dell'informazione. Vedremo come crittografia non è solamente mandare le info con un altro tipo di meccanismo, ma trasformarlo secondo certi criteri non permettere il recupero di informazioni da eventuali testi. L'idea è manipolare i dati in modo da proteggere l'informazione. La stenografia invece invia messaggi e assumiamo che l'attaccante non capisca che in quei messaggi ci siano altre informazioni.

Ci sono tecniche per nascondere informazioni all'interno dell'immagine di modo da non rivelare subito la codifica dei dati all'interno di una determinata immagine.

Tecnica che punta al fatto di non scoprire che ci siano i dati. Queste due sono discipline separate, ma possono collaborare anche se solitamente in contesti la crittografia è molto più importante e ha molte più applicazioni in contesti moderni. Materia duale è la cripto analisi che cerca di rompere le tecniche che vediamo in crittografia e stenografia.

Viene definita come crittografia classica fino al 1900. Ci sono stati vari step che hanno segnato lo sviluppo della crittografia:

1. invenzione del telegrafo (1700-1800)
2. utilizzo della radio (1900)
3. internet (1980)

Questi sono metodi molto più comodi ma sicuramente più rischiosi in termini di sicurezza. Nell'ultimo caso abbiamo proprio degli attori terzi a cui affidiamo le nostre comunicazioni. Tutti questi sono potenziali avversari. Si stava più sicuri quando si stava peggio. Con la storia parte nell'antica Grecia con lo **scitale**. Questo essenzialmente è una trasposizione colonnare, per cui usiamo una trasformazione per andare a creare un messaggio cifrato. Abbiamo nel messaggio finale tutto quanto quello che serve. Non è però molto difficile rompere il cifrario. La sicurezza al tempo nel metodo di cifratura richiede nella segretezza del metodo, cioè il problema è sapere che algoritmo è stato usato per ottenere questo tipo di messaggio. Nel mondo moderno dobbiamo cercare di fare un pochino meglio. Sicuramente il cifrario più famoso al tempo è stato il cifrario di Cesare che gli permetteva di spostare le lettere di 3 valori, spostando sempre tutto e trasformando il messaggio originale. Possiamo vedere questa operazione come somma modulare $c = (m + k) \text{mod} 26$

Esempio:

$k = 4$

$ITALY \rightarrow [8, 19, 0, 11, 24] \rightarrow [12, 23, 4, 15, 2] \rightarrow MXEPC$

In media in 13 tentativi (conoscendo il metodo ce la posso fare).

Crittografia classica

Assumiamo che il metodo sia segreto e poi c'è da dire che nella storia l'unico scenario in cui si è cercato di proteggere informazioni è stato fatto in riferimento a degli avversari passivi.



Avversario passivo = solamente read only

MANCA LA SECONDA PARTE DELLA LEZIONE RECUPERA DALLE SLIDES

[02/03/2023]

Security guarantee of secure communications

Siamo in uno scenario in cui vogliamo comunicare con alice, ma abbiamo eve in mezzo che cerca di intercettare e comprendere le comunicazioni. Ad implementare uno schema di cifratura abbiamo alice con una funzione encrypt, una decrypt e entrambi una funzione k. Siamo in uno schema simmetrica. Che cosa vuole dire che siamo in uno schema sicuro? Come garantiamo la sicurezza delle informazioni?

La parte fondamentale che dobbiamo capire è: **che cos'è la sicurezza?** Questa va identificata e definita in modo più o meno rigoroso, più o meno formale. Se non la definiamo non possiamo nemmeno capire come fanno le funzioni cifra e decifra. Se non capiamo da cosa dobbiamo difenderci non sappiamo nemmeno cosa progettare. Vogliamo capire, intuire, concepire, cosa vogliamo garantire e poi dobbiamo modellarlo. Possiamo concepire che cosa sia e posso creare schemi di crittografia, ma il nostro sistema finale potrebbe esser insicuro perché la definizione fin dall'inizio non è corretta. Se sbaglio a definire che cosa è sicuro poi tutto quello che c'è dopo crolla, inoltre se la modellazione è stata fatta bene, noi possiamo usare male gli strumenti di difesa. Ogni schema di cifrature risponde a caratteristiche diverse e dobbiamo capire rispetto a quali modelli i nostri schemi sono sicuri in termini di garanzie e scenari. Vogliamo riuscire a capire che tipo di sicurezza stiamo considerando per riuscire a capire gli standard esistenti.

Immaginiamo l'attività di modellazione. Nella slide 5 abbiamo alcune informazioni su libri utili in termini di definizioni.

Simulation Based

Facciamo modellazione immaginando oggetti ideali che anche se non possono esistere li concepisco per come sarebbero secondo noi, in termini di costruzioni ideali per quello che vorrei ottenere in termini di sicurezza. La parte di prova riguarda il proporre effettivamente delle costruzioni che soddisfano o comunque si comportano come l'oggetto ideale all'interno del mondo reale. *Porto una idea dall'iper uranio alla terra essenzialmente*. Non ho una modellazione standard per andare a fare una cosa super safe, ma sono il primo passo per concepire concretamente questi schemi. Parliamo di una costruzione che magari non è l'oggetto ideale, e ho dei limiti che non supero, ma Eve non riesce a distinguere la costruzione reale dall'oggetto ideale. Cerchiamo di trattare il problema di alice, bob e eve. Dobbiamo capire cosa vuole dire essere al sicuro da eve. Che cosa potrebbe fare eve? Il primo modello di sicurezza che prendiamo in considerazione è quello in cui eve è un **passive eavesdropper (EAV)**, acronimo noto in letteratura): attore read only del traffico in transito che vuole informazioni riguardo il traffico originale. In questo caso Eve non fa nulla legge solamente. Considero che ho più leve su cui agire. Qui Eve non fa nulla, legge solamente quello che passa. Il nostro oggetto ideale è un attaccante che potrebbe fare solamente queste cose. Non sceglie niente Eve. La seconda parte è: che cosa voglio andare a proteggere? Che cosa vogliamo che Eve riesca a scoprire? Cerchiamo di garantire che concettualmente Eve non riesca a distinguere i dati cifrati da dati random. Questo concetto è a se stante. Il fatto che questo sia in modo utile il concetto di confidenzialità ci permette di ragionarci sopra, ma non c'è un vero motivo dietro, non vogliamo fare scoprire ad eve alcuna informazione ed è una modellazione di questo tipo. Il fatto che questo concetto nel mondo reale significa proteggere la confidenzialità è un'opera di modellazione che l'esperienza ci ha dimostrato essere affidabile → **indistinguibilità (IND)**. Per esempio vedremo come uno schema potrebbe essere IND-EAV. Le tecniche moderne non si preoccupano di nascondere la dimensione dei dati. Se ci interessa coprire anche quella informazioni ci dobbiamo arrangiare noi facendo del padding per esempio o usare schemi di codifica.

Un esempio classico è quello di avere dei messaggi che coprano "si" o "no", magari in inglese che hanno dimensioni diverse. Se mantengo fedelmente la dimensione del testo in chiaro posso capire bene che cosa c'è dietro. La sicurezza dello schema si rompe subito. Uno schema perfettamente sicuro, potrebbe essere insicuro in termini

di obiettivi finali perché ci dimentichiamo che alcune parti non sono garantite dallo schema di cifratura. Dobbiamo saperlo usare correttamente. Lo schema non protegge la dimensione.

Il nostro attaccante di tipo EAV vuole rompere IND del testo cifrato, che cosa vuol dire che il nostro schema ci protegge? Dobbiamo inventarci una parte chiamata *experiment* che sono uno o più algoritmi che l'avversario esegue in collaborazione con gli attori legittimi del sistema. La persona onesta la definiamo come challenger l'esperimento riguarda IND-EAV funziona come segue:

1. challenger sceglie un messaggio con il 50% da uno scelto random e il 50% da uno fatto da uno schema di cifratura che stiamo studiando. Sceglie tra cifrati e random praticamente
2. Da il messaggio all'avversario che è di tipo passivo e vuole avere informazioni; questo deve decidere se questo è dato random o output di uno schema di cifratura. Devo capire se ho spazzatura o dati random. Vince se capisce che cosa è spazzatura o dati random (indovina correttamente)

Questa è solamente la definizione di esperimento. La sicurezza la definisco dopo, quando effettivamente l'avversario ha una certa probabilità di vittoria. Un avversario può vincere, ma quale probabilità ha di vincere effettivamente. Identifichiamo due tipi di sec:

1. **perfetta (unconditional security o information theoretic security)**: avversario vince al massimo al 50%, che è la probabilità casuale
2. **computazionale**: se l'avversario riesce a vincere al 50% + (*quantità trascurabile*)%

Unconditional security and one time pad

Contento di cifratura simmetrico. Schema che si basa sull'operazione di XOR, che praticamente è somma modulare modulo 2. Questo è molto bello perché dato "m" qualsiasi, se "k" è scelto in modalità random, allora la probabilità del testo cifrato "c" di essere 0 o 1 è 50%. Questo lo capiamo vedendo la tabella di verità. Fissato m, prendendo random k, ho sempre 50% di possibilità per andare ad avere un 0 o 1 in output, non ho alcun tipo di bias.

Cifrario di Vernam

Ho un **chiave** che è lungo tanto quanto il messaggio e la cifratura è letteralmente lo XOR tra chiave e messaggio. Praticamente ho uno XOR bit a bit che è sempre

random e non sto dando alcuna informazione utile all'attaccante. L'operazione di decifratura è praticamente identica. Se faccio lo XOR del c e k riesco a riprendere m. Molto forte per IND-EAV, ma per attaccanti diversi come quelli attivi che manipolano le informazioni in transito potrebbe essere un problema. Uno schema di cifratura fatto in questo modo è detto completamente malleabile perché l'attaccante modificando un certo bit nel testo cifrato sa esattamente quale posizione del testo in chiaro sta andando a toccare. Vedi come in base ai modelli di sicurezza io sia completamente insicuro. Non devo riusare la chiave, avere pattern ripetuti. Se cifro due messaggi con la stessa chiave posso andare ad ottenere due messaggi xorati, perché mi appoggio alla proprietà della xor ed escludo la k. Tutto ciò banalmente a partire da c_1 e c_2 .

Concetto di dato random

Dobbiamo distinguere dato random in sicurezza e dato random statistico. Per chi lavora in statistica la randomicità è quella di avere una distribuzione uniforme. Questo non è sufficiente in contesti di sicurezza, è richiesto un concetto di "non predicitività". Anche se osservo una sequenza di bit non ho alcuna informazione per capire quale potrebbe essere il prossimo valore. non riesco a prevedere se il successivo valore che gli verrà dato sarà effettivamente 0 oppure 1. In ambito statistico potrebbe esserci una chiave random, solamente per la distribuzione, ma è solamente una cosa statistica. Se guardassi algoritmi che riguardano generazione di dati random.

Sicurezza perfetta e sicurezza reale: principi di Kerckhoffs



Principi di Kerckhoffs:

- algoritmi pubblici
- segretezza della chiave
- uno schema potrebbe non essere perfetto, ma non deve essere fattibile rompere lo schema

La chiave deve essere abbastanza grande da prevedere attacchi brute force. I dati devono essere indistinguibili da dati random e questo deve essere valido per ciascun tipo di dato in chiaro che si decide di andare a cifrare. Quando parliamo di sicurezza computazionale, il primo oggetto ideale che prendiamo in considerazione è chiamato: pseudo random function. Questo oggetto teoricamente non esiste, ma se esistesse sarebbe una funzione come descritta → quest'ultima funziona effettivamente e ci da una sicurezza computazionale. Immaginiamo una funzione come la seguente = $F : K \times P \rightarrow C$

dove:

- K è **uniformemente** distribuito su $\{0, 1\}^{Lk(n)}$
- P è il plain text con la stessa probabilità $\{0, 1\}^{Lc(n)}$
- l'output ci soddisfa come scenario IND-EAV in termini computazionalmente in quanto la quantità trascurabile è espressa in una funzione di **negl(n)**

controlla bene questa ultima affermazione (28 marzo 2023)

| n sarebbe la lunghezza dello schema

Questo potrebbe essere un one time pad. Vogliamo che n sia abbastanza grande per poter resistere a brute force, ma abbastanza piccola per poter essere gestibile (non voglio memorizzare delle chiavi che sono infinite per andare a cifrare tutti i miei documenti)! Dobbiamo capire che abbiamo alcuni parametri per andare a configurare questi schemi che hanno moltissimi effetti tangibili.

La dimensione del testo in chiaro che vado a cifrare $Lc(n)$ non è molto efficiente tecnicamente e sarebbe anche molto grande, ma se fosse anche la chiave inefficiente sarebbe un bel problema in termini di usabilità.

Uno schema di cifratura sicuro deve avere una dimensione della chiave espressa da funzione efficiente, ovvero che **ha un costo polinomiale**. Diciamo efficiente se il loro costo espresso con una formula è di tipo polinomiale. Una funzione **negl** deve poter essere espresso come inverso di **esponenziale o super esponenziale**. Voglio

avere uno schema con un k che cresce rispetto ad una formula di questo tipo (polinomiale sicuramente). La funzione **negl** deve invece crescere con una formula (come segue intorno alle slide 20).

Il punto fondamentale è che il fatto di identificare due tipi di funzioni mi permette di avere degli algoritmi in cui regolando un certo parametro regolo la sicurezza del mio sistema. Devo avere il migliore n per tutte le funzioni e al fine di avere $ngl(n)$ che sia quanto più basso possibile.



Perché ngl è inverso polinomiale? Voglio avere una situazione in cui ho due funzioni che divergono molto in fretta e ho uno schema efficiente.
Altrimenti dovrei sempre scegliere chiavi molto alte.

In realtà questi parametri di n hanno anche effetti sui costi di calcolo degli algoritmi. Immaginiamo di avere degli algoritmi di cifratura e decifratura che hanno i loro costi. La sicurezza computazionale deve avere delle formule di tipo efficiente con costi di tipo polinomiale, invece l'algoritmo che usa Eve deve avere sempre un costo inefficiente esprimibile con costo **negl()**. Da una certa dimensione della chiave in poi i costi di Eve diventano realmente intrattabili. Immaginiamo la chiave grande n e la chiave costi n . Per rompere ho 2^n , mentre cipher e decrypt sono inefficienti.



Legenda

ngl = inverso inefficient
efficiente = polinomiale
inefficiente = esponenziale

Il costo per calcolare encryption e decryption cresce molto piano rispetto alla dimensione della chiave, il costo per rompere lo schema cresce con la dimensione della chiave. Voglio una divergenza che sia comoda per me e scomodissima per Eve.

Rispetto a questo abbiamo dei parametri fissati precisi rispetto al parametro n . Questo viene stabilito da chi studia in dettaglio queste cose e abbiamo 3 valori chiave:

- 80 bit security → not secure since 2010. Circa 2^{80} operazioni per rompere lo schema
- 112 bit security → sicuro fino al 2030

- 128 bit security → sicuro dal 2030 in poi

Ad oggi si tende a 112 come valore di riferimento. Questi parametri esprimono il numero medio di operazioni necessario per rompere uno schema di cifratura in senso logaritmico. Se lavoriamo simmetricamente l'attacco più efficiente per Eve è il brute force sulle chiavi. In uno schema simmetrico ho delle funzioni di cifratura progettate in modo tale per cui questo è il modo più veloce.

- 128 bit key → 128 bit security (2^{127} operazioni da fare)

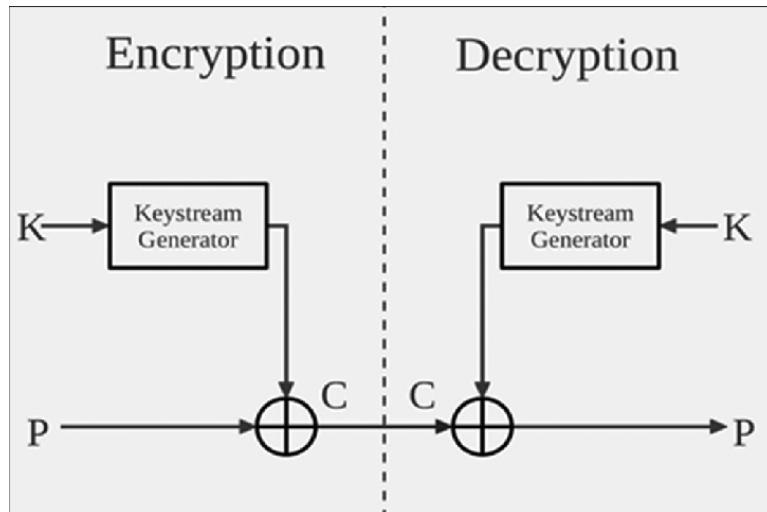
In un livello di sicurezza intermedio è utile se vado nell'asimmetrico, perché in questo non ho più il brute force, ma l'esecuzione di alcuni algoritmi:

- 1024 bit rsa → 80 bit security
- 2048 bit rsa → 112 bit security
- 3072 bit rsa → 128 bit security
- 256 bit p256 ecc → 128 bit security

La dimensione della chiave RSA non aumenta linearmente con il livello di sicurezza. Con gli schemi che vedremo: cifratura con curve ellittiche hanno costi in cui la dimensione della chiave è due volte il livello di sicurezza. Per gli schemi asimmetrici dovremmo andare a studiare la matematica che abbiamo dietro. Immaginiamo di avere dei costi che governano le funzioni e questi sono espressi da formule differenti con funzioni che variano diversamente. Queste relazioni sono legate al **miglior attacco noto**. Se c'è un evento *disruptive* in termini di "ricerca per rompere algoritmi di cifratura", allora cambio tutto → lo schema diventa insicuro quando cambia il tipo di formula che esprime l'attacco più efficiente per l'attaccante.

Se vedi le considerazioni di RSA in cui era stato ideato si diceva di scegliere una chiave ad numero di bit per avere 60 bit di sicurezza. Poi sono stati scoperti tanti attacchi più efficienti e sono state cambiate le raccomandazioni. Si considera che il quantum computing potrebbe andare a gestire nuove classi di algoritmi che potrebbero rompere gli algoritmi di cifratura simmetrici odierni.

Stream Cipher



Il primo tipo di schema pratico è chiamato stream cipher. Oggetto che si avvicina molto alla pseudo random function, in quanto si mappa quasi direttamente sulla pseudo random function. Termine concettuale (cifrario a flusso), ma tecnicamente è un **deterministic pseudorandom bitstring generation (DPGB)**. Data una piccola fonte di dati random genera una grande quantità di dati random, la caratteristica è che sono dei dati generati da algoritmi di questo tipo che non possono essere distinti da dati random (*in teoria secondo il modello, poi nella realtà non è sempre così semplice*). Questi dati in presenza di contesti IND-EAV non possono essere distinti da dati random, ed un generatore di questo tipo è effettivamente uno stream cipher se non ho dati predibili.

In questo tipo di schema introduciamo il concetto di pseudo random e di bit stream; lo stream cipher non è altro che un algoritmo che mi permette di fare *onetime pad*, ma usabili anche nel mondo reale. Praticamente ho un *onetime pad*, ma la chiave non è effettivamente random, con dati presi nativamente casuali da fonti fisiche (vedi [/dev/random](#)), ma sono l'output di un algoritmo deterministico. Ho tendenzialmente una chiave a 128 bit o 256 bit che con lo stream cipher espando tantissimo per poter andare a cifrare grandi quantità di dati.

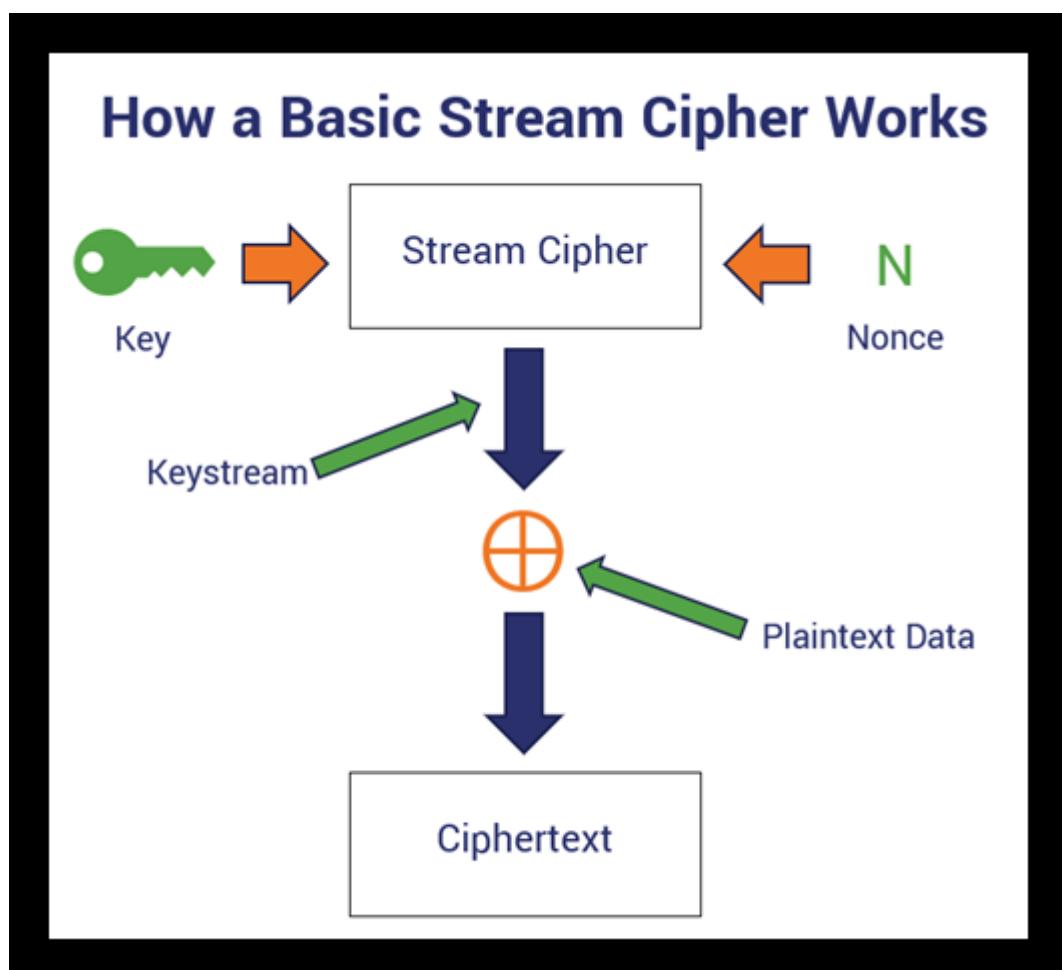
Sono in una situazione in cui ho: key + nonce = stream cipher. Poi praticamente metto a xor la stream key con il plaintext e ottengo il ciphertext. Devo pensare che sto usando schemi per cifrare tantissimi messaggi. Se usassi solamente la stessa chiave avrei lo stesso problema visto prima, cioè che l'attaccante può, sfruttando il funzionamento dell'xor e della mia disattenzione, riuscire ad avere lo xor di due messaggi in chiaro.

Gli schemi di cifratura moderni devono supportare il fatto di cifrare più messaggi con la stessa chiave. Per fare questo posso andare ad usare il nonce.



Nonce = number used once

Nonce lo dobbiamo sempre inviare allo stream cipher un valore differente, altrimenti creo due volte la stessa stream key. Ovviamente non devo andare a riusare lo stesso nonce più volte altrimenti sarei esattamente allo stesso punto del problema di prima in cui usavo la stessa chiave per più messaggi.



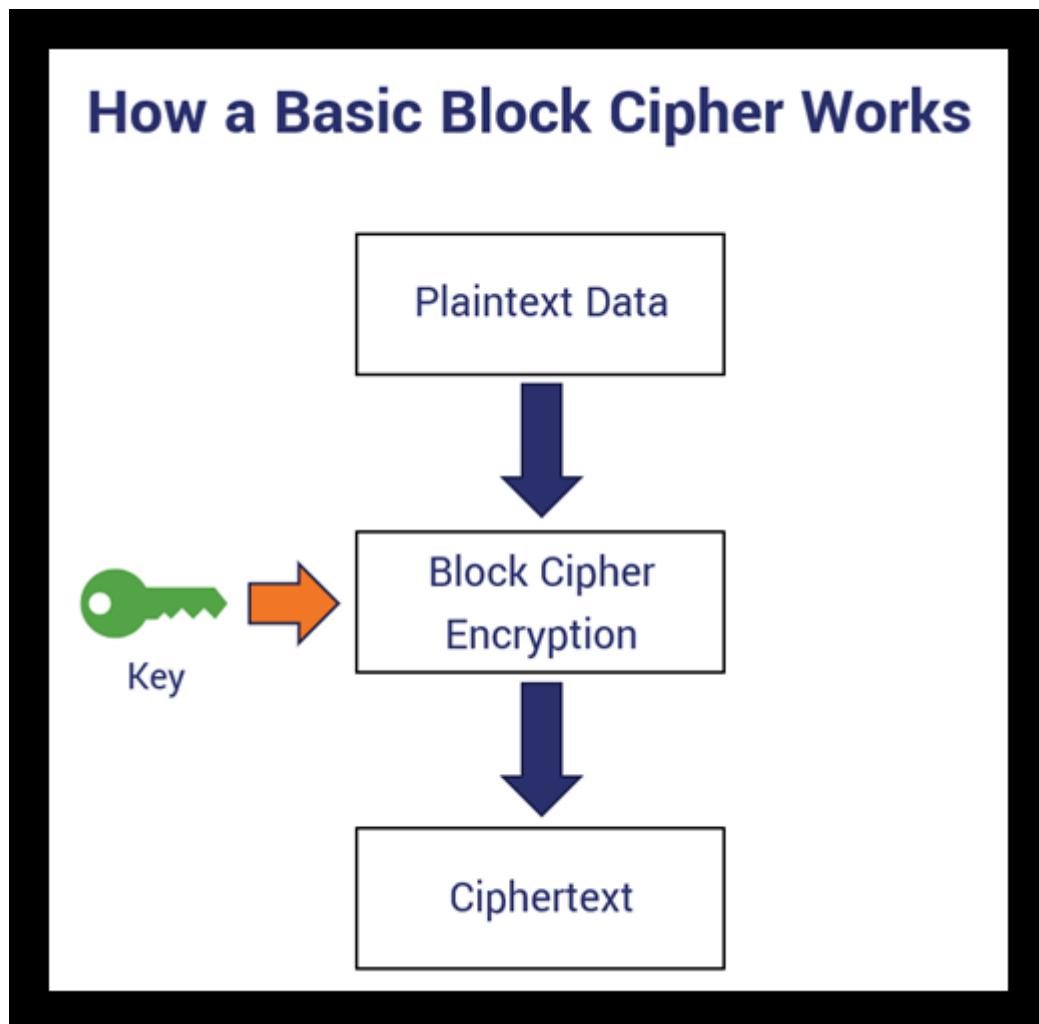
Standard importanti

ChaCha20: uno standard importante nel mondo degli stream cipher (dal 2005) che è presente in tutti i computer general purpose (vedi RFC7539 del 2008). Principale alternativa a AES. Abbiamo molti stream cipher rotti che sono stati rotti tantissime volte: *rc4*. Controlla sempre gli standard perché potrebbero cambiare in fretta e ci si potrebbe fregare da soli non controllando questo genere di cose.

[08/03/2023]

Vedi algoritmi che sono resistenti ai computer quantistici

Modellazione dei Block Cipher



Block cipher famiglia di permutazioni pseudo random. Il testo generato in uscita ha la stessa dimensione del testo in chiaro. Il dominio del dato di output è una permutazione del dominio del dato in input. Lo stream cipher lo uso in modo diretto, il block cipher invece a volte lo uso a volte solamente in una direzione, ma spesso anche con l'operazione inversa. L'esempio classico: immaginiamo di avere un alfabeto e immaginiamo che il block cipher ideale è un oggetto che è praticamente una mappa con una corrispondenza biunivoca tra una lettera dell'alfabeto ed un'altra scelta a caso. Idealmente avrei una permutazione davvero random, perché non ho alcuna corrispondenza matematica che mappi ogni lettera in chiaro sulla lettera cifrata, in questo caso il **cifrario stesso è la mappa**. Trasforma un elemento in un altro dello stesso dominio in maniera completamente sciolgente. Chiaramente non è utile in pratica perché dobbiamo ragionare in termini di dimensione. Immaginiamo di doverlo usare: entrambi lo devono conoscere, i due attori di una comunicazione, e pensiamo a quanto sia grande in termini di bit: $n\text{-dati} \times \log_2(26)$ bits

Immaginiamo come nella slide 40, di avere il numero dell'indice come chiave possibile. Possiamo capire quanto sia però grossa questa tabella da salvarci. Pensiamo che se vogliamo trasferire direttamente la mappa, in termini però solo di corrispondenza del numero 15. Questa riga a cui si riferisce l'indice è grande $n \times 2^n$ e l'intera tabella sarebbe $n \times 2^n \times 2^n$!

Example of Ideal Block Cipher with block size = 2

- Build a table that includes all possible permutations of 2 bits data (block size $n = 2$)

00 01 10 11	00 01 10 11	00 01 10 11	00 01 10 11
0: 00 01 10 11	6: 01 00 10 11	12: 10 00 01 11	18: 11 00 01 10
1: 00 01 11 10	7: 01 00 11 10	13: 10 00 11 01	19: 11 00 10 01
2: 00 10 01 11	8: 01 10 00 11	14: 10 01 00 11	20: 11 01 00 10
3: 00 10 11 01	9: 01 10 11 00	15: 10 01 11 00	21: 11 01 10 00
4: 00 11 01 10	10: 01 11 00 10	16: 10 11 00 01	22: 11 10 00 01
5: 00 11 10 01	11: 01 11 10 00	17: 10 11 01 00	23: 11 10 01 00

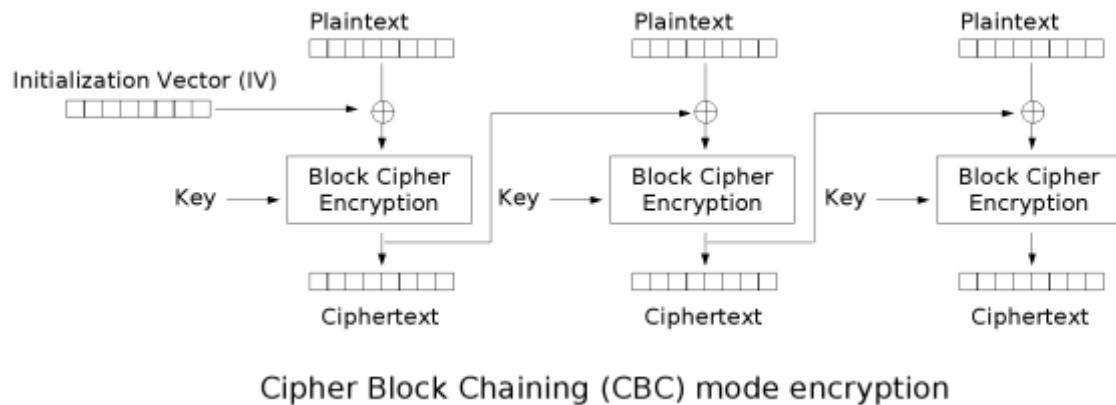
The **key** acts as the **selector** of the mapping to be used

- Size of the table for n -bit?
- **The table is the ideal block cipher**
 - a raw is transferred as the key?
 - $n * 2^n$ → too large (grows exponentially and thus it is inefficient)
 - if the table is public and the index of the raw is the key
 - $n * 2^n * 2^n!$ → it is impractical to store the whole table

Ci rendiamo conto che la dimensione della chiave non è per nulla efficiente, pensa al teorema di Kirchhoff che parlava di efficienza della chiave.

Il block cipher è una primitiva su cui però possiamo andare a costruire uno schema di cifratura, non possiamo usarlo direttamente come lo stream. Questa contesto viene definito come *operation mode* (modalità di cifratura). Questo è un algoritmo che sfrutta un block cipher per costruire uno schema di cifratura come quelli che abbiamo visto la settimana scorsa. Nel mondo reale un block cipher reale può essere per esempio: AES (advanced Encryption Standard, dal 2001 è uno standard definito dal NIST). Semplicemente il nome AES definisce una famiglia di block cipher che definisce una serie di varianti: 128, 192, 256. Queste sono tutte particolari istanze che definiscono la dimensione della chiave. Se vogliamo lo schema di cifratura dobbiamo aggiungere una nomenclatura del tipo: AES-128-CBC, sappiamo

che ho uno schema di cifratura basato su AES128 che tramite una modalità CBC crea effettivamente uno schema di cifratura usabile all'interno del mondo reale.



AES esiste in tre varianti e la dimensione del blocco è fissa: 128 bit, non di più ma nemmeno di meno. Non possiamo dargli 100 bit, ha senso solamente per 128 bit alla volta perché è una permutazione che voglio in questo modo e basta.

Questo è AES, non facciamo design interno, ma la parte fondamentale è confrontarlo internamente rispetto all'oggetto ideale. AES rimpiazza interamente DES e 3DES, che era il precedente standard definito negli anni 70. DES esiste anche come 3DES, come block cipher costruito a partire da 3 volte DES.

[sono uscito 5 minuti magari riprenditi le slides]

finisci di vedere queste slides mancanti da

[02-symmetric-encryption.pdf](#)

Concetti importanti per gestione block cipher:

1. Diffusione
2. Confusione
3. Extra-extra-extra avanzato: "*the design of the Rijndael the advanced encryption standard*" come libro se si è interessati, ma è veramente stra tecnico

Una considerazione interessante è che un block cipher non è uno schema di cifratura se parliamo di andare a cifrare qualsiasi tipo di dato. Abbiamo però dei contesti in cui sono usabili direttamente come schemi di cifratura, come situazione

KEM → metodi di incapsulazione di chiave
(https://en.wikipedia.org/wiki/Key_encapsulation_mechanism). Quando nativamente ho un dato di input di dimensione fissa e voglio cifrare per incapsularla allora potremmo dare questa chiave direttamente in pasto al block cipher senza andare ad usare alcuno strumento di contorno.

Esiste un teorema PRP/PRF che in questi contesti particolari ci dice che le pseudorandom permutation sono indistinguibili dalle pseudorandom function e questi sono interscambiabili a questo punto.

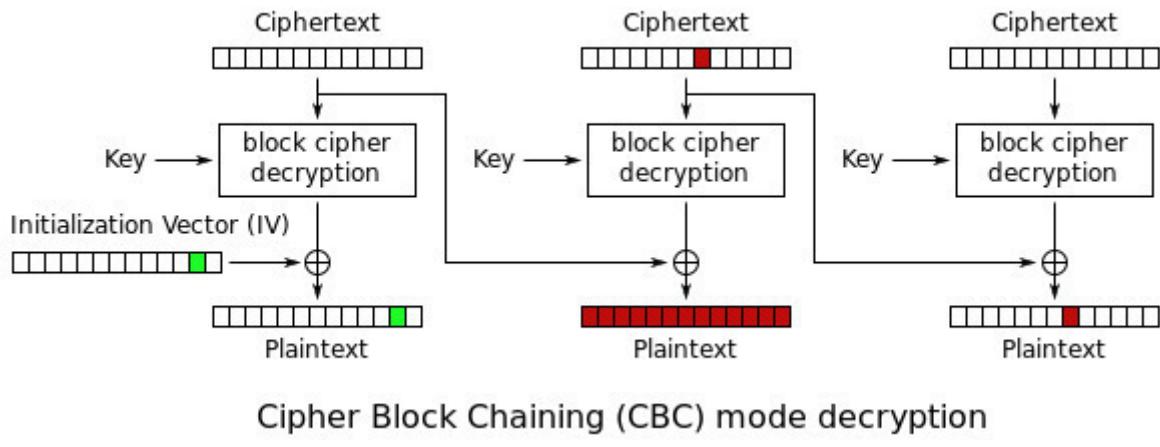
Altri block cipher sono: Ascon (nato a febbraio 2023, e lo standard non esiste ancora, ma potrebbe essere il futuro per contesti con bassa capacità computazionale), Simon and Speck (che si pensa che sia stato backdoorato). Contesti light cryptography: pensiamo a contesti in cui AES potrebbe incontrare delle difficoltà in termini di calcolo perché magari lavora davvero con pochissime risorse hardware e non so come fare.

Operation Mode

E' quello che ci interessa sicuramente di più perché sono algoritmi su cui possiamo ragionare. Questo può cambiare lo schema di cifratura effettivo finale! Potrei andare ad ottenere degli stream cipher o anche blocchi a blocchi o schemi che mi garantiscono anche l'integrità. Il block cipher è una primitiva che in base al logaritmo con cui lo combino ottengo cose molte diverse tra di loro. Questa conoscenza è quindi molto importante. Il primo schema che ci nomina è in realtà un non schema.

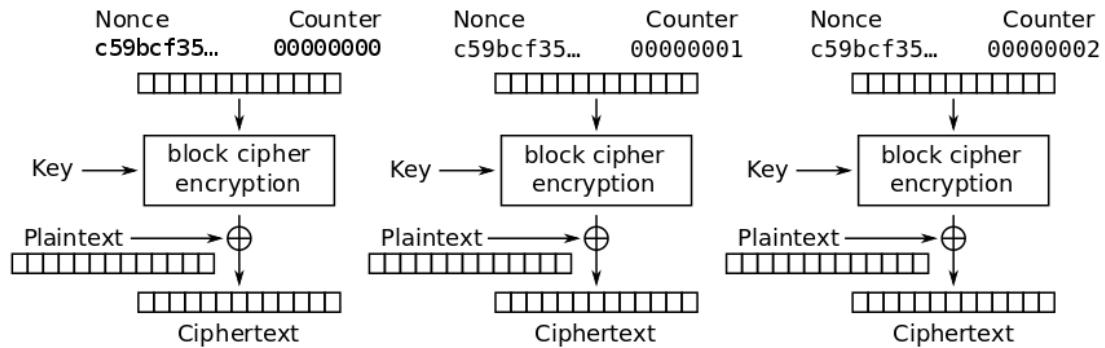
ECB

Modalità di cifratura che vede l'esecuzione iterata direttamente del block cipher. Questo è insicuro perché se immaginiamo di avere dei pattern ripetuti nel plain text ci ricordiamo che questo è comunque uno schema deterministico e quindi se abbiamo pattern ripetuti in modalità ECB questi sono individuabili con analisi di frequenza. Questa modalità esiste ma non dobbiamo mai usarla, l'unico caso in cui usarlo senza avere un effetto di problemi di sicurezza è quando sono in KEM. Caso storico di Zoom che usava questo modo direttamente.

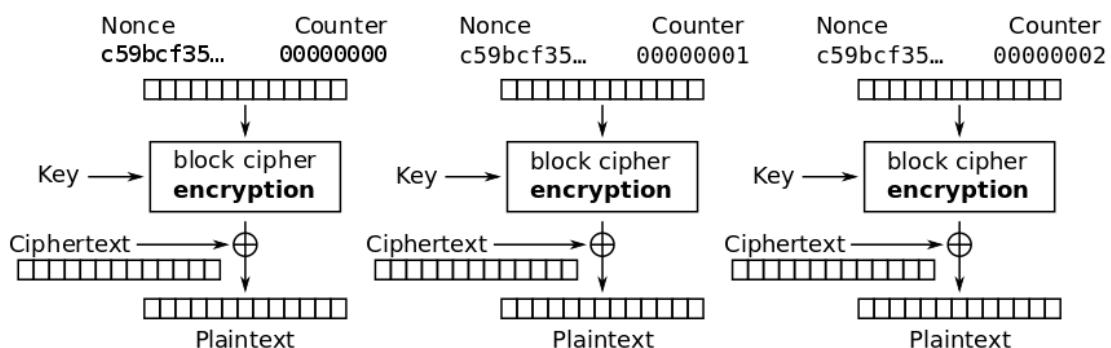


CTR

Ci permette di andare a costruire uno stream cipher attraverso un block cipher (slide 47). I dati output che genera sono la stream key degli stream cipher. Genero un flusso di dati pseudorandom. Abbiamo praticamente uno stream cipher che chiama *n volte* un block cipher. Esiste anche un *nonce* che possiamo considerarlo come offset iniziale per iniziare a contare. Questo mi serve molto perché spesso cifro tanti messaggi con la stessa chiave. Si sta facendo **context separation**. Esistono classi di attacco che si basano sul fatto che posso usare informazioni che vengono usate per un ruolo per un altro improprio. L'idea è che tengo le informazioni sempre ben separate. Se io ho ben separate la parte sinistra e destra, so che se ho un certo nonce, tutti questi valori sono effettivamente univoci. In nessun caso ho conflitti per cui l'input dello cipher diventa uguale ad un altro input con un nonce differente. Più è grosso il nonce più genero messaggi di dimensioni piccole. Nei protocolli reali, un protocollo deve sapere quanti dati massimi può cifrare con un singolo nonce o una singola chiave e riesco a fare una key rotation o reinizializzo lo schema con un altro nonce prima che sia troppo tardi. Devo sapere queste cose altrimenti è un casino, rischio collisioni insomma. Schema completamente malleabile



Counter (CTR) mode encryption



Counter (CTR) mode decryption

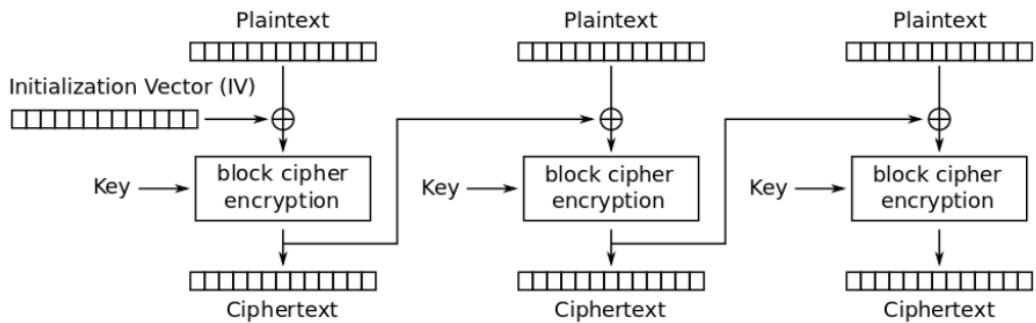
CBC

Altra modalità molto popolare che sta per cipher block chain, per cui la parte interessante è il chaining che praticamente fa una catena per quel che riguarda l'invocazione del block cipher. Qui praticamente il testo in chiaro è dato direttamente in input, come in ECB, ma qui abbiamo un *initialization vector* in input insieme e questo è simile al *nonce*. Questo IV viene xorato con il primo blocco del plain text. Perché così anche i dati di due plain text, con IV diversi, per dare due input differenti al block cipher. Dopo di che: ogni cipher text viene fornito in ogni blocco successivo come IV. Il primo IV lo diamo noi, il secondo lo diamo in automatico e ho un effetto a catena. In CTR ho una cifratura completamente indipendente, qui invece è fortemente sequenziale. Questo tipo di cifratura è storicamente molto utilizzata perché i dati son cfrati un blocco alla volta. Con CTR come definizioni abbiamo 128 bit, ... 256 bit alla volta. Faccio multipli della block size. Con CBC essendo davvero come uno stream cipher posso veramente avere una dimensione di bit arbitraria. Se ho un plaintext di 150 bit sono 128 + 22, quindi i primi 128 li faccio con stream key poi la seconda stronco la stream key e uso solamente i bit utili ai bit che voglio senza farmi problemi. **Il limite del block cipher accetta solamente dati uguali alla sua block size.** Questo introduce in teoria qualche garanzia di integrità.

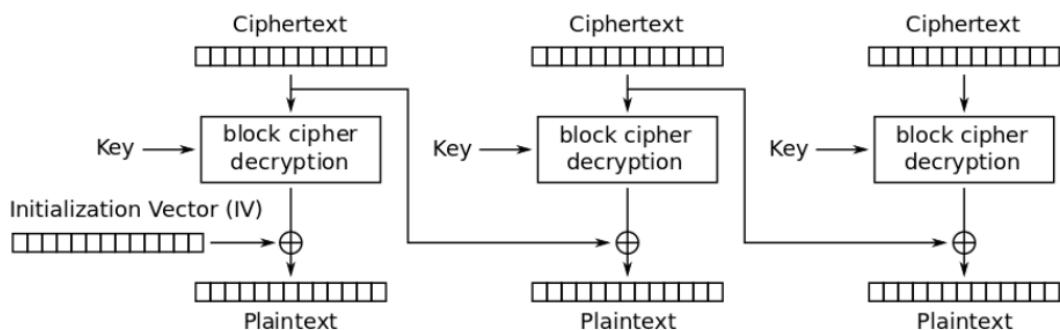
Immaginiamo un attacco di manipolazione dei dati. Con uno schema di cifratura a flusso, si sa che se un attaccante effettua una modifica al terzo bit, modifica anche il terzo bit in chiaro. Qui una modifica di un bit causerà una modifica completamente casuale nel testo chiaro del primo blocco. Questo perché è veramente indistinguibili da dati random, non posso sapere che cosa sto andando a cambiare, Eve non sa che danni sta facendo in termini di manipolazioni. **Questa era una credenza storica che negli attacchi reali non ha avuto effettivamente le garanzie di sicurezza aspettate.** Se faccio una modifica in CBC e poi il dato mi diventa IV del blocco dopo allora comunque l'attaccante sa esattamente cosa sta facendo, perché comunque creo caos e so che modifico esattamente il terzo bit del terzo blocco, perché lo metto in xor con il secondo blocco di cifratura. Questo è stato sfruttato per condurre degli attacchi realmente.

Tipicamente noi troveremo sempre cifrazioni a flusso e oggi si usa di più CTR.

C'è comunque un IV, che è simile al nonce, in quanto oggetto che dobbiamo andare a cambiare, ma che ha diversi requisiti di sicurezza. Ricordiamoci che il nonce ha un unico requisito: usare sempre valori diversi scelti davvero come ci pare. IV deve essere per forza un valore random, perché altrimenti se ho un contatore allora al momento posso creare uno schema vulnerabile. Random per altro con vincolo di unicità. Se lo replichi per altro fai bene danni. Se uso in CBC due IV fai meno danni che usare lo stesso nonce in schemi a flussi. Ti salvi con lo schema a valanga essenzialmente. Il motivo per cui esistono è che vogliamo la multi message security. Vogliamo cifrare tanti messaggi e chiavi indipendenti. Se non avessi questa componente che cambia vorrebbe dire che cifro tante volte lo stesso messaggio e ottengo lo stesso testo cifrato. I nonce o IV sono informazioni non confidenziali. Possiamo metterli in chiaro con il testo cifrato, la segretezza non è importante, ma non devono essere sotto controllo dell'attaccante. CTR in particolare verrà usato come schema interno di altre modalità. Alcune modalità sono alla base per costruire modalità complesse intorno.



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

Operations Framework

Interfacce che vengono usate per interagire con protocolli. Abbiamo un interfaccia solitamente fatta così:

1. keygen([size]) → key, la size è letteralmente la dimensione della chiave per la cifrazione
2. encrypt(key, iv, p) → c, se sono in CBC o altre modalità a blocchi devo avere che p sia di dimensione multiple della block size del cifrario usato. Potrebbe essere necessario andare ad implementare del *padding* per poter avere una dimensione che sia comoda al fine di andare a cifrario. Un padding famoso è **PKCS7**. Prendo in input un dato e aggiungo n volte il valore N che serve per raggiungere una determinata dimensione. Cosa succede se sono già apposto? Aggiungo un blocco intero di valore uguale a 16. Il caso peggiore è proprio questo. Perché abbiamo ciò? Perché la questione fondamentale è che chi fa unpadding deve essere sicuro di quale sia parte di padding e quale sia senza padding. Chi decifra lo deve riconoscere, soprattutto il destinatario non deve avere incertezze. **So sempre che c'è almeno un byte di padding in genere.** **L'ultimo byte di padding mi dice quanto spazio occupa tutto quanto il padding.** Esiste un caso molto interessante in cui quando ancora la crittografia

non era definita c'erano problemi di padding anche per comunicazioni di telefono senza fili. Questi avevano dimensioni fisse e poi faceva padding inventato dall'operatore che ci metteva solitamente una frase scollegata dal resto dell'ordine. In momenti di ambiguità il destinatario non era sempre in grado di riconoscere il tutto e a volte successero delle incomprensioni.

PKCS#5 / PKCS#7 Padding

- When pad with n bytes, all of them having value n

Original	B	a	c	o	n			
Padded	B	a	c	o	n	0x03	0x03	0x03
Original	E	g	g					
Padded	E	g	g	0x05	0x05	0x05	0x05	0x05
Original	S	a	u	s	a	g	e	
Padded	S	a	u	s	a	g	e	0x01

- When the message is divisible by the block size, you still need to pad a dummy block with n = block size

Original	S	p	a	m	S	p	a	m							
Padded	S	p	a	m	S	p	a	m	0x08						

- decrypt(key, iv, c) → p

Possiamo avere anche uno schema del genere:

- keygen([size]) → key
- encrypt(key, n, m) → c
- decrypt(key, n, c) → m



Mi raccomando i nonce SEMPRE diversi tra di loro, altrimenti ricavo materialmente il messaggio. Il nonce vuole univocità, IV vuole randomness. Non fidiamoci dei nomi delle vars di chi ci da la libreria, aprete sempre la documentazione perché molto spesso i nomi possono essere ingannevoli. Non dobbiamo fidarci devo vedere che sta offrendo la variabile e la funzione.

Il danno minimo è quello di creare uno schema deterministico, se uso lo stesso IV con lo stesso messaggio praticamente (qualsiasi schema che io utilizzo). Nel peggiore dei casi (stream cipher) posso creare uno o più danni. Se la stream key è la stessa posso avere i due testi in chiaro xorati. Con CBC ho delle mitigazioni. Vedi la storia della WWII Midway Island Battle. Ad oggi ho scemi deterministicici ancora in uso e quindi veramente quanto è grave tanto è critica la situazione. Potrei avere degli schemi apposta per cui se vado anche a replicare l'IV non ho uno schema deterministico. Dobbiamo vedere bene le situazioni in cui ci troviamo per cui per esempio anche se uso ancora il nonce non ho grandi beghe. Li vedremo più avanti.

[battle-midway.pdf](#)

Ci sono alcuni trucchetti: se ho uno schema di cifratura che mi prende un IV come input, ma nel mio sistema non ho modi di fare dati random fatti bene, posso usare uno schema CBC usando un nonce e cifrando il nonce prima dell'uso in modalità CBC. Do *nonce* in pasto al block cipher e quello che ottengo è il mio primo IV. Così cambiando sempre il nonce alla fine ho sempre roba random come output che posso usare come IV. Questa è veramente una best practices in cui potremmo tranquillamente trovarci.

A volte le librerie mi danno cose in cui non ho né IV né nonce. Questo capita per libreria in cui si assumono sviluppatori meno a testa cazzo. Nel dubbio meglio non mettere in mano agli sviluppatori a questa scelta.

Abbiamo delle best practices, per cui nascono l'input che potrebbe essere dato random. Questo è dato fw (?) probabilistico perché genero tanti testi cifrati differenti, ma questa è una astrazione, sotto ho lo stesso schema precedente. La libreria wrappa lo schema di basso livello in cui sono forniti anche nonce e IV. Questo ha un difetto queste funzioni sono stateless, con le funzioni una indipendente dall'altra. Effettivamente in tutti questi schemi anche se sotto lo schema è basato su nonce questo è generato in maniera random.

- controlla cosa volessi dire esattamente in questa sezione

Il difetto ad usare un approccio di questo tipo è ridurre il numero di messaggi che possiamo cifrare. Se usiamo un counter posso cifrare due alle 96 messaggi. Se generiamo un nonce in maniera random dobbiamo stare attento alle collisioni e che prob ho generando tanti nonce? Allora la cosa è che dobbiamo usare una soglia di tolleranza bassa. A livello di standard si è stabilito come standard del NIST ha definito una threshold come massima prob di collisione è di 2^{-32} invocazioni. Immaginiamo sempre dei nonce in maniera random e se la prob di creare un dublicato è più piccolo di 2^{-32} bit. Con nonce di 96 bit il numero di invocazioni è di 2^{32} . Questo numero cala tantissimo. Se i nonce più piccoli ho un numero non così alto come possiamo pensare.

[09/08/2023]

Parliamo di un contesto di comunicazione unidirezionale con la solita Alice che cerca di mandare un messaggio a bob. Parliamo di confidenzialità rispetto ad attacchi passivi. Cerchiamo di proteggere la privacy e la confidenzialità al fine di non dare informazioni di alcun tipo all'attaccante.

Integrità = capacità del destinatario di rilevare se ci sono state modifiche del tragitto

Autenticità = rilevare se il messaggio è stato mandato da qualcuno che non è autorizzato

Integrità

Parleremo di funzioni **hash** a tal proposito. L'unico scenario di una comunicazione in cui voglio garantire questo è uno scenario in cui il destinatario bob conosce già un digest di piccole dimensioni. Questo digest è un output di una funzione hash. Non abbiamo alcuna informazione che rappresenti il mittente alice, quindi anche Eve potrebbe mandare un hash. So semplicemente che il messaggio è arrivato integro,

da dove arriva BOH. Se voglio che Bob riceva dei messaggi da Alice devo andare a verificare questa informazione, altrimenti non ho controllo riguardo la persona con cui sto comunicando. Gli hash non hanno una chiave. Effettivamente le funzioni hash si inseriscono nella crittografia simmetrica perché internamente come costrutti sono più vicini alla crittografia simmetrica.

Autenticità

L'autenticità è quella che ci interessa all'interno delle comunicazione. Garantire l'autenticità vale dire garantire che una comunicazione è effettivamente autentica. Prendo in input chiave e messaggio e ottengo un tag. Bob riceve il messaggio + tag e ha una funzione di verifica a cui da chiave già nota, messaggio + tag e praticamente la funzione verify mi ritorna 0 o 1 in base al fatto che io abbia un messaggio autentico o meno.

Quale sarebbe il vincolo: l'assunzione che Alice conosca la mia stessa chiave e invece Eve no. Chiunque ha questa chiave è la persona autorizzata a mandarmi questo messaggio. Non ci interessa come Alice ha ottenuto la chiave, lo assumono, ma è effettivamente un problema.

In un mondo reale se voglio mettere l'identità del mittente: nome oppure host name, identità legale, metadati vari. Devo vedere come inserire le informazioni di identità. Posso avere un protocollo sopra per far vedere che effettivamente sia stato fatto tutto da Alice. L'entità è chiunque possieda la chiave. Integrità e autenticità sono strettamente legati, integrità è una condizione necessaria, ma questi due termini vengono spesso usati in maniera intercambiabile. Le funzioni di autenticazione le possiamo andare a definire come MAC = message authentication code.

Vediamo che quando parliamo di integrità una questione fondamentale è che tipo di integrità vogliamo andare a garantire. Pensa ai protocolli di comunicazione non sicure. Possiamo parlare di integrità anche senza andare a parlare di crittografia se consideriamo situazioni con problemi "naturali" o statistici. Integrità crittografica e NON crittografica cambiano in termini di assunzioni rispetto ai miei "avversari". Nel caso NON crittografico ci sta mandare un digest senza chiavi e basta.

Nonostante il codominio della funzione hash sia più importante del dominio (il dato > digest come dimensioni), so che già avrò più dati che faranno lo stesso digest, voglio garantire che siano funzioni crittografiche se non troviamo diverse input che trovano stessi output: $\text{data1} \neq \text{data2} \rightarrow \text{digest1} \neq \text{digest2}$

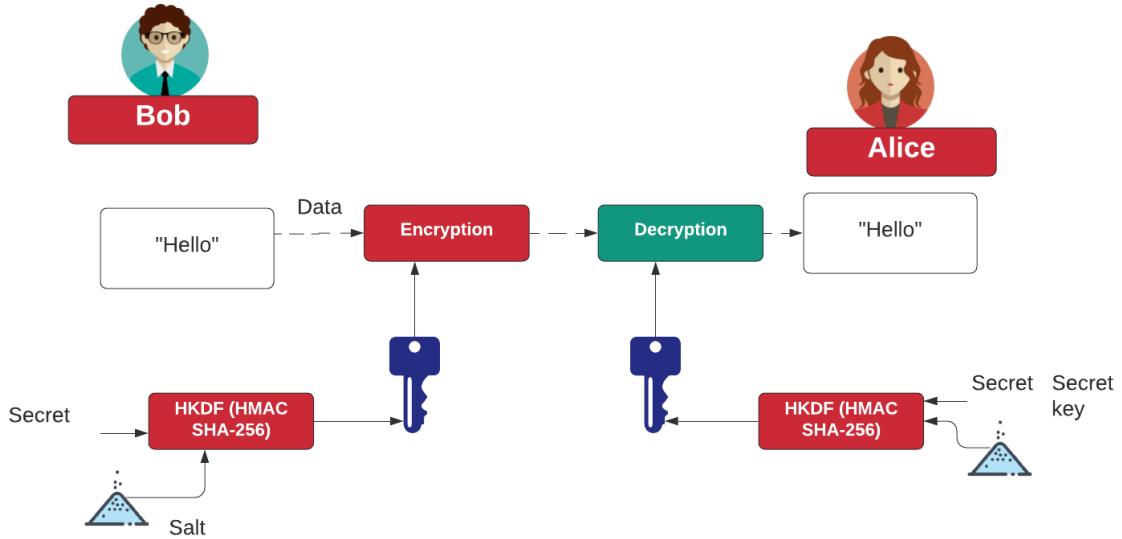
Se vediamo due digest uguali assumiamo che vengano dallo stesso dato. In caso contrario e l'assunzione fallisce si dice che all'interno della funzione hash ho una

collisione. Una funzione hash in particolare in cui delle collisioni è definita insicura e viene subito deprecata e abbandonata. Questa considerazione è vera per le funzioni hash crittografiche; in altri casi non è vera come assunzione e non è un problema avere collisioni. Le funzioni hash si comportano come funzioni pseudo random. Dato un input qualunque posso considerare il dato come output come pseudo random. Non posso prevedere che digest possa uscire fuori essenzialmente.

Le funzioni hash possono essere usate per garanzie di integrità, ma li posso vedere anche come blocchi per fare altri schemi. Qui abbiamo una considerazione che è interessante: una funzione hash la si può studiare a seconda dell'attack model e del contesto in cui si usa. Consideriamo 3 modelli di attacco:

1. **One Way:** unidirezionale per cui l'attaccante sappia un digest e conosca la funzione hash riesca ad invertire la funzione hash per trovare il dato generato del digest → forced pre-image resistance. Quando lo consideriamo? Le funzioni hash a volte le uso per delle funzioni al fine di derivare chiavi, calcolo key crittografiche a partire da altre. Il digest della chiave K è anche esso stesso una chiave K. Quando consideriamo che garanzie ci da questa funzione hash: pensiamo che l'attaccante violi K' (derivato da K) e si vada a calcolare K originale. Quando vince? Se calcola esattamente questo K. Se l'attaccante arriva a K riesce a decifrare tutto quanto. One wayness definisce che l'attaccante rompa completamente la funzione hash perché trova esattamente l'input originale. Nel mondo reale romperla ci dice che esiste una algoritmo efficiente per cui mi posso calcolare tutti gli input ammissibili per andarmi a calcolare un determinato output. E' l'attacco sicuramente più difficile e quelli successivi sono in un qualche modo più facili per l'attaccante. Vedremo come nel contesto reale uso hash per contesti di derivazione non le uso direttamente, ma standard HKDF che useremo in contesti specifici.
2. **Second pre-image collision resistance:** attaccante sa un messaggio ed il digest derivato e possa generare un messaggio M2 che è uguale a M1. Immaginiamo che Bob conosca già il digest e questo esista. Un avversario manda a Bob un M1 tale che bob si calcola il digest e pensa che il messaggio è autentico. Ricade nei contesti in cui è stato generato un digest per un messaggio corretto e l'attaccante vuole calcolare un messaggio M1 che faccia collisione per fingersi Alice. Qui voglio fare collisione su un digest scelto in precedenza.
3. **Collision resistant** = attaccante sceglie un messaggio random m1 ed m2 siano uguali. Presi in maniera random. L'attaccante fa effettivamente un pochino di tutto, sceglie m1 ed m2 e questi generano lo stesso digest. Questo è tutto a scelta dell'attaccante.

Esempio di funzionamento di HKDF



Potrei fare considerazioni riguardo il dimensionamento della funzione hash e i tipi di attacchi che posso andare fare come attaccante. Il valore N deve essere tanto più grande quanto è alto il livello di sicurezza. Dato quei 3 tipi di attacchi quanti operazioni in media ci mette l'avversario per vincere quegli attacchi sapendo che il digest sia grosso N.

Partiamo dall'ultimo caso di attacco: se l'avversario può scegliere una qualsiasi coppia di messaggi per avere una collisione, il modo sarebbe scegliere in modo casuale iterativamente i due valori → paradosso del compleanno. Questo ci definisce come la probabilità di trovare una collisione dato un digest di dimensioni N è tale che per cui riesco a trovare una collisione al 50% dopo $2^{(n/2)}$ operazioni. Se il digest è grosso 2^n , ho come sicurezza $2^{(n/2)}$ come sicurezza.

Se consideriamo un attacco di tipo second pre-image, la probabilità di vincere cala. Se prendiamo un attacco in cui il messaggio di partenza è già dato e poi abbiamo un elemento singolo di libertà: se la funzione è ideale io comincio ad andare in brute force Il numero medio di tentativi possiamo assumere che sia molto vicino alla dimensione dei dati stessi.

Potremmo trovare una funzione hash con 128 bit e leggere che ha come sicurezza 128 bit, viceversa funzioni hash a 256 bit che mi garantite 128 bit. Il dimensionamento della funzione hash dipende da che cosa voglio fare. Non modellare male il protocollo rispetto al mondo reale.

Slides 14 per andare una situazione reale

In questo caso il sito mirror potrebbe essere la superficie di attacco ed un tipo di attacco second pre-image. In questo caso vorrei 128 bit di sicurezza. Diminuiamo le assunzioni di sicurezza: potrei avere altri attaccanti diciamo che anche il content provider possano fare danni. Questi creano dati ad hoc per poi dire che il sito mirror ha fatto cattivo. Questo ha un attacco più forte perché sceglie dato e digest di riferimento. Successivamente succede che il content provider era l'avversario perché è lui che ha modificato in origine i dati, il sito mirror li redistribuisce senza sapere nulla e poi l'utente non se ne accorge. Anche lo stesso scenario modellato rispetto ad altre assunzioni di sicurezza evidenzia attacchi diversi. Vogliamo fare in modo che la funzione hash resista ad attacchi di collisione di tipo generico.

Esistono alcuni nomi notevoli da conoscere:

- md5: deprecata e usata NON in sicurezza, ma per altre sue proprietà
- sha1, sha2, sha3: sha1 relativamente recente ed era da considerarsi insicura, sha2 e sha3 li consideriamo sicuri. Questi ultimi esistono in diverse varianti a seconda del digest che ci interessa. Nei nomi SHA spesso si indica la dimensione del digest generato: sha224 oppure sha3-224

Tutte queste sono state funzioni hash crittografiche.

Storicamente come sono fatte dentro il design da md5 a sha3 abbiamo una struttura di tipo incrementale. sha3 non condivide nulla con le precedenti in senso di architettura interna. In certi contesti dovrebbe trovare alcune funzioni deprecate e teoricamente sicure, ma dipende dal contesto. Te considerale sempre deprecate.

Quando parliamo di second pre-image immaginiamo di avere un *m* ed un *digest* prefissati. Ci sono dei casi applicativi reali per cui la funzione è vulnerabile a collision resistance ma non a second agreement. Ci sono certe classi di messaggi per cui riesco a trovare delle collisioni. Rompere la second pre-image dipende dai casi essenzialmente. Esistono dei tool che possono dire se il messaggio che stiamo distribuendo è attaccabile da collisioni. Alla fine è tutta una questione di probabilità e se uso vecchie funzioni devo usare dei tools che evitano di cadere in casi in cui sono vulnerabili.

Una funzione HASH viene proposta come una funzione che prenda un solo input, ma ci potrebbe capitare di avere delle funzioni che hanno più dati. Ad AWS è capitato di fare degli schemi di codifica in cui ha creato delle tuple in cui non sono state considerate ambiguità. Che cosa succede: **metto insieme vari messaggi e scopro che coppie diverse di messaggi mi da in output lo stesso hash.**

Esistono delle funzioni di codifica per evitare questo genere di ambiguità. Devo far in

odo che una certa struttura dati abbia una codifica univoca dato in pasto ad una funzione hash.

$H('builtin' | 'security') = H('built' | 'insecurity') = H('builtinsecurity')$ → e hai fatto la frittata, bug a livello di codifica delle informazioni.

Message Authentication Code

Torniamo allo scenario di partenza di comunicazione: abbiamo visto che per garantire l'autenticità i due attori hanno la stessa chiave e parlano in maniera simmetrica e hanno una funzione che va a creare un *tag* ed una funzione che la verifica.

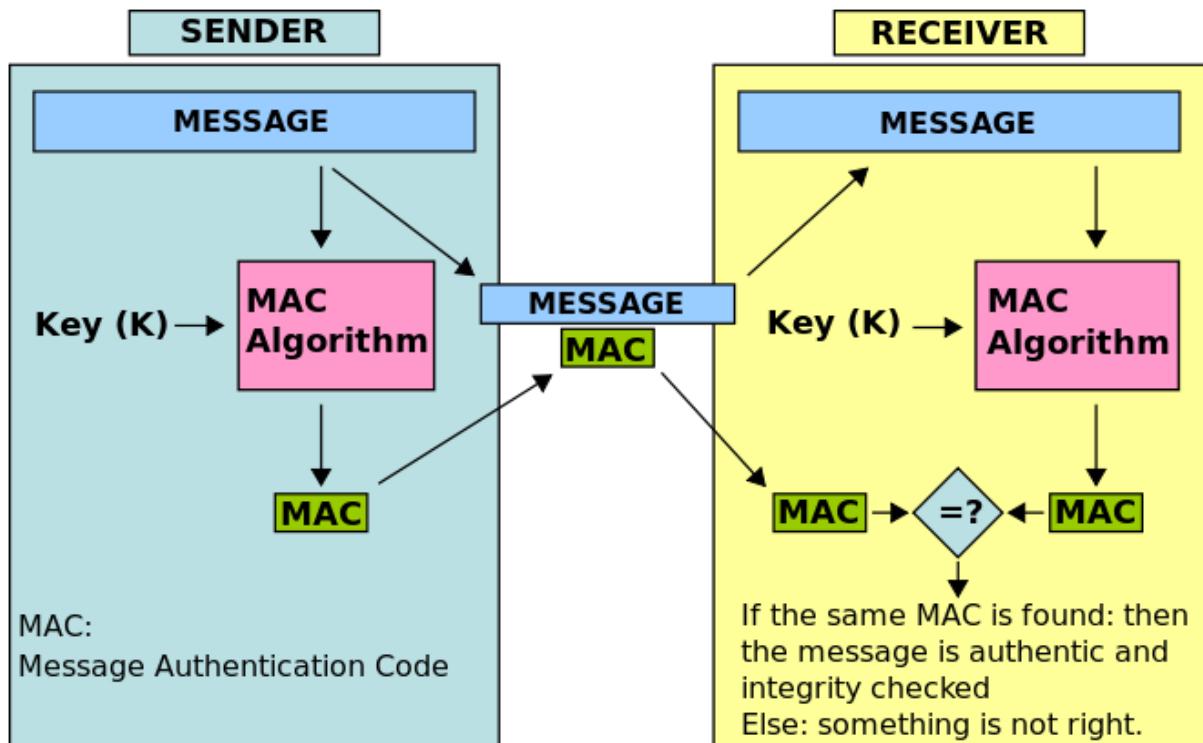
$\text{MAC}(\text{key}, \text{message}) = \text{tag}$

In che tipo di modello un attacco ci troviamo ora? Deve essere studiata dal punto di vista protocollore: consideriamo che Alice manda una serie di messaggi e per ognuno di questi associa un tag. L'avversario vince se, in qualsiasi momento nel tempo, riesce ad inviare a Bob un (m', t') che viene verificato con successo e questa coppia deve essere differente dalle altre appena osservate. Non è necessario che ci sia uno studio relativo alla collision resistance o altro ma immaginiamo che anche se qua il tag è molto piccolo per l'avversario, comunque non è facile indovinare il tag, perché qui ho un chiave segreta di mezzo. Per hash posso fare tentativi offline a non finire. Eve in questo caso deve aspettare che per altro Bob faccia un test e per altro Eve non ha accesso alla chiave né alla funzione di verifica. Qui l'avversario deve agire **ONLINE**, deve interagire con gli attori. L'unico brute force è farlo sulla chiave ma non sui tag. Qui potremmo dire: se il tag è **2^{64}** , ho **2^{32} operazioni** per andare a trovare una collisione ed un tag corretto, ma **queste sono interazioni con Bob**, quindi ti scoprono subito perché concretamente stai facendo milioni di richieste a Bob. subito. Sappiamo che la dimensione del tag riduce il numero di tentativi che Eve può fare, ma il fatto che siano online cambia tutto quanto. Bob può avere dei sistemi di sicurezza diversi per difendersi da questa mole di query.

La dimensione di questo tag quindi può variare e solitamente è dimensionato in maniera pessimista con un tag a 128 o 256 bit. In altri protocolli possiamo scegliere di troncare il tag e averlo piccolino come a 64 bit, ma siamo sicuri comunque perché Bob si tutela in altri modi. La chiave sempre scelta in maniera robusta.

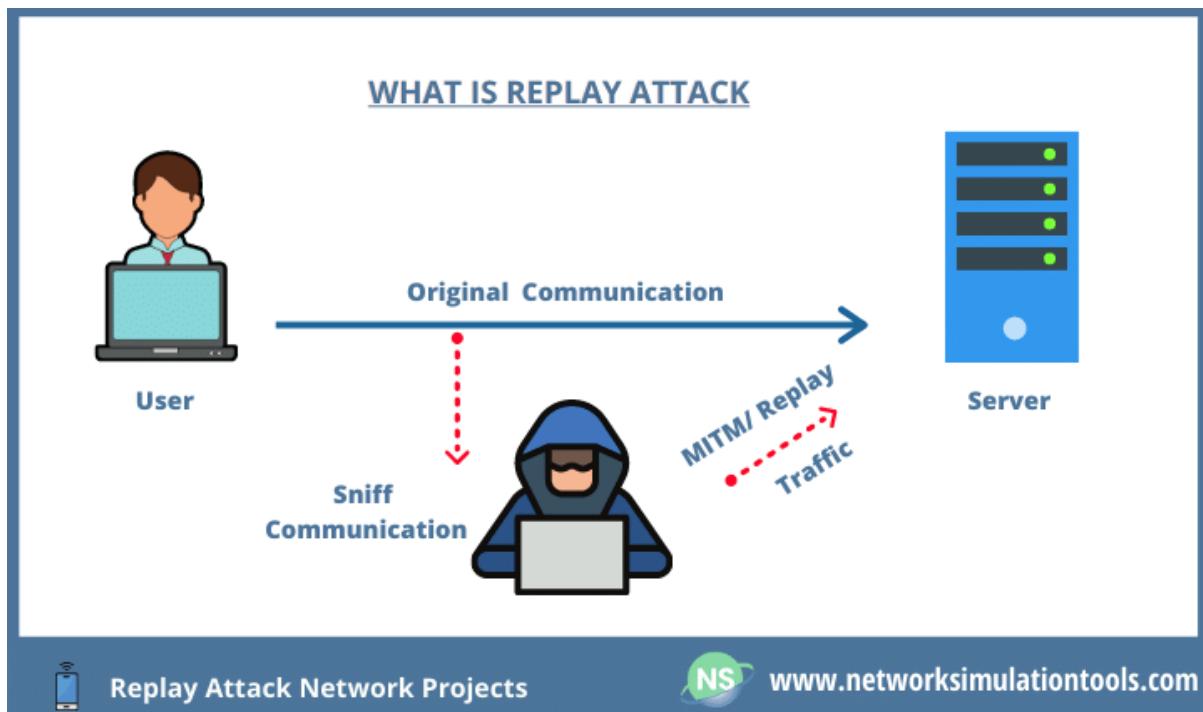
In base al tipo di MAC i trade off cambiano, soprattutto per quel che riguarda la questione del troncare il tag.

Tipi di attacchi in messages authentication codes



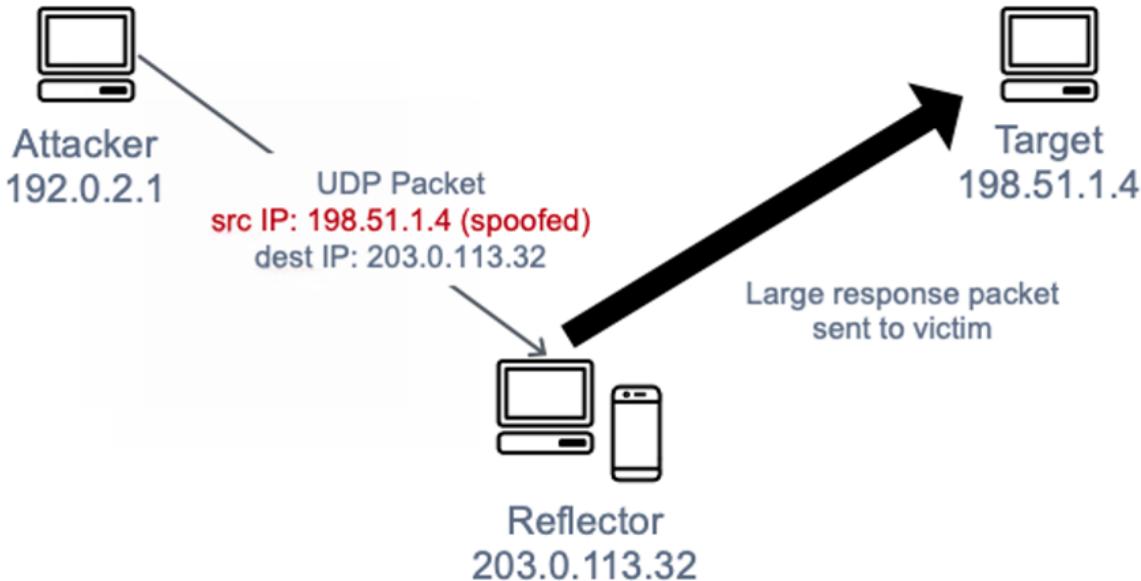
Replay Attack

Compio un attacco senza violare la funzione Mac perché bob non nota nulla di strano. Gli sto rimandando messaggi già mandati. Per proteggermi devo introdurre dei metadati al fine che ogni messaggio sia univoco. Inserisco un campo identificatore che identifica il messaggio all'interno di un intero flusso di scambio dati. La funzione MAC la metto sia su ID che sul messaggio. Il messaggio a livello di protocollo è ID + Messaggio. Molto spesso quando codifico questo genere di informazioni predispongono una dimensione prefissata perché così so che il contatore mi occupa solamente N-bytes. Cerco di non avere alcun tipo di ambiguità. Faccio in modo di non avere ambiguità. Quando finisco lo spazio degli ID.



Reflections

Eve può mandare ad Alice un messaggio con un id che Bob non ha mai usato ancora. Per ovviare posso inserire un bit per la direzionalità del messaggio. Nell'ambito delle comunicazioni ha delle chiavi differenti: chiave per autenticare e una decifrare. Così non ho conflitti nell'uso delle chiavi. Questo è l'approccio comuni tipicamente. Da una singola chiave potrei andare a generare tante chiavi per evitarmi alcuni tipi di attacchi. Il bit di direzionalità potrebbe essere utili in certe situazioni asincrone, perché è un metadata che identifica esplicitamente il mittente della comunicazione da un punto di vista informativo.



In passato si facevano le MAC come hash: $H(\text{key} \parallel \text{message})$, violare la MAC a questo punto diventa un second pre-image collision. Questo in teoria è vero se vediamo le hash come oggetti ideali ma se le vediamo fino a sha2, la struttura interna si basa su una costruzione interna definita come Merkle-Damgard. Tutte queste sono note per essere vulnerabili all'attacco: Length Extension Attacks: $H(\text{key} \parallel \text{message}) \rightarrow \text{tag}$, $\text{LEA}(\text{tag}, \text{message}') \rightarrow \text{tag}'$ che praticamente sarebbe come dire $H(\text{key} \parallel \text{message} \parallel \text{message}') \rightarrow \text{tag}'$

Questo tag è sempre valido, perché è tutto incrementale come funzione, perché ogni tag n è dipendente dal tag $n-1$. Io conoscendo lo stato tag posso evolverlo senza sapere la chiave.

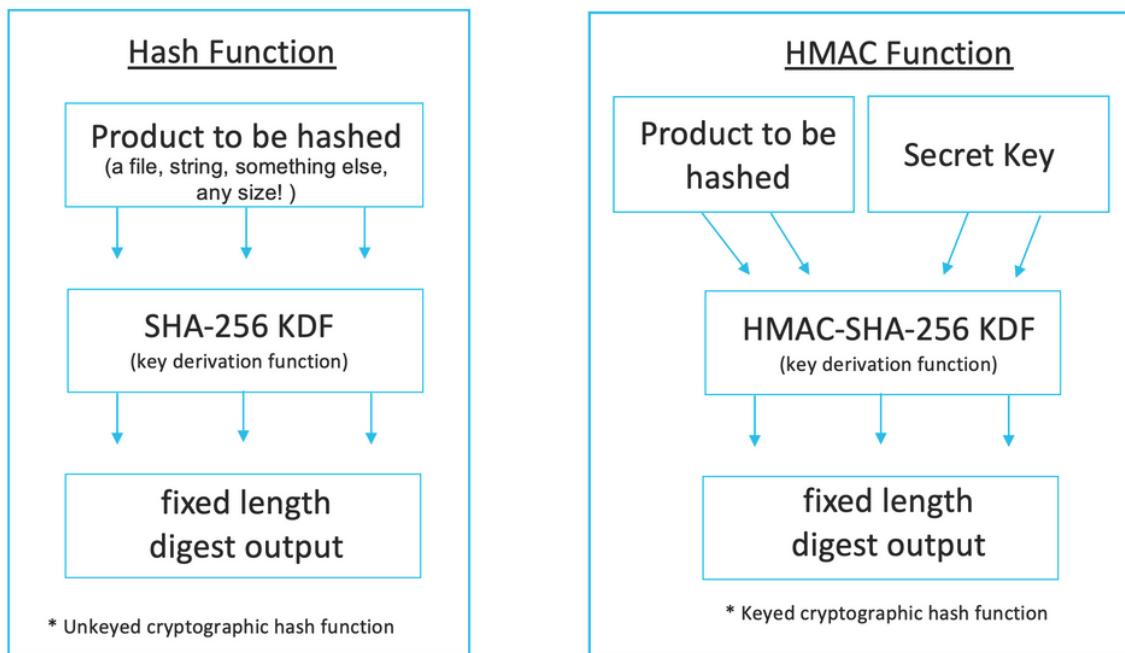
Attack ($H(\text{key} \parallel \text{message}), \text{message}' \rightarrow \text{tag}'$)

Questo è il motivo per cui questa costruzione non è valida, perché faccio dei tag fasulli senza conoscere la chiave originaria e sto violando la funzione MAC. Lo standard più noto è chiamato: HMAC. Questa costruzione concettualmente è che HMAC prevede una esecuzione innestata di due funzioni hash: $\text{HMAC}(\text{K}, \text{M}) = H((\text{Kp XOR opad}) \parallel H(\text{Kp XOR ipad}) \parallel \text{m}))$

Interessante perché ha dimostrato formalmente la funzione è sicura fino a quando la funzione H è sicura. In questo caso non posso usare LEA e giocare con la lunghezza. La parte sottolineata genera sempre un hash di dimensione fissa che non posso andare a cambiare. In questo caso rompiamo le strutture interne delle funzioni hash per andare a sfruttare questo tipo di attacco.

Questo funzioni HMAC le trovo con le funzioni SHA2 perché sono vulnerabili, queste ultime, a LEA. Le SHA3 invece sono internamente non vulnerabili a questa classe di attacchi.

Esistono una serie di altre funzioni mac popolari: HMAC, CMAC (successore di CBC MAC), GHASH/GMAC basato su funzioni polinomiali che è praticamente calcolo di un polinomio, sarebbe un calcolo matematico che troviamo poi.



[15-03-2023]

HMAC spesso la troviamo associata a sha2. Non conviene fare una funzione mac con una chiave come prima parte del messaggio perché ci esponiamo all'attacco lenght extensione. Con la funzione sha3, progettata con in mente la vulnerabilità del tipo citato prima, potrei usare un hmac, ma in realtà vado ad usare una funzione definita come kmac. Hmac richiede due funzioni hash, fino a sha2 era necessario a prescindere, con sha3 diventa un costo in più e quindi niente si smette di fare questo tipo di utilizzo. Mentre lo standard sha3 già ci da una variante da andare a usare tranquillamente. CBC-MAC è stato deprecato perché anche questo è vulnerabile al lenght extension per sha1, sha2. *Tornaci per vedere un caso particolare di malleabilità di CBC.* CBC è considerato come deprecato ed è assolutamente sconsigliato. Per fare un Mac dallo cipher è CMAC. Praticamente composto da una parte iniziale simile a prima. Molto spesso quando abbiamo una funzione più veloce, sicuramente consuma anche meno in termini di energia. L'ultima sono le funzioni

note con il nome di GMAC → funzioni MAC che si basano interamente su funzioni lineari, su formule matematiche. La funzione mac si possa costruire tramite una funzione che praticamente è un polinomio. Una funzione MAC di un messaggio rispetto ad una chiave è generata come il polinomio. Il MAC non è altro che un messaggio strutturato *come in slide 46*. Questo viene chiamato hash universale. Dove il MAC è un polinomio di grado L dove L è la grandezza del messaggio. R è un nonce al fine di non poter fattorizzare il tutto. GMAC è comodo perché è molto efficiente in architetture general purpose, ma la struttura matematica lo rende molto soggetto a ripetizione del valore N (nonce), quindi possiamo andare a riottenere il polinomio in cui sappiamo che i coefficienti sono dati dalla differenza dei messaggi e otteniamo fattorizzando la stessa chiave.

Popular attack models

1. EAV (eavesdropper) → totalmente passivo e non sa nulla
2. KPA (known plaintext attack) → sa qualche plain text e quai testi cifrati sono ad esso collegati.
3. CAP (chosen attack plaintext) → l'attaccante scegli i dati che noi andiamo a cifrare per farci studiare. Praticamente lui potrebbe accedere all'oracolo di cifratura. Ha accesso all'API e vedere che dati vengono cifrati. Ancora passivo nei confronti dei dati in transito, che non vengono toccati, ma solamente letti.
4. CCA1 (Non Adaptive Ciphertext attack) spostiamo l'attenzione sul plaintext a ciphertext. CCA1 può anche andare ad accedere alla funzione di decifratura, andando a includere tutti quanti gli attacchi attivi sopra i cifrati.
5. Adattività Chose-Ciphertext Attacks (CCA2); cambiamento di atteggiamento in vista di ulteriore evoluzioni., Access to Decryption può accedere all'oracolo di cifratura e all'oracolo della decifrazione.

Un attacco di un determinato livello può assolutamente includere le fragilità superiori. Vogliamo sempre garantire la confidenzialità e non distinguere i dati cifrati e dati random. A volte questa modellazione no è sufficiente e quando dovremmo scegliere il metodo di attacco migliore.

Nomenclatura IND-<attack-model> = voglio fare in modo che l'attaccante non vada a distinguere dati random da dati cifrati.

IND-CPA

slide 13

Attaccante CPA può sottomettere al challenger un sacco di copie di messaggi. L'avversario ogni volta sceglie due messaggi e questo deve decidere un bit 0-1 e se questo bit è zero. Questa modellazione sembra strana, ma EVE vince se capisce che cosa ha cifrato il challenger con proprietà maggiore al 50%. Cioè scopri con più del 50% che cosa è cifrato e che cosa no, riesci a fare una distinzione tra dati random e dati interessanti che magari sono stati cifrati, altrimenti se sei sempre al 50% è sicuro. Uno schema deterministico potrebbe anche essere sicuro per attacchi KPA particolari perché se Alice cifra sempre messaggi diversi fra di loro e diversi da Eve, l'avversario non riuscirà mai a sfruttare che lo schema è deterministico.

Discussione su schemi deterministici perché in alcuni contesti è utile avere anche schemi deterministici ed esistono dei framework che ci dicono esattamente questo. *Vedi framework dead.* Ci sono degli schemi che ci dicono che siano deterministici, ma non sono di default. La modellazione in CPA potrebbe subire delle varianti. Esiste una modellazione di attacco DCPA, questa modellazione implica che l'avversario può scegliere i valori che vuole, ma questi devono essere sempre diversi. Questo attaccante fa tutto quello che devo fare. Studio dei giochi di attacco per andare a catturare la massima sicurezza possibile per un determinato tipo di schema. Per cui modifco un po' l'attacco per cui poi solamente quella cosa è consentita e nient'altro. A parte queste considerazioni, abbiamo parlato di CBC in cui devo scegliere in maniera random. Questa ha un requisito fondamentale → non predicitività. Assumiamo in CBC che non sappia fare una preziione sul valore dell'IV. E' qui assumiamo che questa assunzione non valga. Cosa succede se in CPA è possibile fare una predizione di IV?

Vedi attacco CPA fatto da americani verso i giapponesi durante la seconda guerra mondiale e di come furono soggetti all'attacco deterministico. Praticamente fecero un attacco CPA indotto. L'esperienza ha mostrato che nella crittografia bisogna difendersi almeno da attacchi CPA, perché l'avversario potrebbe trovare escamtoege per **farcirci cifrare quello che vuole lui**. Nelle comunicazioni è ovviamente molto facile! Questo vuol dire che l'avversario può infine crearsi una mappa di riferimento tra i messaggi in chiaro e quelli che noi comunichiamo cifrati ai nostri alleati che hanno per oggetto i testi in chiaro decisi dal nemico.

In CCA1, l'avversario non può leggere come reagisce il destinatario. Dobbiamo avere schemi che ci garantiscano resistenza nei confronti della malleabilità. La prima analisi è lo schema CBC. Pensiamo all'ambito della malleabilità. Visto che siamo in CCA, la questione interessante è che in CBC, nonostante sia meno malleabile di uno stream cipher, abbiamo degli xor che possiamo andare ad attaccare soprattutto

sull'IV. Se flippo un bit dell'IV sono sicuro che il destinatario ottiene un testo in chiaro in cui il primo bit è flippato.

In un contenuto informativo complesso la prima parte è molto importante e avere questo potere di attacco è devastante. Esempio: immaginiamo di avere un file di configurazione e di volerlo farlo diventare un commento! Se conosco il messaggio originale posso avere una manipolazione con effetti catastrofici, vedi file di configurazione ssh in cui commento la password di autenticazione. Posso anche cambiare tipo di formato del file e versione. Attenzione che anche manipolando i testi cifrati abbiamo un effetto diretto di manipolazione sul testo successivo. In alcuni casi potrei anche fregarmene. Tutti questi attacchi richiedono che ci sia una conoscenza del plaintext originale.

IND-CCA1 e IND-CCA2 verrà trattata la prossima volta. L'attacco concreto di tipo CCA1 è stato scoperto nel 2018 come EFAIL: <https://efail.de/>



Attacco a OpenPGP e S/MIME. Questi consentono di cifrare gli attacchi in transito. Abbiamo un mittente che cifra i dati e un destinatario che li decifra. Quello che faceva PGP è consentire l'utilizzo di schemi di cifratura che non ci proteggano rispetto ad attacchi attivi. In questo tipo di scelta veniva giustificata dal fatto che questa comunicazione è asincrona. Il mittente invia un messaggio, il destinatario lo riceve e finisce lì. Questo attacco si basa sul fatto che i messaggi che mandiamo non sono solamente testo e i clienti non sono stupidi ma possono andare ad interpretare codice html per andare ad interpretare immagini. Allora per questa vulnerabilità che cosa fa: effettua un attacco attivo, manipolando la prima parte del messaggio decifrato e trasformando il contenuto della mail in una immagine. Questo attacco vuole far credere al client di posta di aver ricevuto una tale immagine. Pensiamo che nella prima parte della mail ci fosse un mime, relativo al contenuto della mail stessa.

Attenzione perché posso modificare il messaggio in transito creando un messaggio Frankenstein su cui ho in qualche modo un controllo: il primo so esattamente qualche contenuto avrà ed anche il secondo. Io modifichiamo IV e poi metto dei blocchi farlocchi e come secondo blocco di metto un blocco generato da me ad arte. Lavoro sulla malleabilità per ripetere blocchi cifrati che so che generano dati in chiaro di precedente conoscenza.

Tutto questo per fare esfiltrazione: invia una mail in chiaro ad un dominio che decido io!!! Questo caso tipico in cui si assumeva che in un contesto di comunicazione asincrona dare qualcosa di malleabilità non fosse così male, ma si rischia di rilevare una mail cifrata ad un avversario qualsiasi. Impariamo due cose: usiamo il massimo livello di sicurezza (nelle mail stiamo su CCA1,2).

IND-CCA2



Relationships among Security Definitions (2)

Target Attack	One-way (OW)	Semantically secure (IND)	Non-malleable (NM)	
Passive attack (CPA)	OW-CPA	IND-CPA	NM-CPA	
Active attack (Chosen- ciphertext attack) (CCA)	CCA1	OW-CCA1	IND-CCA1	NM-CCA1
	CCA2	OW-CCA2	IND-CCA2	NM-CCA2

6



Avversari attivi adattativi. Nelle modellazioni standard è l'avversario più forte sicuramente. In questo schema vediamo come Eve potrebbe mandare testi in chiaro a suo piacimento ad Alice. Dopodiché potrebbe anche leggere dati in transito, ma anche manipolarli. Può veramente fare quello che gli va e osserva anche le risposte. Questo è assolutamente naturale in uno scenario tipico. Con le comunicazione asincrone invece non è così, potrei avere un destinatario che sia completamente zitto. Si cerca di progettare sistemi che riescano a resistere ad attacchi CCA2.

Padding Oracle Attack

https://en.wikipedia.org/wiki/Padding_oracle_attack

Si sfrutta che ci sia una certa struttura di padding. Quando l'avversario manipola dei dati, ha un primo obiettivo: riuscire a fare accettare i dati modificati al destinatario come se fossero legittimi, non ci interessa che abbiano senso i dati. Non è detto che modificando dati a caso, si arrivino ad avere dati legittimi. Ogni volta che abbiamo struttura nel dato in chiaro, dobbiamo stare attenti per mantenere il dato legittimo altrimenti rischiamo di non farci accettare nulla!

Potrei pensare di andare a modificare in posizione coda, cioè le posizioni finali dello schema ed essere sicuro di avere un padding finale che alla fine sia corretto? Come faccio? Assumiamo di non conoscere il plain text, questa informazione ci sarebbe utile se lo conoscessimo? Se conoscessi il blocco finale potrei manipolare il ciphertext precedente come se fosse una sorta di IV. Il punto fondamentale è che in un contesto reale un CCA1 non può avvenire con successo, perché non ho certezza dell'esito. Succede però che se ho un CCA2 in cui osservo la risposta posso usare un approccio con diversi tentativi. Questo è un attacco peculiare, didattico quasi. Attaccare il padding di KCS7, ha un costo polinomiale. Possiamo usare un algoritmo efficace in pochi tentativi. Questo ha una struttura per cui vogliamo che ci siano n bytes di valore n . Dobbiamo pensare: come facciamo? Immaginiamo di convincere Bob ad accattare un dato di dimensione 6, quanti tentativi ci potrebbero volere per indovinare questo padding di 6 byte? Potrei fare un approccio dicotomico, rendere tutto logaritmico. Se voglio indovinare un dato con un padding di 6 byte, dovrei provare tutte le configurazioni possibili. L'approccio straw man (più facile possibile). Normalmente se non sapessimo come è fatto il padding, ma sappiamo che ci sia, ci metteremmo in media $2^{(n*8 - 1)}$ tentativi (-1 perché è la metà). In questo caso conoscere la struttura del padding mi permette di fare qualcosa di simile ma dobbiamo andare a specializzarlo. Sapendo che il padding è fatto in questo modo posso andare a puntare a fare padding incrementale praticamente. Visto che il padding lavora che 1 byte alla volta, io lo attacco un byte alla volta. Anche se voglio indovinare un padding di 6 byte, la prima cosa è beccare un padding di 1 byte. Questo ci impiega $2^{(8-1)}$ operazioni. Dopo che ho indovinato, se so che impostando. L'attacco anziché che richiedere $2^{(n*8-1)}$ diventa $2^{(8-1)*N}$, decisamente molto meglio! Passo da esponenziale a polinomiale.

Se ci vogliamo difendere usiamo una classe di cifratura autenticata sicuramente. Usiamo cifrature che non soltanto nascondono i dati ma si proteggono da attacchi di integrità, quindi → usiamo funzioni HASH e schemi MAC. Negli schemi di cifratura autenticata, usiamo entrambi alla fine. La questione fondamentale IND-CCA2, vogliamo fare in modo che la cifratura non sia malleabile. Questi schemi sicuri sono

schemi in cui la funzione di decifratura è in grado di rilevare esplicitamente se il testo è stato autenticato. Immaginiamo che la funzione decrypt che accetta una chiave, nonce e testo cifrato, mi da il plain text. Questo è il modello non autenticato. Una cifratura autenticata, accetta gli stessi tipi di input, MA l'output della funzione di decifratura potrebbe essere **fallimento**. Immaginiamo che questo tipo di schema embedda un MAC che implementa una funzione verifica. La funzione di cifratura autenticata embedda questa funzionalità. Se uno schema è sicuro in CCA2 c'è un esito sicuro sulla verifica dell'autenticità del messaggio. La prima cosa da capire è che una cifratura autenticata deve effettuare cipher text expansion → aggiungiamo dati di controllo per andare a fare il controllo di autenticità. Il testo cifrato è quindi sempre più grande del testo in chiaro originale. Non possiamo garantire integrità forte senza dati di controllo, tanti più sono meglio è. Anche se in bb (=black box), immaginiamo che il cipher text dello schema autenticato è composto da due elementi:

1. testo cifrato originale
2. tag

Questo è quello che accade sotto al cofano! quando lo usiamo come software questa struttura viene nascosta per comodità d'uso. Il più delle volte la prima è testo cifrato e l'altra è il tag. Considerazioni interessanti, come rilevo l'errore e la decifratura è fallita? Dipende dal linguaggio e implementazione della libreria. Spesso c'è una funzione più che dobbiamo andare ad eseguire noi, una verifica in più che è lasciata a noi, poi ripetiamo: i codici di errori si differenziano da linguaggio a linguaggio, vediti la documentazione sicuramente. La questione interessante è che possiamo seguire due strade

- usiamo certi standard che già sono autenticati: AES-GCM (GCM è una modalità di cifratura autenticata) praticamente sarebbe AES-CTR encryption scheme con GMAC authenticator
- Chacha20Poly1305 che è Chacha20 encryption scheme + Poly1305 authenticator

GMac (approfondimento sul suo funzionamento)

Questi due sono i due standard di cifratura autenticata sicuramente più popolare. Il primo se c'è accelerazione hardware è molto comodo, altrimenti se non ho accelerazione hardware vado con chacha20.

Se non ho già lo standard AES-GCM come faccio? Queste sono alcune regole d'oro:

1. schema in cui uso schema + mac in maniera indipendente: encrypt and mac

2. struttura in cui decido di usare mac per fare tag e lo cifro con il plaintext: mac then encrypt

3. cifro i dati e poi effettuo i dati del testo cifrato: encrypt then mac

In teoria una generic composition mostra come senza conoscere i dettagli di encrypt then mac in black box, se non sappiamo che cosa c'è dentro è quello in cui uso encrypt then mac. Questo per general purpose, posso avere schemi come gli altri per cui però sarà valida, ma di base uso quella.

Ci sono anche altri tipi di composizioni non in bb, ma prendono in considerazione lo schema GMC con qualcos'altro. Per modellare le figure di slide 4 uso un framework probabilistico, senza vedere l'IV. Dal punto di vista di autenticità è da considerare come il plain text. Quando dico di fare il mac del plain text faccio anche il mac dell'IV. Garantirne l'autenticità è importante, perché se lo so posso fare tutta una serie di attacchi.

Lo schema di cifratura e mac devono usare anche blocchi indipendenti. A volte non è un problema, ma a volte si. Se uso una combinazione (*sinceramente sto ancora cercando di intuire che cosa abbia scritto*) del genere con cbc encryption o mac potrei avere problemi. Se uso una chiave segreta non usarla mai per più scopi. Fai che ogni singolo blocco sia l'unico a usare quella singola chiave di cifratura, non si sa mai che ci siano correlazioni di cui non me ne accorgo e faccio casini. Un'altra considerazione è che la decifratura deve seguire un approccio sequenziale in cui prima verifico l'autenticità e solo dopo faccio la decifratura dei dati. Prima il testo da decifrare deve passare attraverso la verifica del tag. La parte di verifica la devi fare in maniera time costant. Questo serve a non dare informazioni sul fatto che un tag è valido in certe parti ma non è valido in altre. effettuare un confronto tempo costante serve a non trasformare un attacco che richiede 2^{128} tentativi, in $8 \cdot 2^7$! Se parliamo di creazione che cosa, pensiamo anche a un tempo costante rispetto a cosa? Questi algoritmi devono esserlo sempre anche rispetto alle informazioni segrete, ma quando ne parlo in questo caso è rispetto al tempo costante per il messaggio che viene mandato. Questo tipo di schema tra le vulnerabilità ha delle prob legate al tempo di calcolo del tempo di MAC, quando vado a calcolare il tag.

AEAD

Tipico framework per crittografia autenticata:

keygen([size]) → key

encrypt(key, n, a, m) → c

$\text{decrypt}(\text{key}, \text{n}, \text{a}, \text{c}) \rightarrow \text{m}$

L'idea sarebbe che l'header diventa un dato importante per lo schema perché praticamente succede che non posso andare a cifrare l'header perché mi serve all'interno di altri protocolli, ma lo posso sfruttare per capire se ci sono state modifiche o altro. devo fare in modo che ci sia un binding crittografico tra payload cifrato e metadato. Quello è un altro caso iper popolare.

Schemi CCA1 li vedremo come legati a shcemi di disk encryption. Nasce dalla necessità di rispondere a due garanzie fondamentali: dimensione testo in chiaro, dimensione testo cifrato. Quando vogliamo applicare una cifratura di basso livello a livello di disco, non vogliamo andare a sprecare spazio. Non vogliamo mettere informazioni aggiuntive di controllo. Voglio una corrispondenza uno ad uno tra disco fisico e disco cifrato. Non voglio aggiungere né tag né IV, non voglio autenticazione, ma voglio qualche garanzia di integrità e poi voglio schemi velocissimi.

Non voglio avere limiti di performance o limiti di dipendenza! Voglio anche avere un pochino di randomizzazione. Storicamente in vecchi standard su cifratura del disco lo schema preferito era CBC, questo ha attacchi di malleabilità, con IV predibili, potrebbe richiedere padding. Esistono comunque ancora degli standard in giro con vulnerabilità note, soprattutto in contesti legacy. Ad oggi se lo cifriamo sappiamo che abbiamo cose fatte meglio. Un tempo si usa quello perché era l'unica alternativa praticamente, dopo avevamo lo stream cipher ma non era granché come cosa.

Se voglio una sicurezza maggiore non posso usare disk encryption, ma object encryption a livello un pochino più alto.

Lo stato dell'arte moderno sarebbe XTS: XEX-based Tweaked-codebook mode with ciphertext stealing. Sarebbe una roba del tipo: XEX \rightarrow XOR - Encrypt - XOR

Sarebbe un metodo per progettare un tipo di cifrario tweaked, un tipo particolare di cifrario tweaked. Lo stealing sarebbe una parte opzionale. Il ciphertext stealing la uso al posto del padding. Strategia nota per usare cifrature a blocchi senza aumentare la dimensione del testo cifrato rispetto al testo in chiaro. Cipher text stealing è una tecnica alternativa al padding tradizionale per supportare dati di dimensione arbitraria con schemi di cifratura a blocchi. L'altra considerazione è che questo tipo di cifratura viene chiamata narrow block encryption. Questo sarebbe lo standard de facto per l'industria. Fatto prima da IEEE il MIST se l'è preso e lo ha fatto suo praticamente. Variante: sempre schema CCA1 ma molto più nuovo.

Adiantum

Progettato da google pensato appositamente per android e dispositivi mobile in gerale. Perfetto per ARM con basse capacità computazionali e poche estensioni hardware. Funziona bene senza accelerazione hardware AES. Per essere molto ottimizzato usa chacha12, usa poly1305, usa NH hash e poi AES una volta per blocco. E' un pochino un casino. Si vuole essere super veloce per specifiche architetture di fascia bassa. Di default per android 10 in poi. Su dispositivi per fascia alta AES XTS è preferito. Disponibile anche su kernel linux standard. Questo è uno standard de facto, fatto da Google, non è uno standard NIST. Ha lo stesso aim di xts.

HCTR2

Questo è un ulteriore modalità che è stata embeddata in linux da maggio 2022, ed è praticamente un adiantum ma è targettizzata per sistemi con accelerazione AES hardware. Tutti sono sicuri da attacchi CCA1. Un attaccante non ha idea di che modifiche sta propagando sul testo in chiaro. Non ho un esito forte rispetto alla cifratura fallita. Non ho una rilevazione dell'attacco, ma assumo che non sappia che danni stia facendo.

Abbiamo detto che questi schemi lavorano a blocchi indipendenti: narrow block → schemi di cifratura CCA1 in cui le modifiche sul blocco cifrato si ripercuotono solamente sullo stesso blocco del testo in chiaro. Ci vincoliamo a lavorare su una dimensione come il block cipher. Se l'attaccante manipola un blocco del genere sa che provoca modifica random ma solo nel blocco di 128 bit corrispondente.

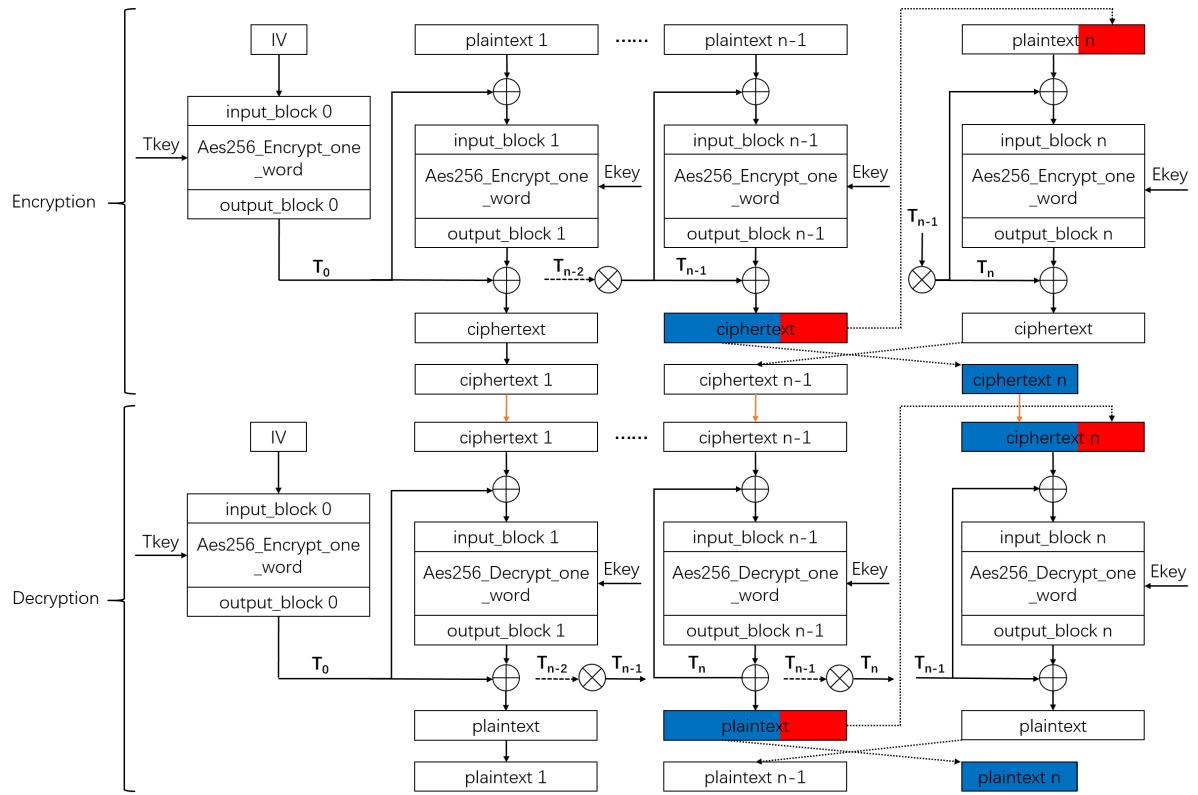
L'attaccante non sa cosa sta modificando, ma sa bene che area di dati, disco o file sta modificando. Wide block è una modalità che ha un obiettivo modellistico andare a creare una super pseudo random permutation, da una pseudo random permutation. Un block cipher è una pseudo random permutation, qui immaginiamo che si effettui una trasformazione da block cipher di 128 in un block cipher di 4096 byte. Voglio fare in modo che l'attaccante riesca a fare modifiche di un blocco molto molto più grosso. Non faccio che modifichi tutto quanto il disco, ma in certi limiti voglio ampliare l'effetto delle modifiche. Si sta andando verso la wide block encryption. *Questo non introduce troppi overhead?* Si.

[22-03-2023]

Riepilogo lezioni precedenti

<https://www.kingston.com/it/blog/data-security/xts-encryption>

<https://en.wikipedia.org/wiki/XTS>



Abbiamo accennato gli schemi sicuri rispetto ad attacchi di tipo CCA1. Gli schemi CCA2 aggiungono per forza un overhead di spazio. Gli schemi resistenti ad attacchi CCA1 sono molto utili per situazioni in cui vado a cifrare il disco e non voglio avere alcun tipo di overhead del disco. Abbiamo alcuni standard importanti, il migliore è sicuramente XTS, più popolare per altro. Altri due schemi molto popolari: adiantum e HCTR2, che sono proposte già implementate in android e HCTR2 da poco anche nel kernel linux per avere livelli di sicurezza più alti rispetto a XTS. E' importante avere una latenza bassa, performance e buona compatibilità. XTS è una modalità sicura, mentre adiantum e HCTR2 sono sicure in una modalità di white box encryption. Narrow block o wide block encryption? La differenza è come propagano eventuali attacchi di integrità. Se uno va a fare modifiche minime, decifrando si ha un effetto imprevedibile sul tutto il testo cifrato ma sul singolo blocco. Con wide block quando cifriamo il dato è come se vedessimo praticamente un unico grande blocco e l'effetto di manipolazione incontrollabile è su una dimensione più grande che posso gestire e parametrizzare, oltre che controllare. HCTR2 è pensato anche per andare a cifrare dati più grandi di 16 byte, ma non esageratamente più grandi. Questo fu progettato per sfruttare AES, ma per funzionare abbastanza velocemente in caso in cui il wide block non fosse così tanto più grande della block size.

XTS

Immagina di avere a disposizione un block cipher che ha un parametro in più.

XEX mode → concettualmente combino il block cipher con una funzione mac. Il risultato del mac lo vado a xorare nel processo di cifratura. Il mac viene xorato sia prima col plain text che dopo col ciphertext. *Slide 25*

In questo modo rendo impredicibile il testo cifrato nella maniera più assoluta. Come gestisco le chiavi? Mac ed encrypt le vado a gestire in maniera completamente autonoma tra di loro.

Come posso andare a non usare padding per proteggermi da padding attack?

Potrei andare a fare cipher text stealing. Questo non è usabile su un solo blocco, e questo è uno svantaggio. Prendiamo il penultimo testo in chiaro e poi prendo quello che gli manca per andare a raggiungere la dimensione che mi serve nel plain text. Prendiamo che mi mancano 2 byte per avere un blocco giusto da cifrare. Con stealing prendo un testo cifrato, l'ultimo che ho cifrato, poi prendo gli ultimi due byte di questo e li aggiungo al mio plain text. I byte che rubiamo dall'ultimo blocco possono NON trasmetterli nel testo cifrato finale! Solitamente alla fine vado ad invertire i blocchi finali perché nella fase di decifratura prima decifro l'ultimo blocco poi il resto. Come faccio a distinguere. Non stiamo creando uno schema forte resistente a tutto, soprattutto ad attacchi attivi, ma preveniamo alcuni difetti su attacchi su padding e ci interessa perché preserva la dimensione ed è un pochino più arzigogolato.

Wide Block Encryption

Mette a disposizione un cifrario che però ha un altro input. Extra: Hints at Adiantum design



[Full Crittografia Applicata \(1\)](#)

Random e Pseudorandom for Cryptography

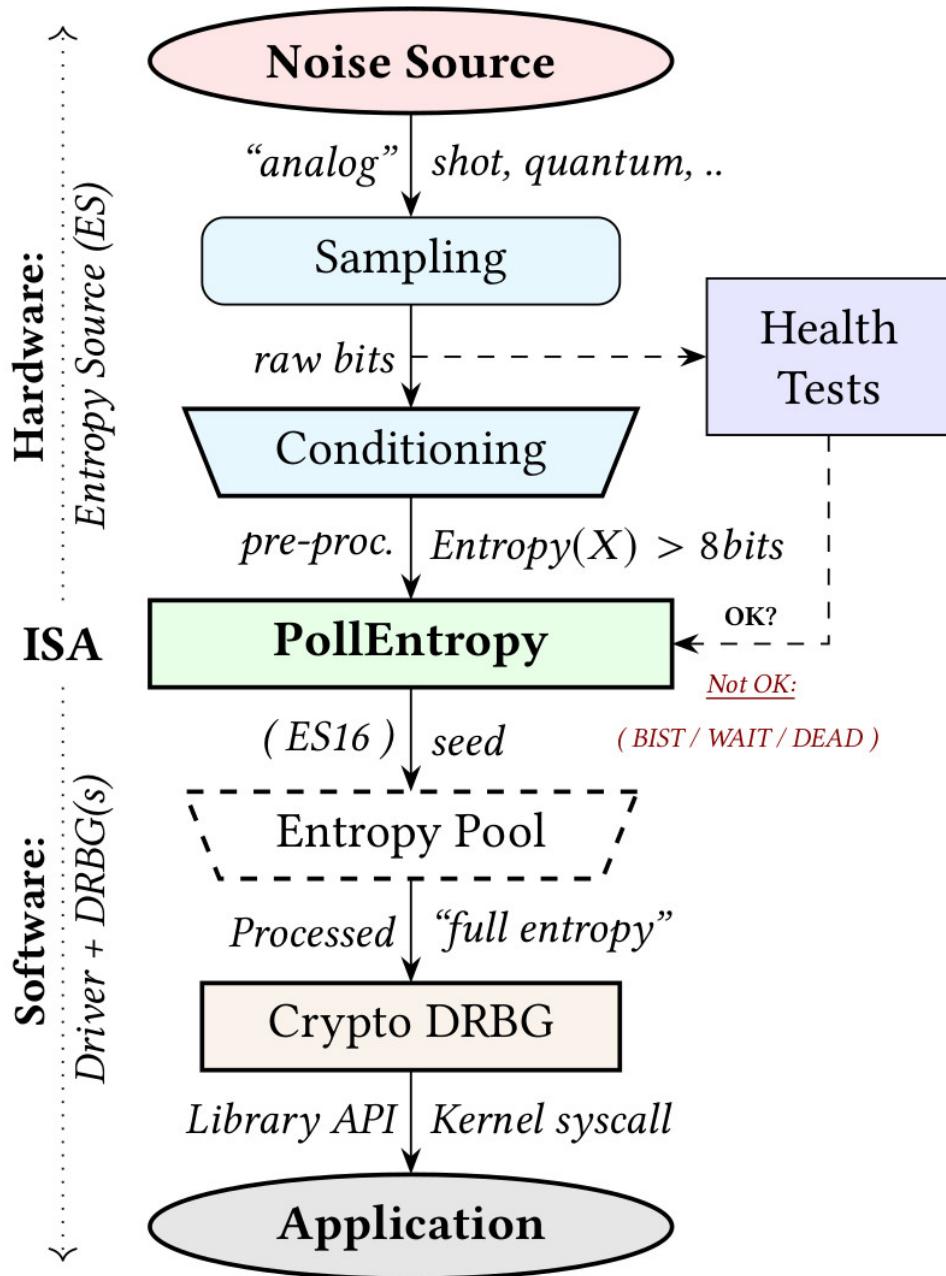
Abbiamo il grande tema e problema della generazione di dati random. Nel mondo informatico moderno che cosa vuol dire andare ad avere dati random? Nell'ambito della crittografia ha interesse particolare. In crittografia è connesso al fatto di non predicitività, oltre che questione statistica. Questa è estremamente importante. Partiamo da una assunzione: noi progettiamo software e i computer sono per natura

deterministici. La parte di non predicitività la vogliamo prendere dal mondo reale, tipicamente visivo e fisico. Sul mondo fisico però abbiamo alcuni eventi che sono davvero random e altri che sono decisamente più predicitivi. Noi cerchiamo di collegare il computer ad un mondo fisico che è fatto in questa maniera, quindi elementi random o elementi difficili da prevedere. Un sistema informatico in un ambiente completamente software non potrebbe essere mai random davvero. Nella crittografia la misura di quanto è random un evento la misuriamo con **entropia**. Questa definisce il numero di possibilità e combinazioni in cui ci potremmo trovare ad avere a che fare. In particolare dobbiamo distinguere numeri random di chiavi crittografiche e numeri davvero random. Sappiamo che una chiave crittografica sicura è sicura se ogni bit ha probabilità di scelta 50% sui singoli bit. L'entropia con dimensione 128 bit è definita come entropia di 128 bit. Attenzione che potremmo avere una chiave random che però è scelta da un algoritmo deterministico che parte da un singolo bit, questo ci porta ad avere un livello di entropia di un singolo bit. Le password non posso assumerla per random davvero, c'è sempre un certo tipo di struttura. Questo vale anche per i *pin* e qui per altro ho ancora meno possibilità di scelta perché tratto solamente i numeri. L'idea di questa lezione sarebbe andare a vedere come generare chiave crittografica che sia sicura e quindi che ci sia sempre la questione dei bit random 50%.

Il nostro sistema operativo ha accesso a diverse periferiche hardware reali: tastiere, mouse, interrupt che arrivano da diverse periferiche. Oltre a queste oggi abbiamo componenti hardware: TRNG true random number generator. Di fatto solo device fisici che internamente sfruttano quei processi fisici che accennavo prima. Ci sono dei fenomeni legati a correnti passive di componenti elettronici o altre questioni elettriche che possiamo andare a sfruttare per andare ad ottenere dati random: <https://www.intel.com/content/www/us/en/developer/articles/guide/intel-digital-random-number-generator-drng-software-implementation-guide.html>

In passato si usavano delle chiavette che erano preposte a questo tipo di attività. Questi dispositivi dedicati sono chiamati *fonti di rumore*. Nei sistemi operativi moderni queste operazioni vengono processate senza andare mai ad usare una singola fonte. A noi non interessa la specifica fonte die dati random quanto che siano credibili. Il SO mixa tante fonti e poi da un flusso di dati che possiamo assumere come effettivamente random. Internamente al sistema ho un'area di memoria che preserva una certa quantità di dati random, che possono essere andati ad usare in qualsiasi momento. Storicamente soprattutto nei sistemi hardware non dotati di TRNG abbiamo pochi dati random a disposizione e cala tanto meno andiamo a interagire con il sistema. Si è cercato di avere un insieme di dati random, ma viene anche fornito uno pseudo random number generator a livello di kernel.

Cioè il kernel ha un generatore pseudo random per andare a usare valori pseudo random, un po' come se fosse uno stream cipher che parte da una chiave piccola per andare a generare una chiave molto grande. A seconda dei contesti in genere si usa sempre roba pseudo random di base. Si va a usare Kernel PRNG perché si pensa possa essere più sicuro, perché magari se ho dati random generati da CPU ma non sono sicuro che sia stato manomesso è un casino. Diciamo che alla fine potrebbe essere molto più sicuro.



Nella modellazione di attacchi nel mondo reale troviamo attaccanti che si collegano ad un certo istante di tempo. Se un attaccante accede al momento di output di dati pseudo random possiamo identificare due garanzie:

- **Forward Secrecy**: un attaccante che si collega ad un certo istante di tempo, non riesce ad accedere ad informazioni passate del sistema. Non abbiamo un accesso agli stati precedenti per la generazione di dati pseudo random.
- **Backward Secrecy**: l'attaccante che accede allo stato 22 non possa accedere a tutte le informazioni segrete generate successivamente. Effettivamente quello che fa l'update è riprendere dati random, non mi affido sempre ai dati random ma faccio in modo di aggiornali di modo che essendo un vero random non possa andare a prevedere dati random reali che potrei andare ad avere nel futuro.

Kernel and User Space

C'è stata una serie di gestione di dati pseudo random a livello di kernel e a livello di spazio utente. C'è una vulnerabilità del 2008 legata al fatto che fino ad un certo periodo il software di generazione pseudo random: openssl era re implementato da zero a livello di libreria stessa. Openssl aveva un suo generatore pseudo random. Il problema è che: questi spesso hanno portato ad avere problemi e fragilità, vulnerabilità per cui ad oggi è meglio che ci pensi sempre il kernel a fornire i dati random. Openssl prendeva pochi bit e generava grandi chiavi! Ma in realtà il numero di bit random era piccolissimo! Qui qualcuno facendo un analisi ad ampio spettro ha scoperto che c'erano tantissime chiavi duplicate. Il numero di chiavi differenti erano davvero poche. Questo era dovuto ad una vulnerabilità interna al generatore pseudo random di openssl. Visto che ciascuna libreria a livello utente ognuna faceva errori, è meglio andare direttamente a livello di kernel. prima si assumeva che il kernel era una merda. Abbiamo anche una considerazione da fare rispetto alle fork: questa copia tutto quanto lo stato del processo ed una vulnerabilità nota è che due processi forkati dallo stesso processo generano dati pseudorandom uguali perché internamente si parte dallo stesso stato per generare dati pseudo random. Ci si raccomanda di ri-aggiornare lo stato del generatore pseudorandom non appena fai una fork altrimenti stai generando dati pseudorandom uguale. Se usiamo un generatore a livello di kernel non abbiamo la fork e non possiamo avere questo tipo di problema. Questa problematica l'abbiamo trovata anche in altri contesti come quello delle virtual machine. Si osservava che snapshot di macchine virtuali clonati e respawnati avessero la generazione di dati pseudo random perfettamente identici. Si notava in macchine virtuali che appena avviate certe macchine virtuali sembrava di

avere chiavi openssl tutte quante uguali. Ad oggi il sistema sa del problema e ogni macchine virtuali viene re-inizializzato con nuovi dati random reali.

Un altro entropia è la bassa entropia nelle prime fasi di un sistema. Magari un sistema si avvia per la prima volta e non ha dati random, perché il sistema magari è vergine e quindi non ha ancora abbastanza dati realmente random. A questo punto potrebbe convenire andare ad usare qualche software, come fonte esterna che ci vengono in soccorso.

In python abbiamo un bellissimo modulo che si chiama *numb*, ... in quasi tutti i linguaggi abbiamo un problema di randomicità. Pensa solamente alla libreria random in python che non va bene per la sicurezza. Per usarla in python sarebbe meglio usare il package os con solamente urandom

```
from os import urandom
```

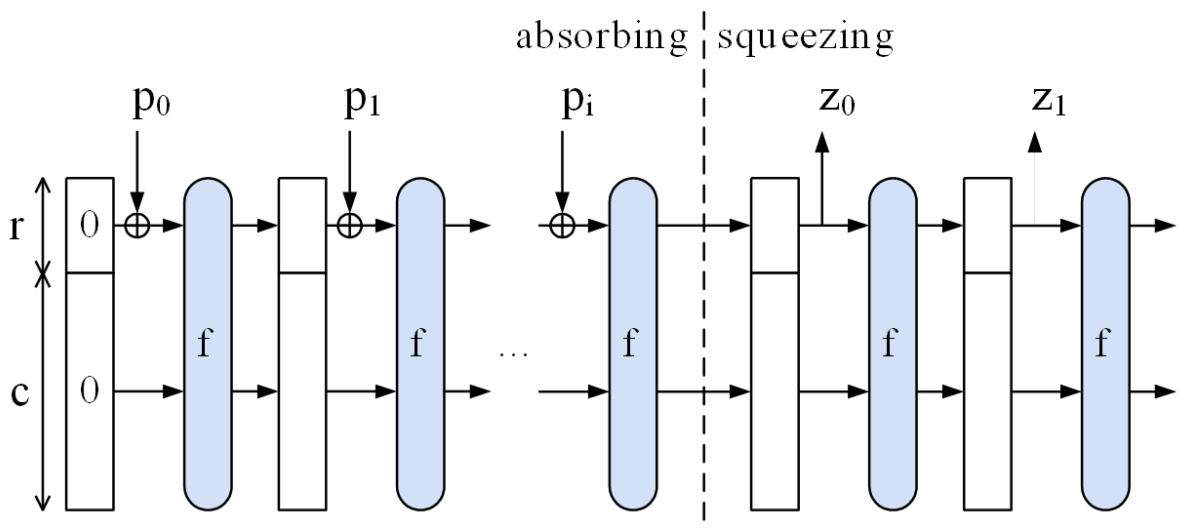
Questo urandom è una funzione che va ad accedere a generatori pseudorandom del sistema operativo. In python se voi farti dati radnom non andare su random ma vai su urandom del tuo os. Abbiamo anche una periferica random (/dev/random) che sono dati effettivamente random prima di essere dati ad un inizializzatore pseudo random. L'unica cosa è che random è considerata una fonte bloccante, per cui se finisco i dati devo aspettare di averne altri per tornare a funzionare. /Dev/urandom invece ti da sempre qualcosa. Nella realtà questa situazione si trova un sacco di informazioni e opinioni in rete. Negli ultimi dieci anni a questa parte comunque ci sono state un sacco di modifiche e potrei trovare una o l'altra implementazione davvero a seconda dei singoli hardware e dell'user space.

In python system randoms in base al sistema operativo posso andare ad python il cui interprete cerca di capire il sistema migliore. Tendenzialmente sarebbe bene usare questo, perché astrae per il programmatore il sistema operativo da andare ad usare.

Symmetric cryptography: derivation and applied schemes

Derivati di SHA3

[23-03-2023]



Sha3 derived functions. Quando è stato progettato ed è stato fatto il contest si era pensato di sopperire a delle lacune di sha3 e si dice di fare un contest per fare qualcosa che fosse realmente scollegato da md5, sha1, sha2, ... in concomitanza sono state pensate che sono riconducibili a sha3, ma che hanno una interfacci leggermente diversa, più vicine a quelle che potrebbe servire ad uno sviluppatore. Tra queste funzioni la più popolare che troviamo implementata in python per esempio è cSHAKE. Le altre sono meno diffuse tipo TupleHash, ParallelHash, KMAC. La prima funzione da che cosa nasce? Quando lavoriamo in funzioni hash queste accettano input di dimensioni variabili (c'è una lunghezza massima ma non ci interessa), l'output invece è di dimensione costante e lo associato a livello di sicurezza con il modello di attacco. In alcune applicazioni ci troviamo a voler generare dei digest più piccoli o anche più grandi. Sono state pensate allora delle funzioni chiamate XOF (extendible output functions) in cui ho un output di dimensioni parametrizzabile. Quando parliamo di avere un digest in output di dimensione variabile parliamo di due contesti differenti: vogliamo avere un digest di 128 bit oppure vogliamo avere un digest più grande. In generale nel mondo reale la funzione più elegante sarebbe andare ad usare una XOF in cui voglio andare ad usare una dimensione X. Nella realtà dei fatti quando voglio un digest piccolo prendo una hash standard e lo vado a troncare. La prima cosa è che con le funzioni hash questo non è un problema, e questo con le funzioni hash moderne non è un problema.



Sha224 ha gli stessi interni di sha256 ma con l'output che viene troncato

In generale se prendiamo una sha256 posso andare a fare un taglio sull'output digest senza andare a creare problemi di sicurezza. Questo lo posso andare a fare anche con altre funzioni derivate da funzioni hash, come HMAC, per cui posso andare a troncare l'output. Con le funzioni MAC se usiamo HMAC non è un problema, se ho altre funzioni mac non è scontato. Potrei andare a provocare delle vulnerabilità molto più forte. Tipo CBCMAC meglio non andare a fare un troncamento. Una XOF in realtà non ha questo problema, ha la funzione cshake. In python potresti andarlo a trovare come shake. Il nome non indica la dimensione dell'output, shake128 e shake256 utilizzano il numero per indicare il livello di sicurezza. Si progetta sha2 in due versioni sha256 e sha212. Anche da sha3_256 è stato pensato shake128, mentre da sha3_512 è stato pensato shake256, questo perché si identifica 128 e 256 come sicurezza a collision resistance. Al posto di avere una funzione hash e ci metto le mani andando a troncare il digest posso andare ad usare una XOF perché è molto più elegante! Una XOF potrebbe anche essere uno stream cipher, ma in base a come è progettato internamente cambia il suo uso. Uno stream cipher mi richiede un input con una chiave con distribuzione uniforme, pensa di andare a chiamare chacha. Non posso dargli come input una password o una parola segreta. Hash o XOF non ha questo requisito. Posso anche dargli un input segreto senza distribuzione uniforme. La differenza tra cSHAKE e SHAKE: è che cSHAKE oltre che accettare la dimensione di input, si prende anche una custom string (domain o context string meglio). questa stringa agisce come un tweak nei tweaked stream cipher. Viene usata per andare a fare domain separation. Quando uso una funzione hash voglio andare a customizzare l'esecuzione per un certo scopo. Nelle funzioni hash questo obiettivo realizzo l'obiettivo nella stringa che mi sto calcolando.

In base alle librerie che si usano alcune usa la dimensione in byte altre in bit! Mi raccomando vai sempre a leggere sempre le documentazioni ufficiali, non quelle riassuntive magari da IDE.

ParallelHash

Concettualmente sarebbe una funzione che parallelizza l'esecuzione del dato...
devo approfondire perché non ci ho capito tantissimo

MAC for SHA3

sha3 è stato pensato per andare ad evitare di avere lenght extension attacks. Non usare mai MAC + SHA2 perché sarebbe un casino sicuramente.

KMAC

Standard basato su una versione modificata di cSHAKE. Questa funzione mac è chiamata come variable lenght mac. Non devo effettuare una troncazione o riduzione del mac ad hoc. Posso andare a definire quanto mi serve e via. C'è una stringa di customizzazion e eposso andare a fare context separation e poi è una funziona mac tradizionale ottimizzata per essere eseguita su sha3, che ha come input dimensione output e stringa di customizzazione. Abbiamo delle scelte interessante come la lunghezza.

Key Derivation Function

Prende come input una chiave e genera una chiave derivata. La chiave in input la definiamo come input key material. E' stato pensato uno standard ad hoc a questo scopo, per aggiungere features a questo standard di derivazione. Una key derivation function è simile ad una funzione pseudo random, in qualche modo è però anche diverso. Dobbiamo tenere a mente che le funzioni che sono state standardizzate come input non accettano una chiave segreta, ma una stringa segreta. E' come la questione degli hash prima. Questa funzione potrebbe anche non essere un valore uniformemente distribuito di n bit. Una key derivation function funziona come una stringa segreta arbitraria. Le key derivation function per essere ottimizzate per derivare delle chiavi sono ottimizzate per andare a genere dati random non tanto più grandi rispetto alla chiave di partenza. Gli internals sono pensati diversamente dalle funzioni di generazione pseudo random. C'è supporto ad input che potrebbe non essere uniformemente distribuiti. Ho anche funzionalità di scoping e domain separation. Vedremo una funzione che da K1 mi genera tante chiavi per diversi tipi di applicazioni! Voglio avere un input che mi permette di andare a customizzare in che contesto quella determinata chiave sarà utilizzata. Il salting e il domain separation spesso ci sono molti casini in internet e trovi brutte implementazioni, tipo come su Django. HKDF è una funzioni di derivazioni di chiavi basata su HMAC con tutte le derivazioni del caso. Questo è proprio uno standard de facto. Prendo in input una chiave che può anche essere una stringa segreta e nemmeno una chiave crittografica a distribuzione uniforme. Si prende subito in input anche un valore che si chiama salt. Questo è una informazione a volte pubblica, ma a volte no. Questo dovrebbe essere effettivamente ad una chiave crittografica, quindi n bit a

distribuzione uniforme. HKDF extract sarebbe partire da una chiave che gli diamo tirarci fuori una chiave crittografica. HKDF expand prende una chiave segreta e mi da un info per contesti diversi, praticamente fa domain separation. Le librerie mettono a disposizione anche i due sotto blocchi perché è utile anche per motivi di performance andare ad usare un blocco anziché due. La fase di extract potrebbe essere opzionale. Il salt lo si può studiare il contesto di esecuzione di HKDF. Il salt posso anche non usarlo se non sono sicuro, non c'è problema.

Entropia

Se parliamo di pin, password e stringhe segrete, le cose cambiano perché l'entropia di una sequenza di n numeri è pari a n volte per 3.32 bit circa se scelgo sempre i bit in maniera totalmente randomica (sarebbe $\log_2(10^n)$). Una stringa di 39 simboli numerici scelti totalmente random è 128 bit di sicurezza (128 per 3.32), viceversa se voglio 128 bit di sec faccio 128/3.32. Per quel che riguarda le password possiamo applicare la dimensione dell'alfabeto, se ho 62 simboli uso l'operatore 5.7

Accetto stringhe segrete ad ALTA entropia, altrimenti genero pseudo chiavi vulnerabili, perché parto da poche combinazioni di partenza. Meglio dare stringhe che siano codificate e mantenute come stringhe alfanumeriche. La parte di estrazione di HKDF ha già entropia, ma non si presenta come formato binario e alla fine devo vedere HKDF come una sorta di conversione. Devo avere una entropia alta, perché tutto gioca sull'extract. Quando uso delle informazioni segrete con entropia BASSA introduco un'altra tipo di derivazione: password based key derivation function. Ho davvero un'altra famiglia di funzioni.



Non usare HKDF con password generate da utente, perché hanno bassissima entropia e posso generare solo altre chiavi fragili

[29-03-2023]

Ha fatto un notebook in cui ci sono su python vari tipi di attacchi implementati in vari tipi di situazioni.

Hash tables flooding and small tag MACs

Argomenti derivati da cifratura simmetrica e di come questi costrutti vengano applicati anche nel mondo reale. Oggi parliamo un po' di un contesto diverso:

funzioni hash e funzioni mac, con considerazioni sulle funzioni per uso crittografici e per garantire anche i canali di comunicazione non sicuri. Le funzioni hash le sono andato a vedere anche per altri tipi di strutture dati. Le hash sono famose perché hanno tendenzialmente un costo costante. In una hash table abbiamo dati che vengono tenuti in hash buckets e so in che posto è salvato il mio dato usando una funzione hash. Le hash tables sono ottime per il loro costo, anche se ho delle differenze dipende dalle dimensioni dei bucket. L'hash table sono usate in miliardi di contesti diversi, soprattutto se la hash table sono in un contesto attaccabile dall'attaccante. Interessante se l'attaccante sceglie o trova tanti messaggi che trovano tante collisioni. In questo caso non è più efficiente la funziona perché non è uniforme la distribuzione di indici, perché letteralmente vado ad esplodere su singoli indici e ricado nel worst case scenario. Questo tipo di attacchi è definito come denial of service → attaccante che cerca di violare la disponibilità del servizio, quindi l'accessibilità e la disponibilità.

Dobbiamo andare a pensare quanto sia facile e difficile per un avversario andarsi a calcolare determinate collisioni: quale aspetto rende più o meno facile o difficile? Sicuramente quanti bucket sono può essere determinante da questo punto di vista (= quanto è grande il codominio della funzione hash). Siamo in una situazione **collision resistance**, in cui scelgo i messaggi che voglio.

Il codominio solitamente è piccolo, perché il numero di bucket non è mai paragonabile al livello di sicurezza a cui siamo abituati, come due alla 128 bucket. Anche se immaginiamo una funzione hash davvero ideale il fatto che il codominio sia piccolo ci richiede di andare sicuramente verso un'altra soluzione. Modifichiamo la funzione hash per avere non una funzione hash ma una funzione mac → non voglio che l'avversario calcoli offline le connessioni. In questo modo l'avversario non riesce a calcolarsi messaggi che generino lo stesso output. La funzione hash di una hash table è scelta solitamente per essere di base estremamente veloce in termini di look up nella bucket table. Quando questo tipo di attacco è diventato rilevante è stata prodotta una funzione mac apposta. Questo standard è una funzione specializzata per fare tag più piccoli del solito → 64 bit che potremmo. Questa funzione è stata progettata per scopi più limitati, ma sicuramente ad hoc rispetto al caso d'uso. Questa chiave segreta chi la mantiene e sceglie? In realtà non è molto importante.

In python le hash table sono ovunque soprattutto per la gestione delle variabili che vengono gestite dall'interprete. Quando avvio l'interprete python questo crea una chiave segreta a run time che verrà usata per tutte quelle le hash tables che verranno usate durante la sua esecuzione.

Commitment Schemes

Schema di Commitment è uno schema usato nei protocolli effettivi che serve a delle entità e che le previene di negare di aver preso certe scelte in passato. Una la chiamata commit e una la chiamiamo open. Alice non potrà mentire a Bob che in passato ha scelta una determinata cosa. Questo protocollo ha una valenza intrinseca, che ha valore a prescindere da come poi effettivamente venga istanziato. Assumo che ci sia un canale di comunicazione sicuro!

La prima proprietà è di binding, ovvero di vincolo: alice non può dare un valore che sia diverso, rispettando il commitment stesso, non può dichiarare il falso essenzialmente.

La seconda proprietà di hiding: definisce che il commitment stesso protegga la confidenzialità del valore di partenza. Questa sarebbe opzionale, perché potrei esser in contesti in cui potrei scegliere X per cui sia difficilmente indovinabile anche se non ho hiding rispettata. Ho hiding di default quando ho un N che può essere scelto così alto che è impossibile andare a indovinarlo. L'alternativa è usare un commitment basato su HMAC o su costruzioni analoghe. Non tutte quante le funzioni mac sono utili per questo tipo di contesto. Non posso ridurre in black box il commitment hiding.

Questo discorso aiuta ad andare a capire altri protocolli dopo, ma spesso online non vengono indicati con i loro veri nomi come in questo caso. Attento che cambia la cosa e cambia anche il modello di attacco possibile.

Synthetic Initialization Vector

Nome che identifica una classe di modalità di cifratura che garantiscono che anche se usiamo il nonce la modalità è pensata per mitigare il danno. Le modalità SIV sappiamo in automatico che se vediamo un nonce probabilmente non facciamo un disastro. Se ricicliamo un nonce più volte con lo stesso messaggio ci stiamo esponendo, ma non stiamo andando oltre ad uno schema effettivamente deterministico. Synthetic sta per una fase ulteriore di generazione derivata per l'IV che sarà utilizzato durante la fase di decifratura.

SIV + AES

La dimensione della chiave di questa schema è doppia rispetto a quella tradizionale. L'input è la concatenazione di due chiavi indipendenti una la do alla PRF(CMAC). Per evitare attacchi correlati una parte della chiave la usa PRF e una parte la va ad usare il block cipher. L'altro standard molto popolare è quello: AES-GCM-SIV che estende direttamente AES-GCM e per realizzarlo si aggiungono direttamente questi

blocchi necessari. C'è una derivation function anche per creare due chiavi indipendenti dalla chiave di partenza si generano diversi chiavi in cui le si usa in diversi passi di questo contesto di cifratura. Però in questa modalità viene embeddata. Il motivo principale è che questo mi permette di andare a ripensare solamente una parte del mio paradigma senza andare a modificare anche le altre cose → *cryptographic agility*. Più cose nascondiamo all'interno più mantengo le interfacce simili e quindi lo integro meglio in contesti pre esistenti. SIV ha funzioni di derivazioni . Se anche noi andiamo a riciclare il nonce, se questo è una pseudo random function forte avrei un SIV completamente diverso anche con nonce identici. Molto comodo da questo punto di vista. Attenzione perché non è sempre così bello. Che costi ho? Nell'ambito di questi standard, abbiamo anche una limitazione della gestione di dato in chiaro in maniera completamente in linea. Devo passare almeno due volte sui dati in chiaro. Genero un IV, che serve per cominciare a cifrare, poi dopo cifro e il nonce lo processo facendo tutto il testo e poi cifro solamente quando ho un IV sintetico. Quindi in termini di performance non è il massimo. Per dati di grandi dimensioni non è per nulla conveniente, ma è un trad off insormontabile al momento. Ci sono altre modalità che cercano di essere resistenti.

La cosa interessante è che una delle applicazioni principali sono i contesti in cui voglio andare ad avere schemi di cifratura deterministici. In quei contesti ci sta usare schemi SIV che per costruzione devono avere sempre lo stesso IV. Uno schema deterministico è tale in cui l'IV è costante. Solo lo schema SIV ci permette di avere un riutilizzo dell'IV senza avere vulnerabilità.

Se siamo interessati ad altre librerie di cifratura è una libreria che si chiama tink di Google. All'interno abbiamo uno schema deterministico **daead**.

Strutture dati Autenticate

Qua parliamo di funzioni hash. Nell'ambito degli schemi applicati abbiamo una cosa molto importante; il termini di struttur dati autenticata è una estensione della struttura dati delle funzioni hash. All'inizio abbiamo visto un content provider che non vuole gestire i dati ma li fa vedere ad un sito mirror e l'utente può controllare l'autenticità dei dati. Il problema di una funzione hash è che possiamo garantire l'autenticità dei dati solamente nella loro integrità. Mi calcolo il digest solamente quando ho tutto quanto. Qui posso fare che al posto di scaricarmi completamente il file posso avere dei protocolli di query in cui posso verificare determinate proprietà. L'esempio più classico è un Merkle Hash Tree, praticamente un labero con le funzioni hash.

Iniziamo creando l'hash di ogni valore, e ogni nodo superiore è l'hash della concatenazione dei valori sottostanti. Cioè ogni nodo è l'hash dei figli. Come mai

questa struttura dati è più intelligente che concatenare tutto e avere un hash direttamente? Una questione utile è che questo schema di hashing è facilmente parallelizzabile e quindi questo è comodo in termini di performance, nonostante l'overhead id mantenimento dei digest intermedi. La parte interessante è che posso andare a costruire lo scenario differente. Mettiamo che io conosca la radice dell'albero. Immaginiamo che sia V1. nel momento in cui noi otteniamo dal mirror un valore, come faccio ad essere sicuro che i suoi figli tipo x3 non siano stati manipolati? Io devo avere dal mirror un insieme dei nodi che mi permette di ricostruire la radice e alla fine faccio un diff dal mio prodotto e quello che viene dato dal sito e che dovrebbe essere corretto. Praticamente vado a vedere tutti quanti i siblings adiacenti fino alla radice. Il client che ottiene questi dati concatena tutto quello che c'è da concatenare e controllo V1. In generale MHT ci danno questa possibilità e ci consente di avere una verifica del valore di appartenenza in maniera forte e crittografica. La dimensione della prova è logaritmica rispetto a quello del dataset. Questo motivo di schema è estendibile anche per altri alberi ed è applicabile anche ad altre strutture dati. Così come esistono prove di appartenenza posso anche a fare delle prove di NON appartenenza. Dimostrare che esistono due elementi consecutivi equivale a dimostrare che il numero che abbiamo chiesto che non esiste, effettivamente non esiste. Questo concettualmente, non stare a morirci sopra, vedi altre fonti.

Ricordiamoci che la struttura autenticata è intrinsecamente sicura, ma se cambia nel tempo c'è una proprietà non banale che devo risolvere con protocolli nell'intorno ce la freshness. Se questo database cambia devo essere sicuro di stare usando la versione del digest corretta e aggiornata. Altrimenti sono cannato, perché il sito mirror potrebbe a quel punto trarmi in inganno. Attento anche ai reply attack perché non sono perfette come strutture. La freshness è difficile da garantire e abbiamo sempre dei trade off sicuramente. Nelle block chain questa cosa è normalissima.

Format-Preserving Encryption

Quando parliamo di cifratura è chiamato FPE, è molto di nicchia. Praticamente l'idea sarebbe andare a cifrare i dati facendo in modo che da decifrati facciano parte dello stesso dominio dei dati in chiaro. Immaginiamo il sistema di carte di credito o anche codici fiscale. Questo tipo di schema ha diverse necessità in cui magari alle volte vogliamo nascondere solamente una parte e l'altra la mostriamo all'utente per fargli verificare solamente le ultime 4 cifre della carta di credito. La format preserving encryption è molto particolare perché vuole gestire la cifratura dei dati mantenendo il formato → ho spesso un dominio molto piccolo e spesso non definibile in base due,

questo è un csaino in realtà, perché noi ragioniamo sempre in base due quando vado a fare encryption. Questi schemi sono simili ai disk encryption in cui ho dei metadati che possono agire per andare a randomizzare i dati, ma qui ho dati davvero piccoli e con domini molto piccoli. Questo argomento sarebbe tutto da approfondimento. Se mai ne avessi bisogno devo andare ad usare FF1 e FF3-1, raccomandati. Hanno una caratteristica per attacchi noti comunque lavoro su domini che comprendono un milione di elementi!!! Nella maniera più assoluta non vado mai ad usare domini più piccoli.

Authentication Protocols and Cryptographic schemes for password protection

[30-03-2023]

Recupera queste informazioni dal corso di Mantova perché alcune cose vengono riprese pari pari.

Quando parliamo di protocolli di autenticazione ci sono delle primitive che possiamo vedere sotto un altro punto di vista:

- identificazione: mappare univocamente una informazione su una identità, magari un utente che deve accedere a delle risorse. Parte che viene prima della autenticazione. Processo per verificare esattamente chi dico di essere.
- autenticazione: processo a garantire che la fase di identificazione sia andata a buon fine essenzialmente. Spesso è molto accoppiata con identificazione.
- autorizzazione: distinguiamo da autenticazione, ed è un processo che determina a cosa può accedere una entità che si è già autenticata.

Solitamente si usa AuthN (autenticazione) per dire se un utente ha permesso di accedere al sistema oppure no. Per intendere autorizzazione si usa AuthZ. Abbiamo anche dei protocolli per rendere molto accoppiato autorizzazione e autenticazione → access control, authentication based.

Solitamente abbiamo a che fare con questi blocchi:

- utente che ha una credenziale
- può usare uno user agent, componente software intermedio, per autenticarsi. potrebbe essere più o meno elaborato ovviamente

- canale di comunicazione
- servizio di autenticazione (server): composto dal fronted (API con cui interagisco) ed un db (con le credenziali degli utenti). Vediamo come ci potrebbero essere anche informazioni differenti rispetto alle credenziali di un utente e il suo account presente sul server

Abbiamo anche protocolli tipo Single-Sign On, Identity Federation (OpenID), Authorization delegation (OAuth), ... che hanno sempre gli stessi componenti ma sono un po' diversi. Parliamo anche di fattore di autenticazione: elemento che l'utente potrebbe usare per andare a identificare la propria identità. Identifichiamo anche delle macro categorie. Immaginiamo che l'utente abbia degli strumenti che sono proprio anche dei componenti fisici per poter accedere ad accedere al servizio: token usb o radio che ci fanno accedere. Il più grande metodo è quello basato sul segreto, quindi: pin e password che conosco solamente l'utente. L'ultimo metodo è una cosa che caratterizza l'utente: la firma, per esempio, su cui la sicurezza si basa sulla replicazione della firma essenzialmente. Abbiamo anche aspetti di autenticazione di tipo biometrico. Questi sono comodi, ma in contesti di altra sicurezza è molto vicino alla situazione di identificazione e non potrebbe essere sicuro, anche perché non possiamo andare a modificare tali valori. La famosa multi factor authentication, ci permette di andare ad usare più strategie e si ottiene un livello di strategie più alto soprattutto se uso strategie differenti: qualcosa che possiedo e qualcosa che conosco. Attenzione che in base ai contesti i fattori di autenticazione hanno delle aree di attacco e dei vincoli, possono diventare dei informazioni di possesso, anziché di conoscenza, quindi cambia la caratterizzazione di un fattore in base a come lo andiamo materialmente a gestire.

Dove possiamo fallire? L'attaccante potrebbe andare a compromettere a livello software anche lo user agente, potrebbe compromettere il canale di comunicazione, potrebbe cercare di interagire col servizio di autenticazione, ha una superficie di attacco in realtà ampia e dipende da che cosa vuole provare a fare. Potrebbe violare il servizio di autenticazione per effettuare un data breach delle credenziali, per cui ruba lato server delle informazioni. In base a che cosa vogliamo difenderci dipende che cosa vogliamo andare a fare. Solitamente sono ortogonali le soluzioni di difesa e attacco, invece a volte sono accoppiate. A volte non possiamo pensare ad un protocollo di scambio di credenziali senza pensare a come le chiavi sono conservate all'interno del db del server. Gli attacchi si possono fare ad ampio spettro, ma spesso dobbiamo andare a pensare il sistema nel complesso. Non posso dire di usare un MAC e fregarmene di che cosa o nel db delle credenziali, perché ho tutti aspetti strettamente accoppiati.

Che protocolli ho per autenticarmi e scambiarmi le informazioni necessarie per fare ciò su un canale di comunicazione?

se voglio di autenticare delle persone, user authentication, ho una strategia che riguarda l'autenticare il fatto che il servizio stia parlando con un utente umano oppure no: reCAPTCHA essenzialmente. Devo andare essenzialmente a fare un touring test. Relaizzare captcha efficaci diventa sempre più complicato, ma solitamente sono sempre le stesse cose:

- riconoscimento di simboli
- riconoscimento di immagini (reCAPTCHA), uso cose del mondo reale per vedere di capire con chi sto parlando

Fatto ciò posso mandare direttamente la credenziale segreta sul canale di comunicazione. A volte questa secret credential potrebbe anche essere una stringa random, una API key, una stringa random. A volte questa credenziale è chiamato beerer token, perché un token di autenticazione per cui presentare il token stesso è sufficiente per autenticarsi. Se in qualsiasi modo l'attaccante viola il canale la sessione è compromessa e l'attaccante può effettuare un analisi della informazione e accedere alla credenziale e loggarsi al tuo posto essenzialmente, fino a che non revoche le informazioni

Introduciamo il concetto di **one time credential**: anche se l'avversario intercetta una credenziale non è possibile andarla ad usare successivamente. Perché questo invalida ogni credenziale appena la vede. Teoricamente potrebbe autenticarsi comunque prima attaccando online, ma se è passivo ha sicuramente perso il treno. Aumenta la dimensioni delle credenziali lato client e lato serve co questo sistema e diminuisce il numero di volte i cui possiamo loggarci.

Cerchiamo ora di avere un sistema in cui non inviamo esplicitamente la credenziale per fare in modo che l'attaccante anche se intercetta la credenziale, non può impersonare l'utente → challenge response protocols. Informazione univoca per ogni processo di autenticazione. L'utente è in grado di fornire una risposta solamente se possiede una informazione segreta. I protocolli challenge response, sono fondati sulla presenza di uno user agent. Questo tipo di protocollo cerca di prendersi tutti i vantaggi avendo meno svantaggi possibili. Se abbiamo un attaccante passivo che intercetta una challenge response è come la situazione di prima, dipende se un attaccante è attivo oppure no. Cercano di essere simili a one time pw. Lo storage lato utente non è proporzionale al numero di autenticazioni che vogliamo

supportare → complessità, rischio di software compromesso con backdoor, oppure complicato avere un hw che supporta sistemi challenge-response. Una variante dei challenge-response sono quelli basati su una base comune esplicita: il tempo. Se i due software hanno accesso alla stessa base di informazioni, possiamo sfruttare quella informazione come challenge. Al posto di usare la challenge usa questa informazione che però ogni volta devo cambiare. Così evito di inviare la challenge e la tengo solamente implicita. In questo modo il tempo diventa una superficie di attacco, e il tempo potrebbe essere “compromesso” in termini di hw che monitora lo scorrere del tempo. Solitamente si usa NTP per sincronizzare il tempo con un servizio apposito, così che se vado fuori sync posso andare a rimettermi in linea. Per cui se un attaccante cerca di convincervi di autenticarci nel futuro, potrebbe essere in grado di sfruttare la pw in futuro. Solitamente è più facile fare attacchi sul sync lato client, che cerca di loggarsi.

Le cose si possono complicare cercando di istanziare questi protocolli. Uno dei più famosi sono i protocolli OTP: one time password.

OATH OTP

Protocollo proposto da OATH, e ci sono degli standard notevoli usati su internet:

- HOTP, RFC4226: hash based one time password , based on hmac
- TOTP, RFC6238: time based on time password, based on hotp

HOTP

Se ho un hmac lo reimplemento preso. Accetto due informazioni:

- chiave K
- counter C

$$\text{HOTP}(K, C) = \text{Truncate}(\text{HMAC}(K, C))$$

Praticamente prendiamo un hmac e lo rompiamo e questo non ha problemi di sicurezza, anche se usiamo tag piccoli che sono quindi più facili da indovinare. Questo pensa alla challenge come un counter. Ogni challenge in questo caso è un valore incrementale, il client conosce lo stesso algoritmo e la stessa pk e la risposta le hopt della chiave k e del client. Per costruzioni di hmac in teoria l'attaccante anche conoscendo c non può andarsi a fare hmac di c senza sapere la chiave. La probabilità di indovinare dipende dalla dimensione del tag. Il contatore deve essere univoco, non devo avere ripetizioni altrimenti mi autentico di nuovo. Se il client è in

grado di avere uno storage persistente posso implementare delle difese ulteriori. Il protocollo potrebbe essere declinato in vari modi, questa sarebbe la versione base. Questo HOTP potrebbe anche essere software fatto da un componente distinto. Anche le usb di autenticazione supportano HOTP e anche se non so il segreto, ho accesso all'interfaccia di invocazione, in questo caso alla chiavetta usb e la sua interfaccia a disposizione. Prova di conoscenza e di possesso dipende da come istanziamo il protocollo in contesti diversi. Se il segreto è gestito da una memoria che posso accedere è una prova di conoscenza, altrimenti di controllo. Se do come challenge non più il contatore ma il tempo sono in TOTP. Un metodo HOPT potrebbe essere anche io che copio il codice steam arrivato via mail. A volte anche HOTP può essere eseguito da HSM, da hardware dedicato essenzialmente, così anche se il software web viene violato, l'attaccante ha solamente accesso ad una interfaccia che viene invocata per ottenere delle risposte. Queste USB, o hw embedded con HOTP, implementano solamente HOTP, non TOTP. Anche se uso hw dedicato, il tempo glielo diamo noi attraverso il SO. Possiamo anche fare key derivation sul server e darlo al client all'inizio.

TOTP

Mando il tempo, ci tolgo un offset, e posso anche decidere per quanto tempo sia valido, quindi lavorare anche sulla granularità.

PIN e password deboli

Posso dire per definizione che le password sono segreti deboli. Le pw assumo che non siano segreti deboli. Quando parliamo di debolezza di una pw abbiamo due tipi di debolezza:

- enumerazione → suscettibile ad attacchi a dizionario
- brute-forcing → rotta con brute forcing

Dobbiamo garantire che queste cose non siano possibili per un attaccante. Se dobbiamo pensare ad un sistema del genere dobbiamo pensare che il frontend prenda delle contromisure: ritardi in caso di tentativi di log multipli (evito attacchi online a tutto buso), rendere invalida la pw, inseriamo un captcha, ... questo tipo di sistema è sicura fino a che la password non è enumerabile. Altrimenti se no ho un numero di tentativi così basso da farcela. Se no ci provo troppe volte e sono sicuro. Un sistema alternativo è quello per cui vado ad usare dei pin, ma ci metto un limitatore che dopo un certo numero di tentativi accoppa il pin, perché è altamente

numerabile. Chiaramente il servizio è protetto, ma invalidare la credenziale è un modo per fare un denial of service molto veloce! Invalidare una credenziale è una scelta forte accoppiata sicuramente un fallback, usi un'altra credenziale e ripristini il pin.

Data Breach

Contesto più importante in termini di password. Una parte fondamentale è andare a gestire delle strategie per proteggerci. Dobbiamo pensare a tecnica di crittografia per andare a fixare questa situazione. L'utente ha una pw che invia direttamente al canale di comunicazione, Il server ha una db che possa essere rubata. Partiamo dallo strawman: sistema vulnerabile per definizione. Il db delle credenziali le salvo in chiaro direttamente, perché chiunque entra nel db può essere qualunque utente registrato → altamente insicuro. Approccio 1: usiamo funzioni hash, al posto di usare la pw in chiaro, conservo digest oppure hash delle pw. Il processo di autenticazione è uni direzionale, perché non devo fare inversione. L'utente invia la pw, si calcola l'hash e si fa un confronto. La pw è salvata solamente hashata. Questa compare a runtime ma non a livello persistente. Abbiamo un problema: il fatto che questo storage conservi l'hash è sufficiente a garantire la sicurezza della pw stessa. Se è una informazione con entropia alta si, altrimenti no se ho degli hash di password deboli. C'è una soluzione tecnica: rainbow tables → per rendere facilmente brute forcabili delle pw non molto lunghe. Posso fare delle strutture dati che mi permettano di rompere queste strutture. La cosa interessante a livello didattico è che da qui in poi tutti i sistemi devono essere studiati secondo una precisa modellazione degli attacchi. Ho diversi istanti di tempo:

1. t0: l'attaccante ci prende di mira e posso fare un pre computation attack.
Spendo tempo e risorse per andare a prepararmi all'attacco, questa parte di preparazione cosa vuol dire effettivamente. Queste rainbow table mi fanno spendere molto tempo prima di eseguire l'attacco e nessuno lo sa effettivamente. Non sto rischiando nulla.
2. t1: momento in cui interagisco col sistema, da qui ho poco tempo per portare a fine l'attacco perché potrei venire rilevato. Rischio che un sistema in cui viene rilevato possa subire una revoca delle credenziali. Cerco di fare tanto t0 per rendere velocissimo l'attacco, dopo la violazione del sistema
3. t2: reversing delle funzioni hash, tempo offline che deve essere breve, perché sono cercato effettivamente dalle guardie o rischio di rilevare un attacco essenzialmente, e quindi potrebbero esserci contromisure.

4. t3: fix revoke della pula una volta che è tutto sgamato e magari hai anche finito di crackare le password

Se uso solamente funzioni hash non sono sicuro. Posso usare un salt: hasho la password anche con questo elemento generato in maniera random e viene conservato insieme alla pw. A che cosa serve se è pubblico? E' pubblico per authentication server e lo diventa solamente nel momento della violazione. Questo non rende *bruteforcabile* la password finché non scatta t1. Questo, attaccante side, è come se fosse segreto e rompere l'hash sarebbe come rompere una chiave crittografica. Se uso salt specifici anche dopo una violazione non posso fare general purpose per facilitarmi la vita per tutti gli utenti, devo farmi una tabella apposta per ogni utente che ha un suo salt se voglio andare a farmi delle tabelle di ottimizzazione. Posso anche parlare di salt segreto in cui non voglio che non sia conservato nello stesso db delle password, ma lo voglio conservare magari in qualche componente hardware dedicato. Praticamente sto guadagnando tempo lato guardie e le rainbow table non funziona più. Faccio solamente pre computation in t1, poi se ho il salt segreto spendo ancora più tempo online. Questa è "guadagno di tempo per le guardie". Salt pubblico è una best practice, lo posso usare in black box essenzialmente.

Al mondo d'oggi l'ultimo step è che al posto di usare delle funzioni hash standard posso andare ad usare funzioni di hash di password hashing essenzialmente. Funzione hash non invertibile solo fin a quanto ho entropia come chiave crittografica. altrimenti uso funzioni che compensano questa debolezza. Posso teoricamente ho funzioni hash non invertibili, ma in cui anche la funzione diretta richiede un costo; prima avevamo situazione invece in cui la produco super veloce, ma l'inversa ci mette tempo esponenziale. Voglio creare un contesto in cui concettualmente, voglio fare in modo di avere un dealy anche quando l'attaccante non interagisce più col servizio, voglio che il bruteforce diventi lento, come se aggiungessi un ritardo che aumenta la sicurezza del sistema. Più è lontano il t3 meglio è. Questo lo facciamo dicendo che usiamo funzioni hash che sono **nativamente lente**, funzioni di password based key derivation functions oppure password, ...

primi approcci sono puramente computazionali. Uso un approccio iterato in cui calcolo hash 1 milione di volte: PBKDF1. Faccio un hash in maniera ricorsiva essenzialmente, con milioni di iterazioni. PBKDF2 ho risolto alcune vulnerabilità in cui in PBKDF1 potessi evitare delle iterazioni, invece qui è molto diffuso come standard e usato in django come default e usato come schema in ambienti legacy industriali. Bcrypt, come sistema di hashing di pw sui sistemi linux. Approccio migliore: in un mondo reale attaccante e difensore non giocano ad armi pari. Quanto

costa quella funziona hash? A livello reale il costo cambia da hw che usiamo per eseguirla ovviamente. Sappiamo di avere hw dedicati per rendere veloce la costruzione di situazioni crittografiche. C'è anche una questione di energia e di soldi. Un attaccante che vuole fare un bruteforce non sta facendo tutto con un sistema gp, ma si comprano architettura hardware dedicata e fatte apposta per andare a fare bruteforcing delle pw. Se ho una architettura che può eseguire efficientemente ciò lo comprano e gli conviene tantissimo (magari anche in paesi in cui costa meno anche la corrente). Ad oggi posso anche andare a noleggiare sistemi cloud, oppure anche usare gpu o asic apposta. Quello su cui si spinge tanto è progettare funzioni lente che lo sono perché richiedono memoria, tantissima memoria. Algoritmi di hashing memory hard → algoritmi basati su grafi che per essere eseguiti hanno necessità di essere eseguiti con una certa memoria ben definita. E' utile legare tempo e occupazione di memoria perché i sistemi hardware usati dagli attaccanti hanno poca memoria o architettura parallelizzabile, ma non ho memoria parallelizzabile. Il primo era Scrypt, ora invece abbiamo: Argon2, che arriva anche come flavors del tipo Argon2i, Argon2d, Argon2id. Queste a seconda che voglia rendere più o meno parallelizzabile in base ai contesti in cui ci troviamo. Ha una interfaccia parecchio customizzabile e posso anche usare il timeconst per dire quante volte eseguire gli algoritmi in maniera simile al numero di iterazioni (tempo di calcolo sulla singola unità), la gestione della memoria e il parallelismo. Possiamo prendere alcune metriche in considerazione: *slide 50* se andiamo nello standard attuale nell'ultima sezione del capitolo 7 ci sono delle raccomandazioni. Potremmo andare a fare anche key derivation con questi strumenti! Pensiamo ai sistemi di cifratura di pw che poi salvo su disco, qui uso Argon2 per andarmi a derivare le chiavi e poi posso andare a fare del tweaking apposta per questo tipo di situazione.



Pensa che in un contesto reale in cui ho un db, mi interessa solamente avere il tempo di revocare le pw quindi posso stare basso con argon, ma con sistemi in locale tipo sul mio computer posso andare giù pesante, perché tanto non potrò mai revocare le pw e devono essere sicure per sempre.

I parametri sono correlati tutti tra loro.

Crittografia asimmetrica

Perché c'è una conoscenza asimmetrica delle chiavi, ma oggi giorno è un po' confusa la faccenda e usiamo parlare di un certo sotto insieme di primitive e costruzioni di basso livello che derivano da conoscenze diverse dalla simmetrica.

Iniziamo chiamando coppie di chiavi come pubblica e privata. Una public key è vincolata in maniera forte alla private key. IN qualsiasi sistema simmetrico considero di distribuire una chiave almeno a due persone, in asimmetrico la chiave privata è conosciuta solamente ad una entità, mentre la pubblica è disponibile virtualmente per tutti, dipende da noi come la andiamo a distribuire.

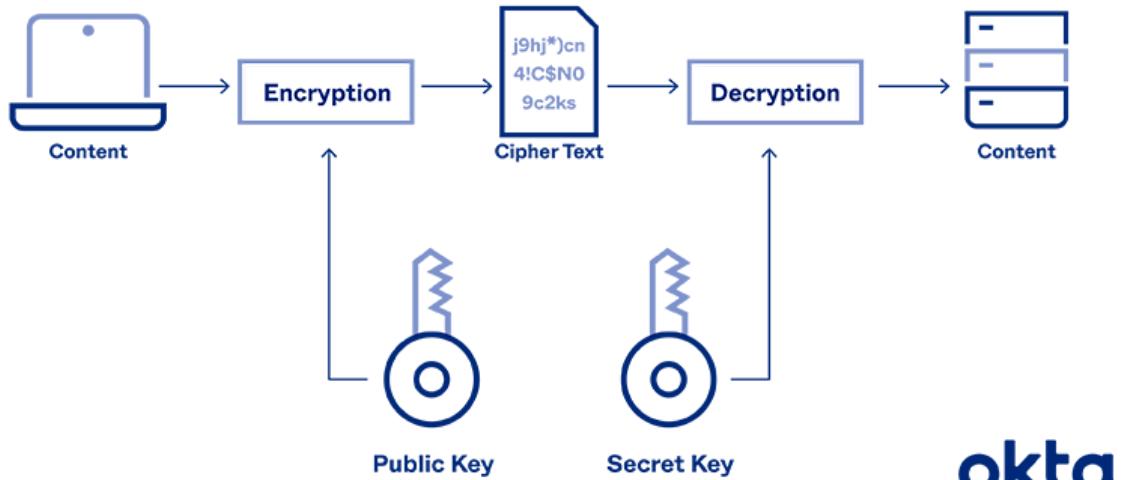
Vedremo alcuni aspetti del tipo:

1. firme digitali
2. cifratura asimmetrica
3. key exchange (protocolli di scambi di chiavi), che ha una sua valenza

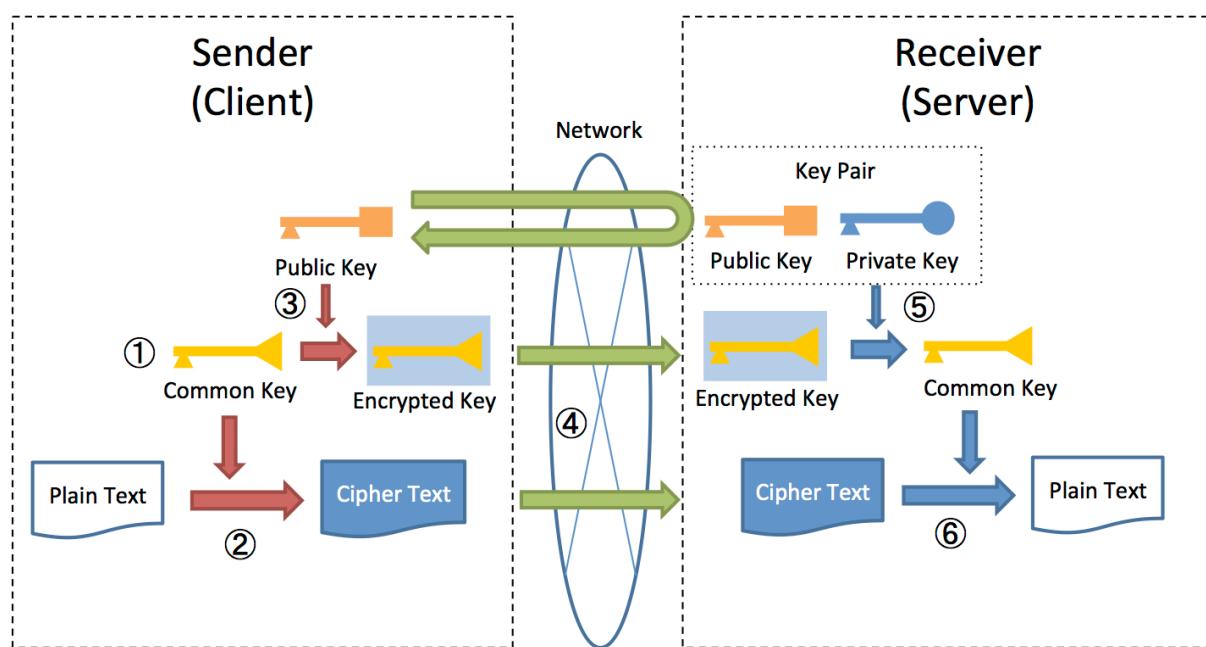
Questi sono 3 schemi primitivi che possiamo ricondurre a schemi matematici. Nel secondo caso possiamo pensare alla key encapsulation, per cui abbiamo uno schema che serve a cifrare una chiave simmetrica. Molto spesso nel mondo reale usiamo asimmetrica ibrida in cui c'è asym per ey encaps e poi simmetrica per parlarci.

In cifratura simmetrica chiunque conosca la chiave pubblica può decifrare i dati, ma per cifrarli c'è bisogno della chiave privata ovviamente. La cifratura asimmetrica ha questo setting.

ASYMMETRIC ENCRYPTION

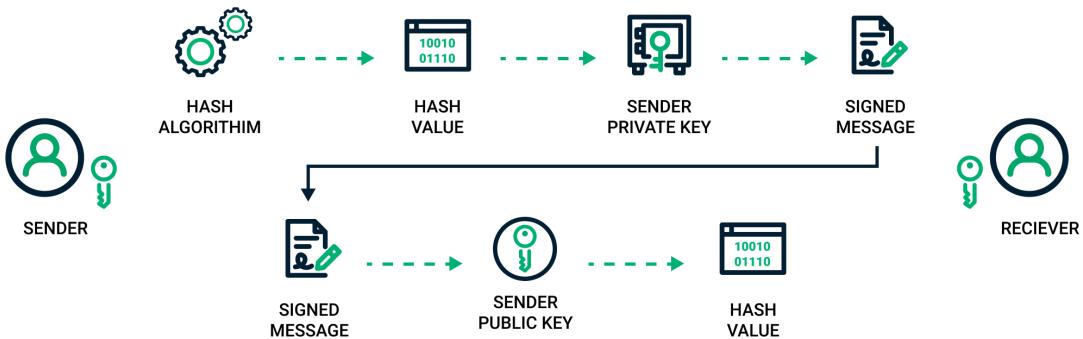


Poi solitamente vado tutto con ibrido:



Crittografia asimmetrica è confidenzialità, mentre firma digitale è autenticità come preconcetto.

How Does a Digital Signature Work?



Il receiver può andare a verificare che il sender abbia firmato correttamente il tutto. E' un sistema molto simile ad un mac, ma aggiunge due garanzie importanti ed ulteriori:

1. verificabilità pubblica, chiunque ha la public key può andare a verificare la firma digitale
2. non negabilità, ma possiamo non garantire la deniability! E' per certi versi un peso, dipende ovviamente dai contesti ecco

In black box in un contesto di key exchange voglio andare ad avere una chiave comune essenzialmente! Il protocollo nativamente assume attaccanti passivi che possono solamente leggere, ma non fanno manipolazione, altrimenti key exchange non autenticato potrebbe essere facilmente violato. Assumo che non via sia per altro alcuna terza parte. Il key exchange potrebbe estendersi, ma dal punto di vista di sicurezza eve potrebbe anche provare ad avere attacchi attivi. Se ha accesso al canale fisico potrebbe fare quel che vuole, ma l'autenticità potrebbe andare a rilevare questo tipo di interventi lato alice e bob! E quindi non andare ad accettare certi messaggi! Quando parliamo di autenticazione non parliamo di autenticazione uguale: potrebbe essere sia una autenticazione da entrambi le parti ma potremmo parlare anche di autenticazione solamente server side. Vogliamo sapere chi sta autenticando i messaggi di chi.

One Way Functions and Trapdoor One Way functions

Dobbiamo capire che un concetto fondamentale che ci serve è un oggetto che ci sembra simile a qualcosa che abbiamo già visto. Una funzione one way è quella funzione in cui possiamo calcolare in un modo efficiente in verso e poco efficiente nell'altro (costi polinomiali - costi esponenziali). Abbiamo già visto delle funzioni one way nelle funzioni hash. Quello che vorremmo usare sarebbe funzioni con una struttura tra input e output e usano dei principi di diffusione e confusione. Meno legami ci sono tra chiavi e output, input e output meglio è. Così vado forte senza essere critto analizzabile. Vogliamo realizzare delle one way functions senza andare a rompere tutta la struttura interna dei dati. Non usiamo operazioni binarie, ma dei problemi matematici per cui ho delle soluzioni e delle strutture, ma che sappiamo non essere trattabili computazionalmente. I crittografi che lavorano in questo ambito sono spesso matematici. Nella crittografia asimmetrica (standard e consolidata), ho tre tipi di costruzioni di cui 2 riconduco ad una unica categoria:

1. RSA
2. Prime Order Fields
 - a. modular arithmetic prime modulo, for finite field on integers (FF)
 - b. elliptic curves ECC
 - c. finite fields on integers

Vedremo che si parla di matematica modulare in cui ci sono delle proprietà in cui $p \times q$ sono dei primi grandi. Questa parte destra i campi che hanno un ordine che è un numero primo grande hanno altre proprietà matematiche. In questi ambiti, non è semplice andare a spiegare. Abbiamo delle proprietà che possiamo andare a implementare con la matematica semplice e poi anche con ECC. Quindi abbiamo RSA, matematica di ordine primo, ma al di sotto posso implementare in modo differente. RSA modular arithmetic modulo product of two prime numbers for cyclic groups e invece il secondo è un po' più complesso. Esistono standard che cercano di usare altri backend matematici per poter resistere al quantum computer: NISTIR 8413, NIST PQC Seminar che sono linkati all'interno della slide 22.

Discrete log on integer finite fields

$$Z_{11} = 0, \dots, 10$$

$$3^7 \bmod 11 = 9$$

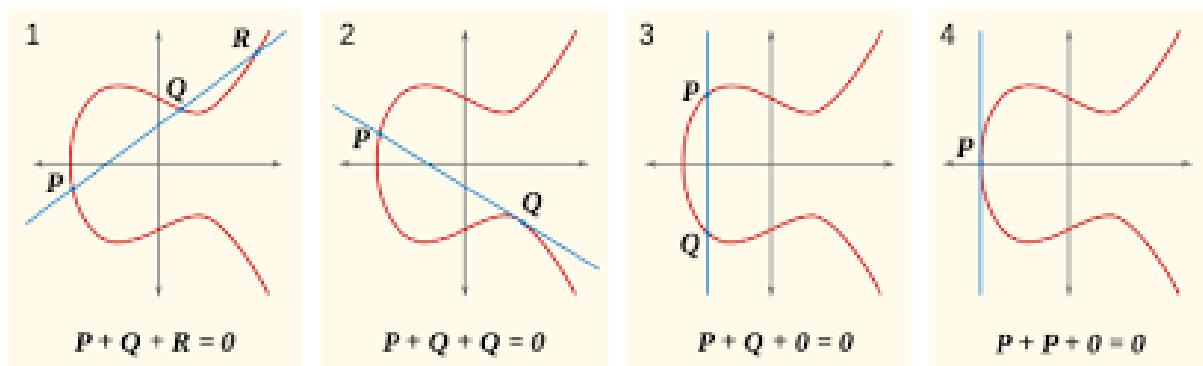
Chi conosce 9 e anche 11 e anche 3, non riesce a calcolarsi 7. Sarebbe il calcolo del logaritmo, voglio l'esponente a partire dal risultato ma non è così semplice da andare a fare soprattutto con numero molto alti. 3 sarebbe il generatore, 7 sarebbe

anche l'esponente chiamato anche ordine del risultato. Sarebbe importante andare a generalizzare per andare a parlare di questi termini. Questo lo generalizziamo in un campo finito e l'ordine di 9 è il numero intero che ci serve per, a partire dalla base, il generatore, calcolare il risultato. Occhio a distinguere i due tipi di ordine: in questo il numero 7 o banalmente il numero di elementi all'interno del campo. Potremmo andare a generalizzarlo con andiamo a trovare l'ordine. E' inefficiente come cosa ovviamente. Questi sono tutti valori pubblici che vado a definire a livello di standard che sono proprio salvati sui file di conf del nostro computer, l'unica cosa segreta è l'esponente o l'ordine del valore pubblico che distribuiamo.

Così come usiamo i numeri interi, così posso andare ad usare curve ellittiche. Immaginiamo di lavorare non con numeri interi ma con delle coordinate. Sappiamo che il nostro gruppo sia l'insieme dei punti su una curva. Che soddisfano una certa forma matematica, solitamente una cosa del genere:

$$y^2 = x^3 + ax + b$$

I matematici che hanno studiato queste strutture si sono inventati di sana pianta le operazioni in astratto di questi tipi di strutture, ma sappiamo che possiamo andare a definire una somma, Questa è ottenuta vedendo l'intersezione di p e q come punti, sapendo che passi anche per un altro punti.



Abbiamo tre curve ellittiche, quelle nella formula citata prima e poi:

$$x^2 + y^2 = a^2 x^2 y^2$$

Le curve ellittiche per scopi crittografiche sono sempre simmetriche rispetto ad uno o entrambi gli assi.

$$x^2 + y^2 = a^2 + a^2 x^2 y^2$$

Ho anche questa variante che è praticamente simmetrica per tutti quanti gli assi. Queste curve hanno senso solamente su domini continui. Ogni valore di coordinata è comunque un numero intero. E' primo all'interno di un numero primo.

Termine interessante: base point, che è un punto su una curva ellittica pubblica. *Vedi slides 29 per maggiori informazioni.*

Esiste una forma compressa, quando del punto dico solamente una coordinata e dell'altra coordinata invio solamente nu bit, questo perché tutte le coordinate sono simmetriche rispetto ad un asse. Per calcolarmi interamente l'altra coordinata eseguo direttamente la formula.

Con le curve di edward praticamente non ho tutti punti ammissibili ma solamente un sotto insieme per cui in fase validazione devo sia validare le coordinate e sia validarle nel senso che si effettui una verifica per vedere se appartengono al punto che mi interessano.

Approfondimento sulla questione del punto mandato compresso

Lorenzo Stigliano, [4/5/23 3:53 PM]

allora praticamente dice che è compressa ma in realtà il discorso è che: dato un punto che ha due coordinate del tipo x,y se so quale sia x so per forza che y possa essere solamente due valori, che sono identici ma uno è negativo e uno è positivo

Lorenzo Stigliano, [4/5/23 3:53 PM]

io anziché mandare per y la coordinata intera, mando o 0 o 1 per dire se sia il positivo o il negativo

Lorenzo Stigliano, [4/5/23 3:54 PM]

la convenzione del fatto che 0 = + e 1 = - dipende dagli standard (IEEE o NIST)

Lorenzo Stigliano, [4/5/23 3:54 PM]

nel dubbio sempre leggere messaggi implementativi

A runtime posso andare a usare anche fino a 4 coordinate con dei parametri derivati che ci permettono di andare ancora più veloce (?)

Questa parte me la devo rivedere da solo

il problema del log discreto è trovare l'ordine di un elemento avendo il generatore, l'elemento e le caratteristiche del gruppo crittografico. Questa è una funzione unidirezionale, so il segreto mi calcolo tutto quello che voglio. Nessuno può calcolare la funzione inversa. Chi conosce l'input di partenza è perché l'ha scelto lui. E' un hash con una struttura matematica sotto stante e ho omomorfismi cioè faccio operazioni matematiche che sono valide anche nel codominio di partenza. Se conosco g^a e g^b , non sono in grado di calcolare $g^{(ab)}$, se almeno non conosco uno tra g e b, questo è chiamato computational Diffie Hellman: assunzione di impossibilità di

calcolo. Abbiamo anche una congettura decisionale. Dato g^a , g^b e dato g^{ab} non distinguo questo da un valore random del tipo g^r . Questo è un problema di indistinguibilità. Queste servono per costruire protocolli di diffie hellman e derivati. Sono interdipendenti ovviamente. Ho alcuni gruppi crittografici e delle curve particolari, che vengono nominati nella crittografia del pairing in cui sceglio delle curve in cui il decisional diffie hellman è risolvibile senza andare a risolvere il computational; questi ancora non sono standard come protocolli.

RSA è una one way trapdoor, che sono funzioni oneway ma in maniera condizionata. Possiamo anche invertirle, sono oneway solamente per chi non conosce certe informazioni.

[12-04-2023]

Tutti gli schemi basati su diffie hellman oggi giorno sono basati su curve ellittiche perché sono più sicure. Abbiamo sicuramente operazioni più complessi da andare a fare, ma riusciamo a lavorare più agevolmente da questo punto di vista e guadagniamo in termini di velocità. Alcune curve standard del NIST:

1. P224, P256, P521

E poi alcune che sono ormai standard de facto:

1. Curve25519 (il numero è $2^{255} - 19$, indica il numero intero del primo campo) and Curve448; questa ultima sarebbe migliore, ma è computazionalmente più onerosa. E' già sicuro usare delle curve che siano più veloci e un po' meno sicure. Pensando soprattutto a situazione di hardware embedded che si deve calcolare queste curve. Queste curve dovrebbero ormai essere state standardizzate anche dal NIST.

Funzioni trapdoor one way è si unidirezionale, ma in maniera condizionata. Esiste un segreto, trapdoor, che ci consente di invertire la funzione che normalmente sarebbe one way. In crittografia l'unica degna di nota, sarebbe quella legata al modello RSA. Questi lavora su moduli che non sono primi, ma sono composti, ovvero prodotto di due numeri primi. Praticamente ho una inversione in maniera selettiva in base alle informazioni che vado a rilevare. Se non conosco i numeri primi ad cui è derivato N non sono invertibile.

L'approccio standard oggi giorno è questo: scelgo random due valori primi, e ci calcoliamo n , poi tengo p e q come nascosti. Poi vado a scegliere un valore di e (esponente pubblico) = $s^{16} + 1 = 65537$, che è un bel numero. Sapendo p , q ed e , posso andare a calcolare tutto quanto quello che voglio.

Con RSA sembra intuitivo andare a costruire sistemi di firme e cifratura senza eccessivi problemi. Abbiamo però dei problemi a livello funzionale, vedi slides 39. Ho due schemi indipendenti tra cifratura e firma digitale.

Sta facendo vedere alcuni esempi sul notebook delle cifrature. Sta parlando anche di funzioni omomorfiche → approfondisci l'argomento in autonomia:

https://it.wikipedia.org/wiki/Crittografia_omomorfica

Ho due problemi molto forti con una situazione di questo tipo:

1. non abbiamo componente di randomizzazione, siamo intresicamente deterministici e rendiamo fragile la funzione di cifratura
2. mettiamo a disposizione uno strumento che permette all'attaccante di tentare, di provare, perché gli sto dando parto delle chiavi di cifratura

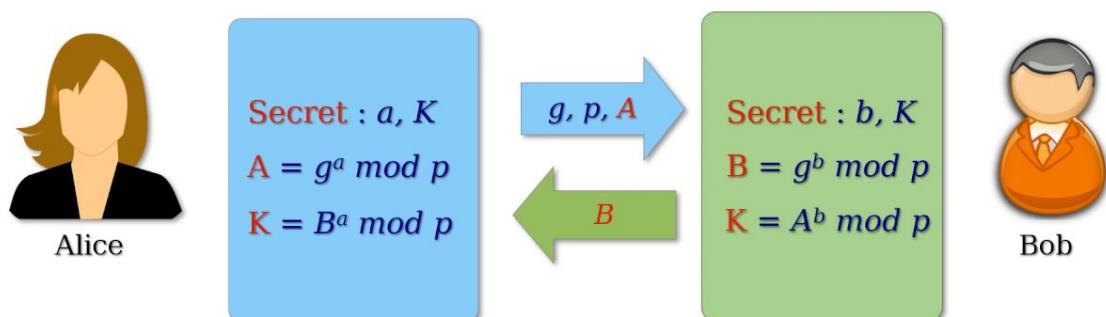
 [Diffie-Hellman Key Exchange \(1\)](#)

 [Cifratura Asimmetrica con Diffie-Hellman \(1\)](#)

Diffie-Hellman Key Exchange

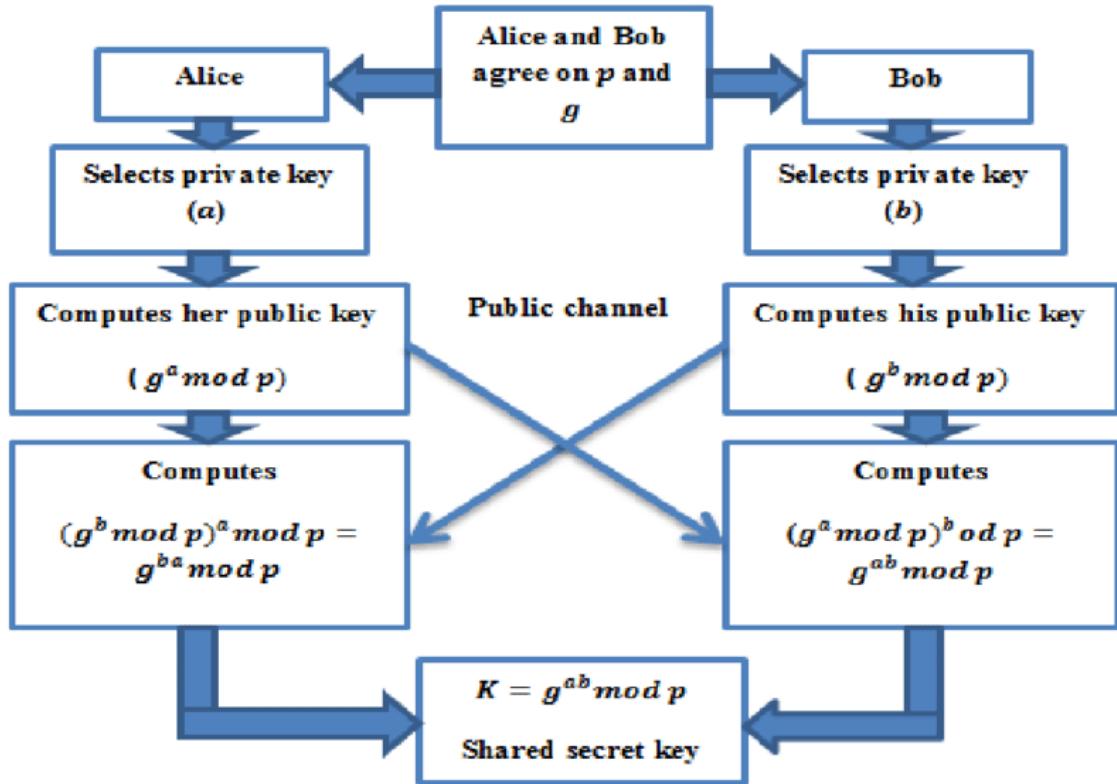
Primo protocollo studiato in campo di crittografia asimmetrica, sono stati i protocolli di scambio di chiavi. Lo scambio di chiavi di diffie-hellman è un protocollo in cui andiamo a usare le congetture che abbiamo detto prima. Qui non metto mod(p), dico che g è semplicemente un generatore che non è invertibile.

Diffie - Hellman Key Exchange Protocol



The Security Buddy
<https://www.thesecuritybuddy.com/>

Si usano chiavi effimere che servono solamente per l'esecuzione del protocollo, mentre g^a e g^b , si definiscono come public contributions.



Un attaccante in questo caso può andare a fare un *man in the middle* come attacco. In questo caso Eve sarebbe un attaccante attivo che fornisce chiavi. Si finge Bob → Alice e Alice → Bob, può andare a fare quel cavolo che vuole e alla fine Eve può decifrare i dati da una parte all'altra senza alcun tipo di problema. Questo perché non ho alcun tipo di protezione per autenticità, proteggo solamente la confidenzialità.

Per fare un key exchange autenticato, vado a combinare il tutto con una firma. In questo modo invio g^a e la firma digitale di g^a . Eve se fa un man in the middle e manda un g^a' , non può riforgiare una firma digitale valida se esiste un sistema di firma digitale sicuro. Bob a questo punto può fare tutte le verifiche del caso per verificare il materiale ricevuto da g^a . Chi deve autenticare deve sapere in precedenza la chiave pubblica di Alice.

Ricordiamoci anche in realtà i protocolli di scambi di chiavi li possiamo costruire con meccanismi di key encapsulation mechanism. La differenza principale è che con Diffie Hellman la chiave simmetrica è costruita con contributi da entrambi i partecipanti e si dice che sia Alice che Bob contribuiscono alla costruzione della chiave simmetrica.

Quando parlo di key exchange qui praticamente la chiave viene scelta unicamente da uno solo dei due partecipanti. Se compromettiamo quell'unico è un casino. Questo setting ce lo teniamo fino a che i quantum non spaccano tutti gli algoritmi.

Cifratura Asimmetrica con Diffie-Hellman

Materiale utile per capire questo argomento

Linko tutta una serie di video utili per andare a capire meglio gli argomenti che vengono trattati in questa sezione:

<https://www.youtube.com/watch?v=NmM9HA2MQGI>

<https://www.youtube.com/watch?v=NF1pwjL9-DE>

Generalmente questi sono tutti quanti schemi ibridi. Perché ho una natura fortemente unidirezionale e non i consente di andare a fare la decifratura asimmetrica. Concretamente sarebbe come andare a fare uno scambio di chiavi asincrono.

Questo schema oggi giorno viene chiamato Hellgamal, come standard all'interno delle librerie troviamo scritto ECIES, praticamente hellgamal implementato su curve ellittiche.

Come costruire firme digitali a partire da Diffie Hellman

Creiamo un programma che praticamente sia un proof of knowledge, ovvero andare a dimostrare di sapere un segreto senza andarlo a rivelare essenzialmente. Come costruiamo tutto ciò? Praticamente vado a fare zero knowledge Proof [of knowledge]. Questo lo posso fare in due contesti:

- interattivo: ho due agenti che sono online e possono parlare

- non interattivo: pensiamo alla firma digitale che funziona anche in maniera offline

Assumiamo che il mittente in questo caso si onesto (Alice = prover, Bob = verifier). Il prover vuole dire a verifier di conoscere un valore senza dirglielo essenzialmente. Vedi slide 53. Verifico che prover sa X , perché uso una proprietà omorfica di diffie hellman. Tutto questo funziona che il prover è fidato e segue alla lettera il protocollo. Assumere però un mittente onesto è una boiata perchè io voglio andarmi a difendere dai disonesti, perché così chiunque può fare messaggi con s ed r fatti in questo modo. Creo un protocollo interattivo, perché le operazioni sono molteplici e inter dipendenti l'una dall'altra. Praticamente vado ad aggiungere una challenge per andare a fare il protocollo. Ora vorremmo andare andare a fare una versione NON interattivo per andare a usare le firme digitali.

Per farne un modello non interattivo, esiste una euristica, o trasformata che viene chiamata: Fiat-Shamir Heuristic. Quello che succede è che spostiamo la challenge dal verifier al prover, al posto di generare la challenge random questa è generata a partire concettualmente da una funzione hash. Il prover genera il commit e poi fa da solo la prova, infine il verifier verifica tutto quanto da solo. La parte di challenge viene eseguita dal prover calcolando la challenge come hash dei messaggi inviati in precedenza nel protocollo. Questo protocollo è un protocollo di autenticazione. Estendiamo tutto con lo schema di slides 57. Di fatto succede che la challenge la carico non solo sul commit ma anche sul messaggio. Estendiamo il tutto per cui prima di generare la challenge, penso che questo sia un hash di tutti i messaggi generati precedentemente. Questo vincola il commit ed eventuali messaggi scambiati in precedenza.

Potrei avere anche alcuni tipi di firme per cui la firma digitale questa autentichi anche se stessa. Immaginiamo sempre un protocollo interattivo che viene esteso. Vogliamo esplicitamente vincolare una determinata esecuzione e per evitare di avere un proof of knowledge che possa anche essere valida per chiave derivate che un attaccante potrebbe andarsi a generare. Gli standard moderni fanno questo vincolo così evito che lo sviluppatore ignaro debba andarsi a preoccupare di questi aspetti. Abbiamo anche delle varianti in cui R viene scelto in maniera random, per cui introduciamo un hash anche per calcolare m . Le firme digitali viste finora sono probabilistiche, ma esistono anche come probabilistiche. Esistono anche standard in uso per tutte queste varianti.

Lo standard più famoso è ECDSA, non fa un binding con la chiave pubblica, è completamente probabilistica, ma ha una versione in cui lo fa. Altri standard sono le firme EdDSA certificate dal NIST. *Slide 65 rappresentazione grafica delle cose dette*

finora. ECDSA concettualmente è simile ma all'epoca è stato fatto leggermente diverso per non andare a violare alcuni copyright.

[19-04-2023]

Ero assente perché stavo facendo lezione ad architettura, ecco gli appunti di snowy:

lecture.txt

Password: kahngu9Eihe4

Ripasso:

Non-Interactive Zero-knowledge proof

Una dottoressa che ha dimostrato che la non-interactive è davvero sicura è stata premiata giusto una settimana fa.

signature algorithms:

Trasformata di ... applicata sull'algoritmo di Zero-knowledge.

Il più famoso ad oggi è EdDSA: dove Ed sta per Edwards

EdDSA:

Ed25519 ha 128 bit di sicurezza

usa SHA-512 come KDF

Variante dove il nonce è calcolato deterministicamente rispetto alla chiave segreta ed il messaggio dato in input.

Problema dei SIV: il messaggio va letto più volte

Qui è uguale, il messaggio viene usato sia nella KDF per il nonce sia nella Fiat-Shamir transformation per calcolare il commitment.

Pure EdDSA: il messaggio è davvero messo tutto dentro ad ognuno dei due digest

Hashed EdDSA: il messaggio viene prima hashato, dopo solo il digest viene messo dentro ai due hash.

Questo si può fare semplicemente nelle firme ma non è possibile nella criptazione

Ma il primo standard NIST sulle firme digitali è DSA/ECDSA

Variante delle firme di Schnorr.

Perché le firme di Schnorr fino al 2004/2005 erano protette.

Variante:

```
k = secret  
K = kB          // public
```

```
sign(k, m):  
    r = random  
    (Rx, Ry) = r*B
```

```

s = r-1(H(m) + k*Rx)
(Rx, s)

verify(y, m, (Rx, s)):
    (Rx', Ry') = (s-1*H(m))B + (s-1*Rx)K
    Rx == Rx'

```

Un pÃ² una porcata, la coordinata diventa uno scalare, non ha dimostrazioni vere e proprie di sicurezza.

PerÃ² non si sono trovati attacchi

Schemi crittografici basati su RSA:

RSA usa non funzioni one-way ma trapdoor one-way
Se conosciamo il segreto riusciamo a fare un reverse.

La volta scorsa abbiamo parlato del perchÃº non usiamo textbook-RSA, per manleabilitÃ .

PKCS1: suite di standard che raccoglie tutti gli schemi basati su RSA

Due versioni: PKCS1-v1.5 e PKCS1-v2
per cifratura e firme.

Cifratura:

PKCS1-v1.5: insicuro rispetto a padding attack.
sicuro se l'attaccante non ha accesso all'oracolo di decifratura.
visto che Ã¨ facile accederci perÃ² Ã¨ da considerarsi deprecato.

PKCS1-v2: versione moderna

a.k.a. OAEP

Va applicato su un testo in chiaro prima di eseguire RSA

Firme digitali:

PKCS1-v1.5: Sconsigliato, ma ancora molto in uso
Non dimostrato formalmente.
PKCS1-v2 (PSS): Consigliato

Deve essere resistente alla manleabilitÃ , ma non ci interessa che sia randomizzato invece.

Criptazione:

PKCS1-v1.5
Schema di padding randomizzato per cifrare i dati in modo simmetrico.

Problema: textbook-rsa Ã¨ vulnerabile a CPA
possiamo criptare i vari messaggi e vedere se sono uguali a quelli man dati, dato che non ha randomizzazione ne IV.

Quindi introduce randomizzazione e delimiter nel padding.

Quindi il padding genera una sequenza di byte random e mette dei delimitatori.

`h'0002' + random bytes + h'00' + padded_message`

Problema: il padding fatto in quel modo Ã¨ intrinsecamente vulnerabile per chÃº ha una struttura forte:

Bleichenbacher's attack

L'attaccante in modo simile al padding attack di CBC, prova un sacco di messaggi fino a che non passano.

Non lo vedremo, ma per quello lo consideriamo insicuro.

N. di query richiesto per ricavare il messaggio in chiaro Ã“ molto superiore all'attacco AES.

Quando Ã“ stato scoperto servivano un milione di passaggi.

PKCS1-v2:

Evitiamo una strutturazione tra padding e messaggio che puÃ² essere sfruttata.

Si introduce una funzione MGF (simile a un KDF)

Non si fa una somma, ma un XOR.

e si aggiunge anche un tag che permette di rifare la funzione anche in decifratura.

Questi sono due schemi di cifratura asimmetrici dove cifriamo direttamente il messaggio,

Ma possiamo costruire anche direttamente uno schema ibrido, dove non cifriamo direttamente il messaggio ma la chiave simmetrica.

scelgo k direttamente tra 0..N-1

cifro direttamente k

Visto che il dato Ã“ direttamente random ed Ã“ da un grande spazio di input, Ã“ sicuro.

RSA-KEM

di solito si trova come RSA-SHA256 o RSA-SHA512

Molto piÃ¹ veloce rispetto a PKCS1

Problema: non si possono cifrare messaggi troppo grandi.

Anche con PKCS1 (v1.5 o v2) il messaggio Ã“ piÃ¹ piccolo rispetto al massimo teorico di textbook-rsa

Il n. di byte che possiamo cifrare Ã“ un parametro che dipende anche dal livello di sicurezza

ex: N=1024 bit -> posso cifrare 62 byte.

N=2048 bit -> posso cifrare 190 byte.

L'overhead di PKCS1 Ã“ quasi costante.

Firme digitali:

Problema:

$\text{Sign}(M1 / M2) = \text{Sign}(M1) / \text{Sign}(M2)$

Per rompere questo dobbiamo usare delle funzioni hash.

La cosa piÃ¹ semplice che si puÃ² fare Ã“ fare la firma dell'hash rispetto alla firma del messaggio diretto.

Le relazioni matematiche rimangono, ma solo a livello di digest.

Non possiamo manipolarle apposta per ottenere l'hash che vogliamo noi senza reversare l'hash.

Allora perchÃ“ non usiamo semplicemente SHA-256?

PerchÃ“ vogliamo una funzione hash che riesce a coprire nella maniera piÃ¹ uniforme possibile il dominio (Z_n^*)

Se genera solo 256 byte, ci sarebbero problemi.

Questi tipi di funzioni si chiamano FDH: Full Domain Hash

Non guarderemo bene com'è fatta questa funzione.
Ma nessuno standard RSA usa questo tipo di funzione, anche se esistono 0_0
Sono dimostrate sicure, esistono, ma nessuno le usa.

PKCS1-v1.5:

Ha un sistema di codifica, non di hashing (ma è uguale)
prova ad essere più similmente possibile un FDH

Covers almost the full domain of RSA (1..N)
Ed è deterministico.

PKCS1-V1.5 È UNO SCHEMA DI FIRME DIGITALE DETERMINISTICA.

PKCS1-v2: PSS (Probabilistic Signature Scheme)
Simile a OAEP, usa lo stesso MGF

Aggiunge un salt di valorizzazione.
Firmando molte volte lo stesso messaggio otteniamo firme digitali differenti.

L'hash usato è standard ora.

Si può costruire con hash standard scelti, solitamente si usa SHA256

Nelle firme PKCS1-v1.5 è ancora usato, ma è deterministico, a volte è un vattaglio e a volte no.

Le firme basate su curve ellittiche e quelle basate su RSA coesistono
A seconda dei contesti in cui le usiamo.

- *DSA hanno firme piccole
Ex: SHA256 (32 byte) + coordinata (32 byte) = 64 byte
In RSA la firma è grande quanto N
128 bit -> 3072 bit (384 byte)
- *DSA La generazione delle firme è più veloce
- *RSA La verifica è molto più veloce (non vedremo perché)

EdDSA ha una forma di verifica ottimizzata per le verifiche batch
Quando vogliamo verificare tanti messaggi.

openssl speed ...
per guardare la velocità

Attack models:

- dobbiamo sempre pensare a:
- A cosa ha accesso l'attaccante?
Cosa sa, a che interfacce può accedere
 - Cosa può ottenere?

Se l'attaccante riesce a generare delle firme digitali si chiama "Forging"

Superficie d'attacco:

- Che conoscenze e a cosa può accedere l'attaccante
- No message
Non ha accesso a nulla, il più debole.
 - Esistono schemi dove questo è possibile?
Sì, ma non schemi sicuri.
 - Anche in RSA si possono creare [(m, s)] randomici
basta scegliere s random e guardare di quale messaggio lui è la firma

a.

Non ho il controllo sul messaggio, ma soddisfa la verifica

- Random message
L'attaccante ha $[(m, \text{sig})]$ per messaggi random
- Known message
L'avversario conosce una lista $[(m, \text{sig})]$ senza sapere quale sarà la chiave pubblica (nchesens??)
didattico, non staremo a fare esempi
- Chosen Message (CMA, Chosen Message Attack)
L'attaccante conosce la chiave pubblica e può richiedere di erogare $[(m, \text{sig})]$ per certi m
Standard si avvicina il più possibile al mondo reale
- Adaptively chosen message
L'attaccante può trovare $[(m, \text{sig})]$ per ogni chiave pubblica.
Un po' strano, è vincolato dalla scelta della chiave pubblica.
Vorremo che lo schema di firma digitale sia sicuro a prescindere dalla chiave pubblica che utilizziamo.
L'avversario ha una scelta di qualsiasi chiave pubblica.
Alcuni standard falliscono questa tipologia di attacco.

L'attaccante vuole riuscire ad inviare a qualcuno una copia (m, sig) e l'altro deve verificarlo con successo.
Noi assumiamo che chi riceve la firma conosce la chiave pubblica, ma magari non lo sa.

Studiamo attacchi più forti rispetto allo standard ma diventa poi un attacco teorico.

Cosa può voler fare un avversario?

- Key recovery:
può forgiare (m, sig) per ogni m
- Selective message forgery:
L'avversario riesce a generare alcune firme particolari
- Existential message forgery
Di per sé non può essere studiato se non lo decliniamo in 2 tipi
Weak (EUF-CMA): noi riusciamo a generare almeno una coppia (m, sig) per un qualsiasi messaggio m mai firmato in precedenza
Consideriamo un avversario CMA, ha già una lista $[(m, \text{sig})]$
Se non riesce a generare un messaggio $[(m, \text{sig})]$ allora vince
Strong (SUF-CMA): Non mette un vincolo su m
L'avversario ha accesso alla lista di messaggi già visti.
L'avversario vince se trova anche un'altra firma, diversa, per un messaggio o dentro alla lista.

Roba strana, difficile capire le implicazioni.

Soltanente la standard è Weak Existential Forgeability.
In alcuni casi particolari gli standard vanno modificati per SUF-CMA.

Su moodle c'è ecDSA1.ipynb
Ha scelto ECDSA e non DSA perché si riescono a fare più cose

```
Usa pycryptodome come dipendenza
python -m venv venv
source venv/bin/activate
pip install pycryptodome

ECDSA: nistp256 (è il nome)
si riescono a ottenere i vari parametri della curva (p, q, G)
```

importa le chiavi da openssl semplicemente
fa vedere come accedere a chiavi pubbliche e private
(e come ricalcolarsi la pubblica dalla privata)

DSS: Digital Signature Standard

Tutti gli standard di firma digitale definiti dal nist
gli passiamo una chiave sefreta e diventa ECDSA
(perchÃ“ gli passiamo una chiave sefreta di P-256)
Gli passiamo anhce una modalitÃ
fips-186-3 Ã“ la modalitÃ originale
Ã“ la modalitÃ standard dove il nonce Ã“ creato ogni volta.

c'Ã“ anche la variante deterministica 'deterministic-rfc6979'
simile a EdDSA, il nonce Ã“ generato in maniera sintetica

Ma se creiamo DSS con la firma, mode Ã“ inutile
perchÃ“ ha solo effetto per la generazione del nonce
La firma se ne frega di come Ã“ generato.

Di default la codifica Ã“ 64 byte, i primi 32 sono r e gli altri 32 sono s
Codifica "raw"
Altri encoding: DER, lo accenneremo.

ECDSA NON Ã“ SUF-CMA (Strong EUF-CMA)

perchÃ“ ogni volta che s genera una firma si puÃ² sempre ricalcolare un'altra firma valida per lo stesso messaggio

sig = (Rx, s)
(Rx, -s) funziona comunque
il punto che generiamo Ã“ l'opposto che genereremmo originalmente.
Ma la firma non include tutto il punto ma solo la x, quindi non c'Ã“ il segno.

Per diventare EUF-CMA dobbiamo introdurre degli standard, tipo concondarsi
di usare sempre il valore assoluto di s (cioÃ“ il piÃ¹ piccolo)

Attacco key recovery from nonce reuse ECDSA

ManleabilitÃ

Possiamo manipolare la chiave per fare cose che non dice perchÃ“ sono le 1
6:02, giochiamoci online

Distribuzioni Chiavi Pubbliche

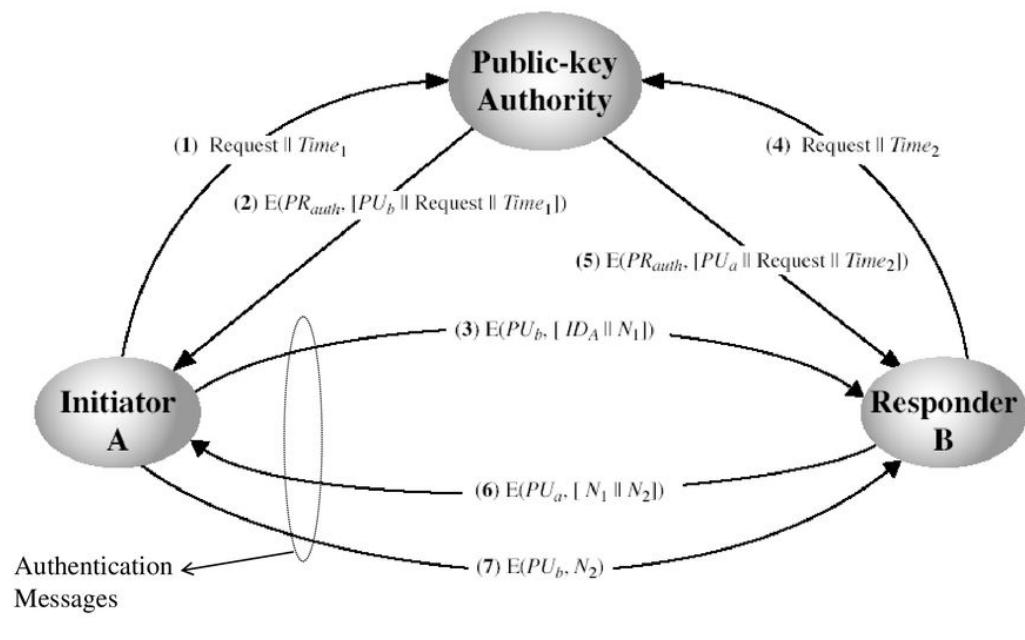
trust anchor come elementi che mi vanno a dare chiavi pubbliche che so essere sicure. Posso andare a identificare 3 tipi macroscopici:

1. TOFU
2. Out-of-band verification

3. Delegated approaches

Il paradigma tofu identifica i protocolli in cui presupponiamo di non essere sotto attacco la prima volta che interagiamo con una entità. Immaginiamo la situazione di una connessione ssh. Praticamente presupponiamo che non stiamo subendo un mit e non dovremmo ricevere una chiave pubblica falsa. Praticamente questo è key distribution su un canale in cui eseguiamo il protocollo stesso. Assumiamo che non ci siano attacchi attivi e che alla fine anche se ci fossero sarebbero passivi. Tofu spesso si mischia ad un approccio out of band, in cui la chiave pubblica la passo su un canale diverso da quello che uso per le comunicazioni. Potrebbe essere una verifica per esempio tramite una telefonata banalmente. Molto utili quando ho un secondo canale utilizzabile. Una situazione potrebbe essere quella per abilitare per esempio un nuovo dispositivo con telegram e whatsapp. Per quel che concerne gli approcci delegati mi trovo in situazioni in cui sto cercando di andare a sfruttare entità terza per occuparmi delle chiavi pubbliche che vado a distribuire. Del tipo: Alice e Bob non si conoscono ma conoscono un terzo, immaginiamo Carl. In questo approccio Alice e Bob delegano Carl nel comunicare loro la chiave con cui andare a comunicare. Carl è una trust anchor di Alice e Bob e tecnicamente il fatto che si fidino si concretizza con la memorizzazione della chiave pubblica di Carl. Questo approccio di avere un ente, che distribuisce le chiavi, possiamo anche definirlo come decentralizzato.

Centralized Distribution - Public-Key Authority PROTOCOL



Distribuire chiavi pubbliche random ovviamente non è sufficiente e sarebbe il caso invece andare a gestire tutto un aspetto legato ai metadati, che siano valori tecnici o meno. Spesso non servono al protocollo di per sé, ma sono molto comodi per alcuni scopi. Dare delle chiavi senza associarle a dati aggiuntivi che diano anche dei limiti temporali di utilizzo, per esempio, è molto rischioso.

In generale troviamo delle attestazioni crittografiche che certificano la bontà di una chiave. Spesso le chiavi sono accompagnate da questi documenti di verifica per una questione di sicurezza.

Se vogliamo andare a specializzarci in ambito web è sicuramente lo standard X509. Nome dello standard che usiamo per verificare dei certificati nell'ambito del web. In questo troviamo nomi del tipo: distinguished name → chi è il proprietario di questo certificato, common name → dominio su cui usare questo certificato, informazioni sull'identità legale. Questo vuol creare un legame forte da un chiave pubblica e distinguished name. In più abbiamo anche altri metadati che abbiamo prima.

Troviamo anche spesso informazioni riguardo lo scopo specifico di un determinato certificato. Concettualmente abbiamo queste informazioni e per dare qualche

informazioni tecnica diciamo che X509 è un concetto astratto che ci dice che cosa possiamo mettere dentro quello che ci inventiamo e come altro standard è ASN.1

Approfondisci argomento standard DER e PEM, in cui abbiamo un contenuto informativo scritto in modi diversi. PEM è la codifica in base 64 di un file DEM. Un file PEM include più blocchi codificati DER con dei tag che aiutano a capire che ci sia effettivamente su questi blocchi.

PEM definisce questo formato, ma un'altra cosa sono le estensioni dei file in cui sono codificati queste informazioni: .key, .pub, .crt, .csr, .crl

Praticamente sono delle estensioni che sono concretamente dei formati pem. Alla fine DEM e PEM sono formati usati per il certificato x509v3, ma anche in altre situazioni per memorizzare informazioni relativi a certificati web.

Altri standard importanti

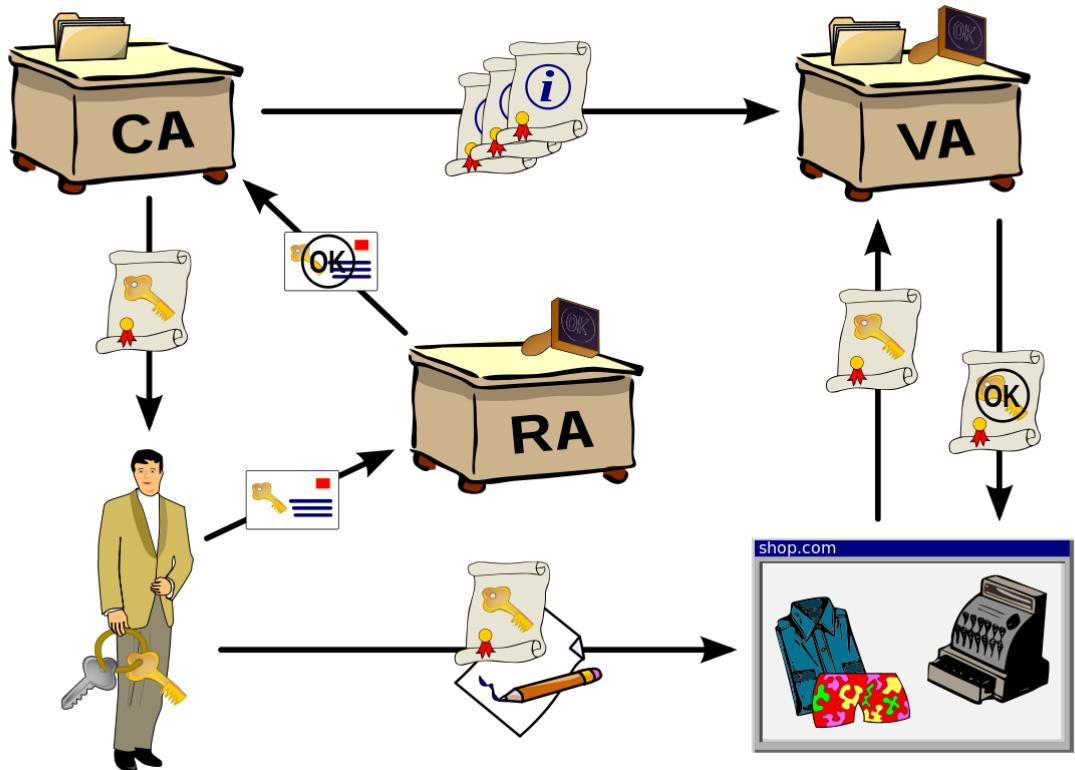
- JOSE → Javascript Object Signing and Encryption, da cui derivano JWE, JWS, JWK
- COSE → CBOR Object Signing and Encryption Framework
- FIDO2-CTAP2 → protocolli per autenticatori hardware e computer. Usato in molti contesti differenti e lo si usa per firmare o autenticare o cifrare dei dati non strutturati.
- FERNET → standard per andare a creare dei JWT praticamente, in contesto cloud.



PKI and Certification Authorities (1)

PKI and Certification Authorities

Se vogliamo immaginare a come passiamo ad una architettura concettualmente delega ad una reale, dobbiamo immaginare che al posto di CARL ci siamo le CA. Le entità che partecipano alla comunicazione sono uno user agent. E un server che possiamo andare a identificare con un certo numero di dominio. L'identità tecnica l'FDN.



Perché uno user agent delegano le CA per autenticare le chiavi pubbliche dei servers? Perché tra i maintainers e le CA abbiamo un accordo per cui ci sono già delle pk raggruppate delle CA di cui i dev del mio user agent (il browser) si fidano. Nel mio caso l'elenco delle CA le sceglie mozilla. Chi ha un server deve ottenere un certificato web firmato dalla CA. Lo standard che vedevamo prima lo possiamo andare a estendere anche qui

In un certo momento i maintainer del software stanno dietro alle chiavi e alle CA, non lo facciamo noi manualmente. Lo otteniamo una volta e poi lo uso tante volte con gli scambi di chiavi.

Il server web non comunica mai con la root ma con le CA intermedie ovviamente. Abbiamo sempre questa situazione a livelli ovviamente. Immagina di avere una catena di certificati dalla base fin verso la root che vengono tutti quanti autenticati.

Chi sono queste root CA?

Le root CA sono delle aziende che devono soddisfare delle policy che non sono identificabile tecnicamente spesso. Abbiamo tutta una questione di policy. Non devo avere affiliazioni con attori che portano a conflitti di interesse, che non consentono come vengono gestiti internamente i dati, abbiamo una serie di policy di governance in cui abbiamo tutto un insieme di regole apposta. Se vogliamo maggiore

informazioni ogni maintainer famoso di browser ha delle sue policy. In teoria ci sono delle policy di internet ma queste possono andare a differire. Esistono anche dei log ad hoc in cui si discutono quali sono le root CA che potranno essere ammesse. Per diventare una root CA bisogna entrare nella comunità, tra le varie cose bisogna proporsi in dei blog appositi in cui ci si presenta e la comunità può investigare se ci si può fidare di noi come CA.



Ricorda che le pk + certificato sono i dati che praticamente otteniamo dalla nostra CA intermedia

Definisco che una root ca è fidata grazie ad uno schema di firme incrociate, praticamente firmandosi da una o da due root ca differenti.

Possiamo usare *openssl* per andare a giocare con i certificati. Attualmente per ragioni di sicurezza i certificati non hanno una vita particolarmente lunga ma si esauriscono prima. Inoltre abbiamo anche la possibilità di andare a revocare i certificati.

Come ottenere un certificato

1. Creiamo un paio di chiavi valido
2. Generiamo un CSR
3. Firma il CSR alla CA (\$\$)
4. Salva la secret key sul web server
 - a. very strict permissions → only root read access
 - b. best if stored on hardware security module (HSM)

Questa struttura fa in modo che la CA che chi fa la richiesta sappia la chiave segreta corrispondente. Quando la CA rilascia il certificato stabilisce anche due END POINT. OCSP è un protocollo che serve per andare a consultare la lista dei certificati revocati. Questo protocollo è basato su http e non su https, perché è un protocollo che serve a https, che usiamo se un certificato di https è stato revocato, se fosse sotto https sarebbe una interdipendenza che non si risolverebbe. Gestiamo la sicurezza a livello di payload. La risposta è firmata digitalmente e ha un timestamp e ha durata limitata.

La revoca si gestisce in questo modo e cosa importante: a livello tecnico il client ottiene il certificato e può vedere se è stato certificato vedendo l'intero file delle

revoche oppure con oscp oppure posso configurare un web server con funzionalità ocsp stapling con una risposta ocsp valida se voglio andare a diminuire la latenza.

[27-04-2023] *Li appunti di oggi li devo andare ad approfondire un pochino, perché al momento sono superficiali su alcuni aspetti*

Ci sono dei paradigmi per fare delle revoche immediate. Paridgma Push e Pull, due modi per andare a gestire i certificati. L'idea comunque è sempre quella di mantenere alta la sicurezza.

Una CA potrebbe andare a chiedere che noi siamo proprietari del dominio di cui stiamo chiedendo un certificati → Domain Validatedkit

Limiti di PKI

Sicuramente il sistema di revoca, poi sicuramente la fiducia rispetto alle CA. PKI funziona ma abbiamo degli attori fidati fondamentali di cui ci fidiamo, ma non si comportano sempre benissimo perché non fanno quello che dovrebbero fare (soprattutto in alcune nazioni). La questione è che le CA potrebbero andare a rilasciare dei certificati per dei siti che sono fasulli e si possono andare a fare dei veri MIT di stato. Si cerca di mettere una pezza a questo tipo di problemi: andare su CA che sono quante più alte possibili. Un'idea sarebbe anche quella di avere delle root CA private che erogano certificati solamente per servizi che mi interessano e non possono generare altri certificati. Praticamente sto riducendo lo scoping di una CA. Questo implica sicuramente dei rischi comunque la si guardi la situazione → devo avere le conoscenze per andare a gestire la CA privata. A livello tecnico le CA pubbliche offrono un servizio per cui se paghiamo agiamo come se fossero CA privata e non hai nemmeno lo svantaggio di dover gestire in casa una CA per mantenere le chiavi in un certo modo. Non uso la CA pubbliche, uso la privata, ma in off source. E' un compromesso accettabile di solito. Ci sono dei servizi online per fare tutto questo in maniera molto comoda. Un altro approccio è fare un fallback: abbandono completamente PKI. Esempio: Google dice che attori governativi hanno cercato di andare a distribuire dei certificati falsi, ma sono molto potente in quanto Google, quindi perché usare PKI? Perché non andare a installare dei certificati importanti all'interno dello stesso software → certificate pinning, meccanismo per cui io dico allo user agent che certificato usare per connettermi ad un certo sito. Questo lo troviamo anche all'interno dei servizi privati con i certificati self signed, in cui il server si auto firma il certificato (campo issued = campo subject). In questo modo il browser non controllo il tutto, ma solamente che sia quello che si aspetta.

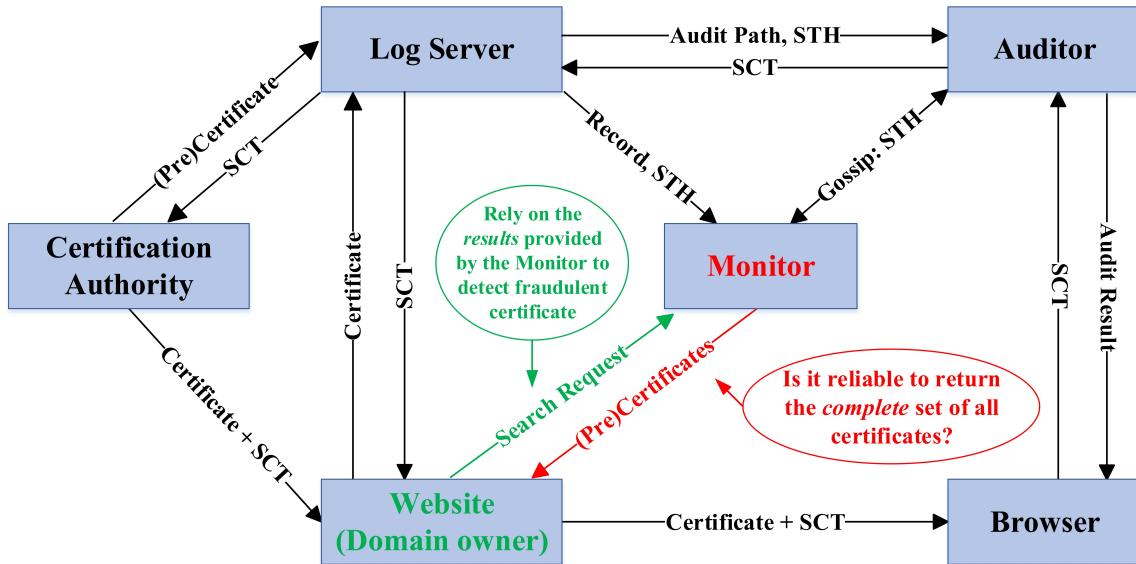
Certificate Transparency

Da quando Google è stata attaccata ha cercato di sviluppare un sistema alternativo per andare a monitorare le CA, praticamente sto inserendo un deterrente in scenari in cui ho delle autorità fidate. Tutto è basato su un sistema di log pubblici che vanno a equalizzare tutti i certificati rilasciati dalle CA pubbliche. L'idea è cercare di beccare in meno tempo quali certificati sono fasulli e quali sono effettivamente rilasciati per i miei servizi. Magari approfondisci un pochino il tema, perché potrebbe essere interessante. Se questi servizi di log si comportano male che succede?

Questi servizi sono implementati con strutture dati autenticate! Queste permette di avere prove di appartenenza o meno di un dato all'interno di una struttura. Questi per policy devono pubblicare la root del MKH3 sui dati loggati, quindi se qualcuno fa una query per un dato su un certo dominio o viene restituito un subset allora siamo in grado di verificare rispetto alla root se la risposta è effettivamente vera o falsa.

Con CT introduciamo un vincolo ulteriore a livello di client: Sappiamo che abbiamo introdotto queste cose per fare comportare bene determinati attori, ma in teoria chi è che inserisce i certificati nel servizio di log? Questi vengono inseriti dalle CA o dai servers. Quindi ci sono diversi flaws. Una modalità per includere i certificati nei servizi di log. Chi effettivamente comanda che servizi di log siano parte di PKI sono i client. Sono i client che controllano i log diciamo. Google dal 2014 fanno un enforcement da questo punto di vista, partendo dai certificati EV con la richiesta di un SCT (certificate transparency). Ho comunque delle parti che si possono comportare male, ma ho un monitoraggio delle autorità praticamente. Questa è comunque tutto in evoluzione perché questi servizi sono mantenuti dalle CA stesse

💡 Si cerca però di introdurre mitigazioni con controlli incrociati praticamente per cercare di ovviare a questo tipo di situazioni.



Sta parlando del funzionamento di openssl

[10-05-2023]

laboratorio PKI, grazie snowy

Untitled

Mi scuso in anticipo per chi legge, dovrÃ² uscire un attimo prima.

Proviamo a fare l'esercizio di PKI

openssl ecparam -list_curves

Fa vedere tutte le curve utilizzabili.

La p256 Ã“ chiamata prime256v1 (NON secp256k1, Ã“ una curva di un altro tipo, sono definiti su campi interi diversi)

I nomi di openssl sono un disastro, il prof consiglia di crearsi un cheatsheet

Nel caso in cui non ci sono le curve che cercate, dovete ricompilare OpenSSL

openssl genpkey -algorithm

PuÃ² generare chiamvi per ECDSA, RSA, EdDSA

(altrimenti c'Ã“ genrsa, ma funziona solo per RSA)

openssl genpkey -algorithm ec -pkeyopt ec_paramgen_curve:prime256v1 -out ca-root.key

(si possono anche generare chiavi RSA con + di 2 numeri primi, per contesti molto strani)

Il file Ã“ un PEM, dentro c'Ã“ il DER in base64.

Solitamente i file .key vengono messi con permessi ristretti (Ex. 400, sola lettura per il solo owner)

Inoltre il file dovrebbe essere solamente dell'utente che puÃ² leggerlo, magari il programma che lo usa.

Con l'opzione -aes{128,256} messa alla fine si pÃ¹Ã² criptare il file generato con un passphrase.

Passphrase = una password creata con diverse parole, perchÃº cosÃ¬ ha un entropia molto alta.

Solitamente perÃ² Ã“ usata come sinonimo di password.

(xkcd comic: password strength staple <https://xkcd.com/936/>)

Ora nel tag scrive che Ã“ una password encrypted (BEGIN ENCRYPTED PRIVATE KEY)

Solitamente si usano le encrypted con le chiavi che devono essere usate da degli umani che possono metterci dentro la password.

Ex: ca-root, Ã“ un amministratore a firmare.

La maggior parte delle volte perÃ² le chiavi sono usate da macchine, quindi non Ã“ necessaria la passphrase

Non vuoi che quando si riavvii il server ti chieda la password, dovreb

```

be prendere la chiave da solo senza un umano.
(a meno di casi molto strani in cui vuoi un intervento umano quando ri
avvii il server)

(ferretti fa ls nel suo .ssh/custom_keys, ha tantissime chiavi).

ssh si rifiuta di usare chiavi private se non hanno 400 come permessi

Nella esercitazione usiamo chiavi RSA, ma noi possiamo usare quella che voglia
mo

openssl riesce ad inferire il tipo di chiave che usiamo
Attenzione che nel mondo reale ci sono problemi di incompatibilità .
Ex: EdDSA sono molto recenti, non è detto che ci siano dovunque

openssl pkey -in private.key -pubout -out public.pub
Generare una chiave pubblica da chiave privata

La chiave pubblica può essere divulgata, infatti non è possibile
le cifrarla.
Chiave RSA > Chiave EC > Chiave Ed.

Nelle slide genera la chiave privata e mette i permessi corretti (non la cifra per
comodità ma dovrebbe essere fatto

openssl req -new -x509
Fai una richiesta (req) e firmala da sola (-x509)

Solitamente le root-ca hanno certificati lunghissimi, perché vanno direttamente
sui client.
Noi creiamo una chiave che dura 10 anni

-extentions v3_ca_root -> Deve coincidere con una sezione del file di configura-
zione
Perché ca-root.conf ha un formato .ini
E ci sono diverse sezioni
    v3_ca_root è una di queste

con extensions <a> andiamo a dire vai a leggerti anche le cose nella sezione <a>

Dentro questo c'è il KeyUsage
    - digitalSignature
    - CRLSign (può firmare liste di certificati revocati),
    - keyCertSign (può firmare certificati)

E anche il scope CA:true
basicConstraints = critical, CA:true

Per il ca_root non metto nessun vincolo, ma quando genererò l'intermedia
dobbiamo mettere pathlen:0
così non può creare altre CA intermedie.

critical?
Tutte le parole chiavi che usiamo sono definite in x509
critical concettualmente va ad identificare quei vincoli che DEVONO essere
verificati da chi va a leggere il certificato in maniera rigorosa.
Possono anche esserci scope raccomandati ma non sono da controllare rigorosamente.

```

Di solito per^{Ã²} se ci sono degli scope sono critical.

x509 ha una lunga storia e ha molte opzioni di nicchia poco supportate.

v3_name_constraint

Puoi erogare domini solo con un certo vincolo.

È presente da molti anni nello standard x509 ma è stato implementato in standard moderni dal 2016.

Chiede di mettere il Distinguished Name

Common Name: richiede di mettere solitamente il nome di dominio, ma stiamo facendo una CA

Con una CA basta che ricordi a cosa serve il certificato

Email Address -> mail dell'amministratore.

Dato che sto generando una gerarchia per un'organizzazione, impongo come vincolo che tutti i certificati devono stare sotto lo stesso ORganization Name.

Vincolo che impongo nella fase di rilascio dei certificati.

Il file CRT ha anche la chiave pubblica

Come si guarda?

openssl x509 -in ca-root.crt -noout -text

noout: non voglio salvarlo su file

text: leggi il PEM e trasformalo come human-readable

Issuer = Subject (quindi è self-signed)

openssl stampa il nome della curva sia con il nome ASN1 sia col nome NIST scrive anche che firmerà con ECDSA con SHA256

Fin'ora eravamo l'amministratore della root-ca

Ora emuliamo di essere il computer dell'amministratore della signing ca (sign-ca) (altro amministratore, altra azienda)

cd ../../ca-sign

openssl genrsa -out private/ca-sign.key 4096

Sperimentiamo e creiamo una chiave RSA

Stesso tag, molto più grossa.

chmod 400 private/ca-sign.key

Ora dobbiamo creare un Certificate Send Request, dovrebbe essere inviato all'amministratore della root-ca

openssl req -new -config ca-sign.conf -key private/ca-sign.key -out ca-sign.csr
stesso di prima ma senza -x509, non self-firmato

La Organization Name deve essere la stessa!! (vincolo imposto da noi, non è obbligatorio)

Abbiamo creato un csr :3

openssl req -in ca-sign.csr -noout -text

Per visualizzarlo (se siamo root)

Ora torniamo root e firmiamo

cd ../../ca-root

openssl ca -config ca-root.conf -in ../../ca-sign/ca-sign.csr -out ../../ca-sign/ca-sign.cer

```

n.crt -extensions v3_ca_sign
    Ci fa vedere tutti i dettagli
    Quando lo firmiamo lo va mettere in un "database"
    cat index.txt -> da tutti i certificati rilasciati
        V -> Valido (altrimenti sarebbe R: revocato)
    cat serial -> da il prossimo identificativo da usare

cd ../servers

Al server serve semplicemente una configurazione, si rigenera delle chiavi.

openssl genrsa -out www.unimore.it.key 2048
    Di prassi i server hanno i file delle chiavi che sono uguali al dominio!

Se guardassimo il file di configurazione potremmo vedere che va a leggere dei parametri dalle variabili di ambiente
    SAN e OCSP
    cosÃ¬ a prescindere da cosa vogliamo generare, basta modificare le variabili d'ambiente.
    Dal momento che erogo un certificato server devo mettere un endpoint OCSP
        Ci inventiamo che l'endpoint OCSP sarÃ  ospitato su ocsp.unimore.it
        SAN: il certificato sarÃ  valido anche per altri domini, non solo per www.unimore.it.

RICORDARE: Nel SAN deve essere contenuto anche il Common Name.
Se non si mettono gli spazi tra le DNS non va niente, che bello

openssl req -new -config server.conf -key www.unimore.it.key -out www.unimore.it.csr
    Common Name: dovrebbe essere uno compreso nei SAN
        Deve essere un nome di dominio valido, non puÃ² essere qualcosa a caso.

cd ../ca-sign
openssl ca -config ca-sign.conf -in ../servers/www.unimore.it.csr -out ../servers/www.unimore.it.crt -extensions server_cert

facciamo fare ad openssl il server (sia il client).
    MA la connessione fallisce, "Unable to verify the first certificate"
    vuol dire che il client non riesce a ricondurre il certificato ad una root-ca nota.
    openssl di default usa i certificati root del sistema operativo.

openssl s_client -brief -CAfile ca-root/root-ca.crt -connect 127.0.0.1:4433
    Qui non abbiamo fatto un self-signed per il server finale, ma in quel caso dovremmo usare come -CAfile il certificato del server.

Ma la verifica fallisce comunque, perchÃº??
    Il client deve ricondurre il certificato che il server gli manda, al certificato root che abbiamo passato al client stesso.
        ma il certificato non Ã stato firmato direttamente dal root-ca ma da una intermedia.
    Quindi il server deve dare la catena intera dei certificati!

Catena di certificati? concatenazione dei file pem

cat ca-root/ca-root.crt ca-sign/ca-sign.crt > fullchain.crt
    In un certificato ha salvato il file PEM e anche il testo, chissene, dovrebbe andare comunque

```

il server distribuirà comunque le informazioni e il client le scarterà .
il prof le scarta a manazza.

L'ordine è importante?

Solitamente no, per esperienza del prof, ma non è detto.

Alcuni client potrevvero usare l'ordine, ma non dovrebbero

```
openssl x509 -in fullchain.crt -noout -text  
mostra solo il primo certificato, non tutti.  
Il prof splitta il file e poi usa il comando sui file spartiti.  
Non conosce un comando specifico.
```

A quanto pare il server invia il PEM testuale, non binario, ma il prof non è sicuro.

I certificati x509 non sono progettati per essere efficienti a livello di standard, ci sono altri standard.

CTLS -> compact tls, un draft per inviare dati in maniera più compressa.

In questo contesto probabilmente non li invierà in formato PEM perché sono binari.

In alcuni contesti, dato che si sa che il root è già noto, si usano due diverse chain:

- chain: la catena con solo gli intermedi
- fullchain: la catena completa con anche la root

```
openssl s_server -CAfile fullchain.crt -cert servers/www.unimore.it.crt -key servers/www.unimore.it.key -port 4433
```

simuliamo anche il client come prima (-brief = meno informazioni)

Finalmente, Verification: OK

Con brief otterremmo tutto il certificato con più informazioni, ma chissene.

Ok quindi? no.

I certificati sono erogati per specifici server.

OpenSSL è un tool di debugging

Di default non ha gli strumenti per capire cosa deve verificare.

openssl non sa che dominio dovrebbe verificare, quindi accetta tutto.

```
-verify_hostname www.unimore.it  
Verified peername: www.unimore.it  
L'ha verificato!!  
e se cambiamo il verify_hostname ci da errore.
```

SAN IP:

Prima abbiamo specificato SAN validi apposta per valori DNS, ma se specifichiamo IP possiamo creare certificati validi per certi indirizzi IP (molto utile nei contesti privati).

Su un browser dovremmo andarci ad installare il certificato root sul browser.

Firefox -> Certificate Manager -> Import

Possiamo usare anche x509 per il protocollo MIME (verificare le mail)

Firefox si è buggato, ma sembra andare

```
openssl s_server ... -WWW  
Serve contenuti statici con HTTP SSL
```

```

Per farlo andare bene:
ascoltare sulla 443
andare a modificare /etc/hosts per mettere www.unimore.it

Ora c'Ã" un errore legato al webserver perchÃ" non trova la risorsa, perÃ² funziona!!
Siamo connessi a www.unimore.it! yayy

Non farlo usando google.com su chrome, perchÃ" lui se ne frega.

Osservazioni:
OCSP non Ã" obbligatorio, il browser effettua una richiesta per il OCSP, non riceve risposta e lo considera comunque fidato.
Anche se il servizio non risponde accetta il certificato come valido.

Creiamo un nuovo certificato server per il server OCSP
Common Name: ocsp.unimore.it

openssl ca -config ca-sign.conf -extensions server_cert -extensions ocsp -in
../servers/ocsp.unimore.it.csr -out ../servers/ocsp.unimore.it.crt
Il certificato viene messo anche nel database.

openssl puÃ² anche essere usato per emulare ocsp
openssl ocsp -port 80 -text -index ca-sign/index.txt -CA fullchain.crt -rkey serve
rs/ocsp.unimore.it.key -...
Attenzione: i server ocsp devono essere sempre sulla porta 80!
index.txt -> database dei certificati validi
fullchain.crt -> perchÃ" anche ocsp firma
-rsigner servers/ocsp.unimore.it.crt

Ora abbiamo il server TLS sulla 443 e OCSP sulla 80

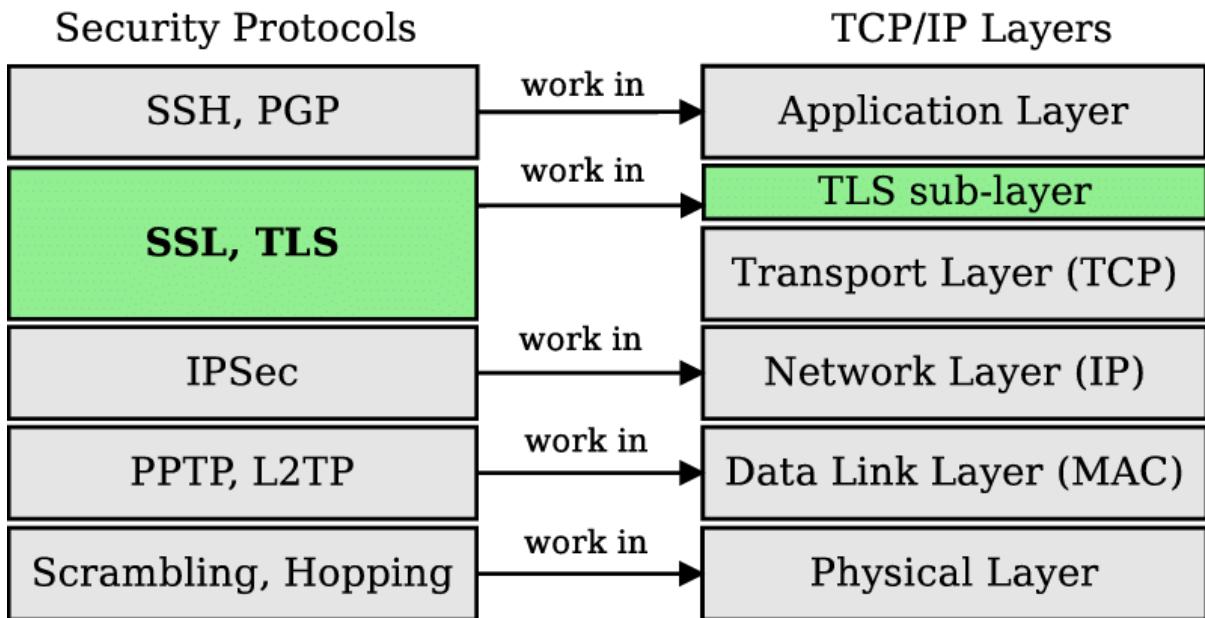
con wireshark ci mettiamo in ascolto su loopback e ascoltiamo solo ip.addr == 127.
0.0.1
Firefox si bugga, togliamo e rimettiamo i certificati

Devo andare, non sapremo mai se la demo va a buon fine o meno

```

[11-05-2023]

A seconda del contesto gli attacchi si complicano. Pensiamo a scenari in cui abbiamo anche comunicazioni di gruppo magari. Si utilizza un termine chiamato object security per identificare un oggetto di alto livello che vogliamo andare a proteggere → end to end security o encryption. Modo per proteggere la comunicazione tra client e server. Abbiamo sempre e comunque dei trade off tra confidenzialitÃ e performance.



In base al tipo di sicurezza che vogliamo andare ad implementare ci fermiamo su un determinato livello.

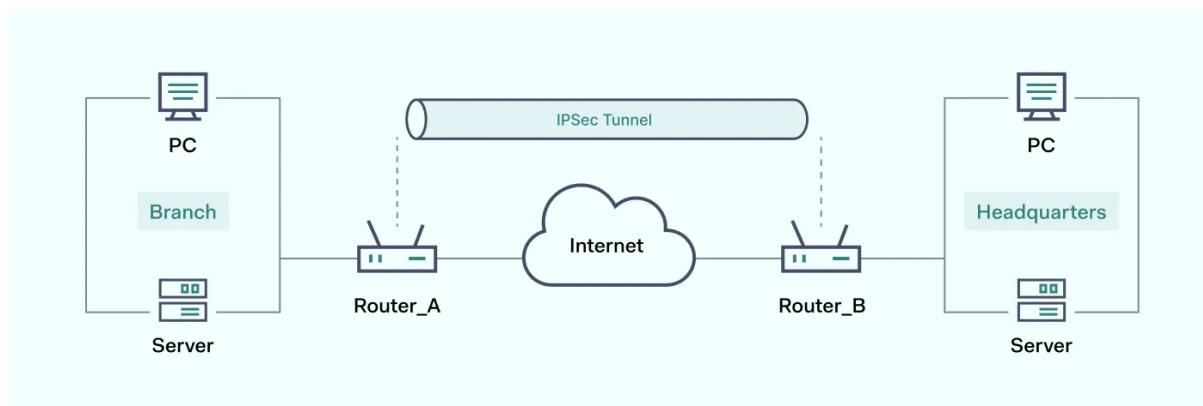
MACSec

Cerca di andare a estendere l'header di ethernet per aggiungere informazioni di autenticazione obbligatoria e in modo opzionale di confidenzialità. Se vogliamo andare ad implementare un macsec dobbiamo introdurre misure di controllo locale in modo forte. Si immagini lo scenario di uno switch di rete che funziona in maniera *whitelist* l'unica cosa è che posso forgiare a mio piacimento il mac address una volta capito che mac address sono nella lista. MACSec invece cerca di introdurre una autenticazione a livello di switch, per cui ogni pacchetto viene controllato e verifica che ci sia una informazione di autenticazione forte. Il protocollo può essere integrato anche con altri tipi di supporti: chiave statica, negoziata con credenziali, ... il protocollo di base definisce come è fatto il pacchetto esteso, ma poi il resto te lo puoi configurare tu. Questo tipo di protocollo cerca di implementare misure di sicurezza che esistono ora solamente per il wifi anche per cavi fisici. Per la crittografia del dato me ne occupo a livello superiore, farlo a quel livello sarebbe molto costoso.

IPSec

A livello 3 abbiamo ipsec. Molto famoso il software StrongSwan per andarlo ad implementare. Abbiamo un campo ad hoc per poter abilitare questa opzione. E' stato proprio formalmente accettato. Abbiamo la comunicazione protetta a livello di host. Funziona in due modalità:

1. Modalità Trasporto che prevede il fatto di avere host A e host B e si voglia proteggere la comunicazione tra due host
2. Tunnel: modalità che prevede una cifratura e autenticità, ma immaginiamo di avere due gateway di due reti. Praticamente sto facendo un ponte cifrato tra le due reti e non vedono la comunicazione passa attraverso infrastrutture esterne alla azienda.



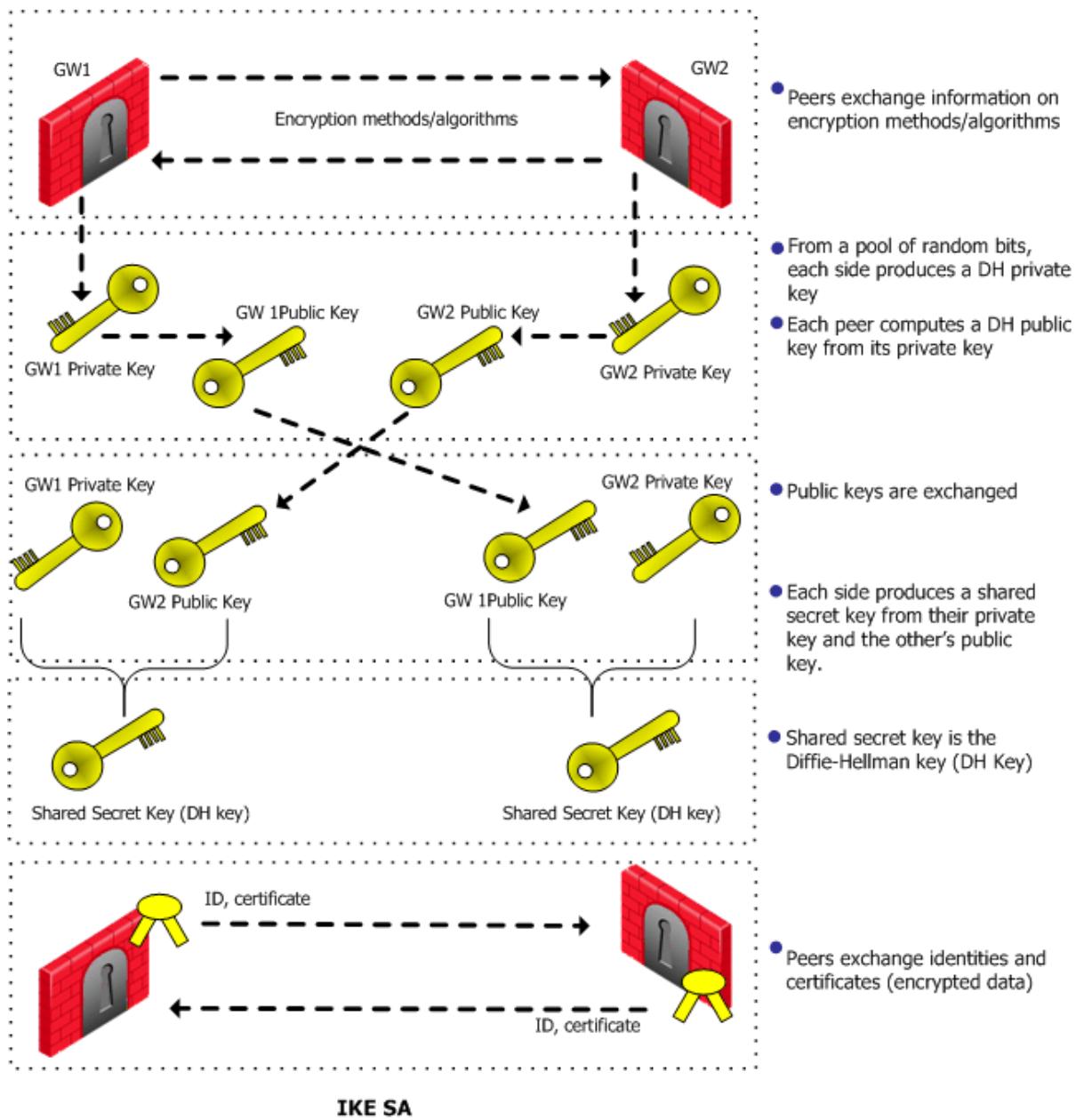
Esiste una vpn unimore: vpn.unimore.it (ferretti usa vpn.unimore.it:443)
Comodo per andare ad accedere da remoto alla rete universitaria per poter accedere a determinati servizi.

Praticamente la vpn mi funge da router verso una rete che non è quella a cui attualmente sono connesso.

IKE

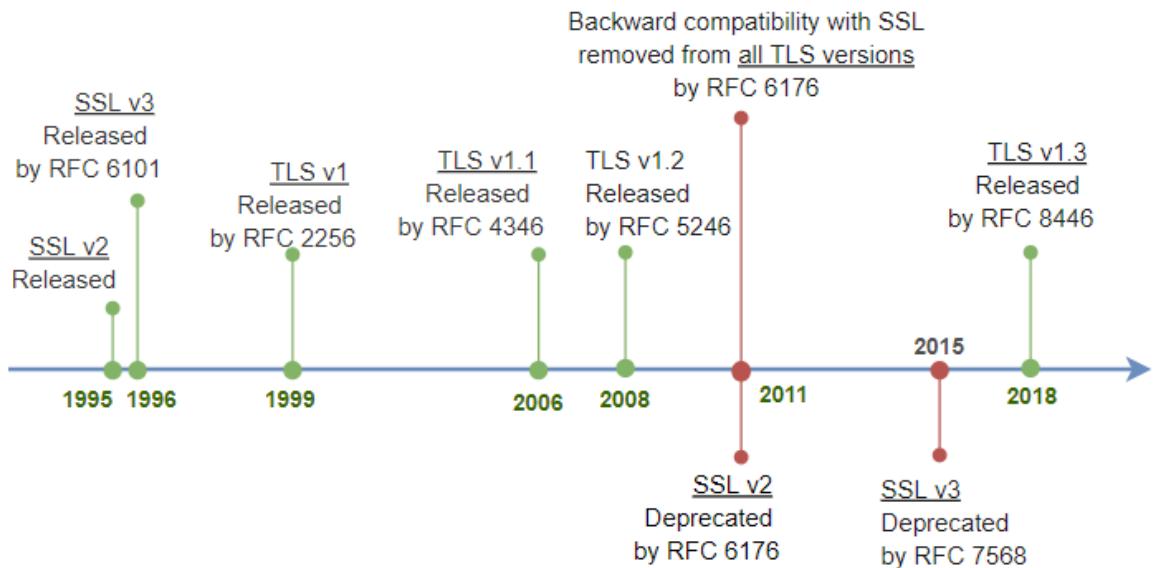
Internet Key Exchange → protocollo che serve in modo specializzato per gestire le chiavi. Come in macsec anche in IPSec non abbiamo una logica di gestione delle chiavi.

IKE Phase I for Security Gateways

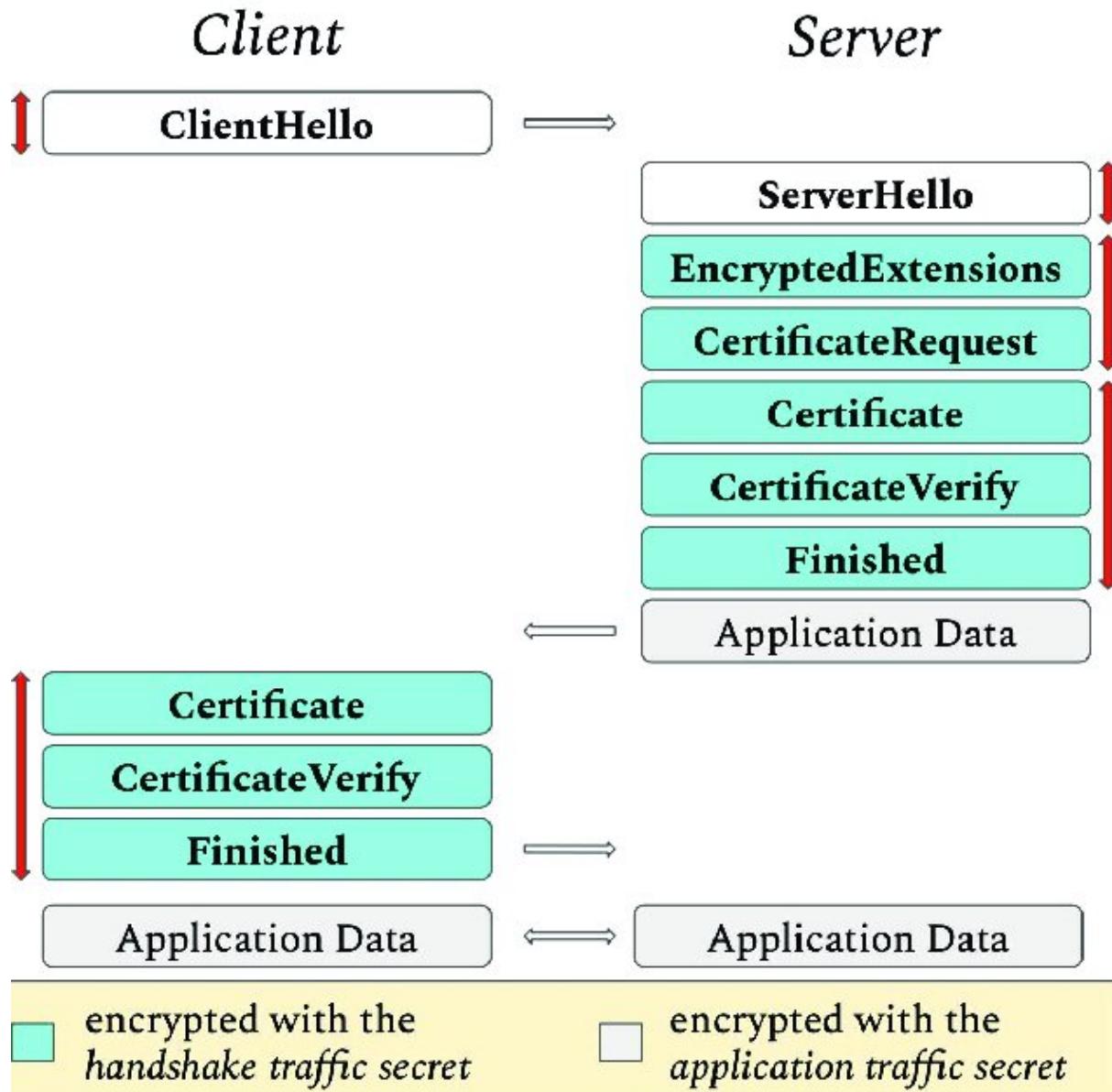


TLS

Evoluzione di SSL che oggi è deprecato.



Le versioni sicure di TLS sono la 1.2 e la 1.3, la 1.0 e 1.1 sono definite come insicure dai nostri browser. TLS lo applichiamo su TCP e ne abbiamo una variante DTLS che mettiamo su UDP. Nonostante sia basato su UDP e sia data oriented, deve comunque andare a implementare un handshake che normalmente non ci sarebbe in UDP. Interessante come DTLS abbia un supporto alla reply detection.

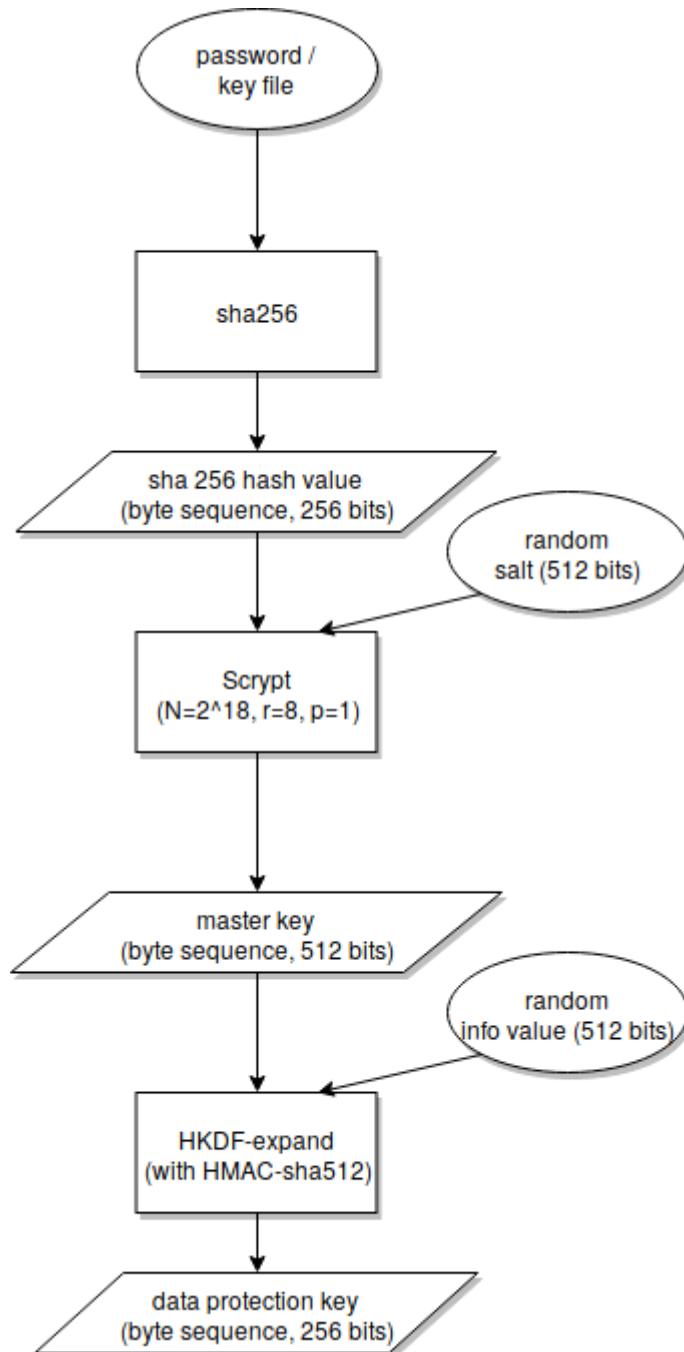


Nelle slides lo schema è più chiaro rispetto a questo che fa vedere come funziona a grandi linee TLS 1.3

Interessante vedere come durante la fase di negoziazione ci sia la scelta del protocollo di comunicazione da andare effettivamente ad utilizzare. La sequenza sarebbe questa:

1. Establish security capabilities
2. Server authentication and key exchange
3. Client Authentication
4. Finish

La negoziazione è sicuramente la fase più difficile da andare a implementare in maniera sicura. Proprio perché è stata difficile a progettarla bene in modo che fosse inattaccabile. TLS 1.3 usa in maniera esplicita HKDF per fare expand per alcune chiavi (?) Devo andare ad approfondire la questione.



Le HKDF posso essere usati quanti per fare key derivation, ma anche per fare dei messaggi finished = HMAC(pre_master_secret, handshake, handshake_messages), praticamente immaginiamo che il payload che mettiamo sia la combinazione di tutti i messaggi mandati fino a quel momento, prima di questo messaggio finished. E'

importante che i due finished mandati siano identici, in questo modo andiamo ad autenticare tutta quanta la comunicazione fino a quel momento. Verifichiamo di aver avuto la stessa visione del protocollo di autenticazione. Se avviene un MIT allora questo finished salta, perché praticamente andiamo ad avere due visioni diverse probabilmente. L'assunzione è che l'attaccante non riesce ad andare a calcolare la *pre_master_secret*.

Immaginiamo che uno standard tls supporti un cifrario insicuro, come il DES per la cifratura asimmetrica. E' interessante vedere come possiamo non deprecare direttamente lo standard ma diciamo a client e server di non andarlo a supportare più. Non dobbiamo deprecare tutto tls, ma solamente il cifrario. L'handshake, se è sicuro, un attaccante non ci può convincere ad usare un cifrario che ormai è formalmente deprecato, perchè se client e server sono configurati in maniera opportuna possono andare a rilevare correttamente questo tipo di attacco. Rimane che se ho delle vulnerabilità in handshake o finished, devo reputare il protocollo come insicuro → la storia delle evoluzioni di TLS. Praticamente questi attacchi sono definiti come *downgrades attack*. Questi sono più complessi da andare a risolvere.

Il downgrade attack quando viene rilevato? Il client e il server hello non sono autenticato e quindi all'inizio abbiamo un supporto di cifrari più deboli, MA se abbiamo un protocollo sicuro l'avversario non riesce a falsificare i finished e quindi alla fine salterebbe tutto. Praticamente facciamo un check alla fine. Potrebbe comunque esistere un MIT che si finge un client malevolo, ma non dovrebbe andare a creare danni.

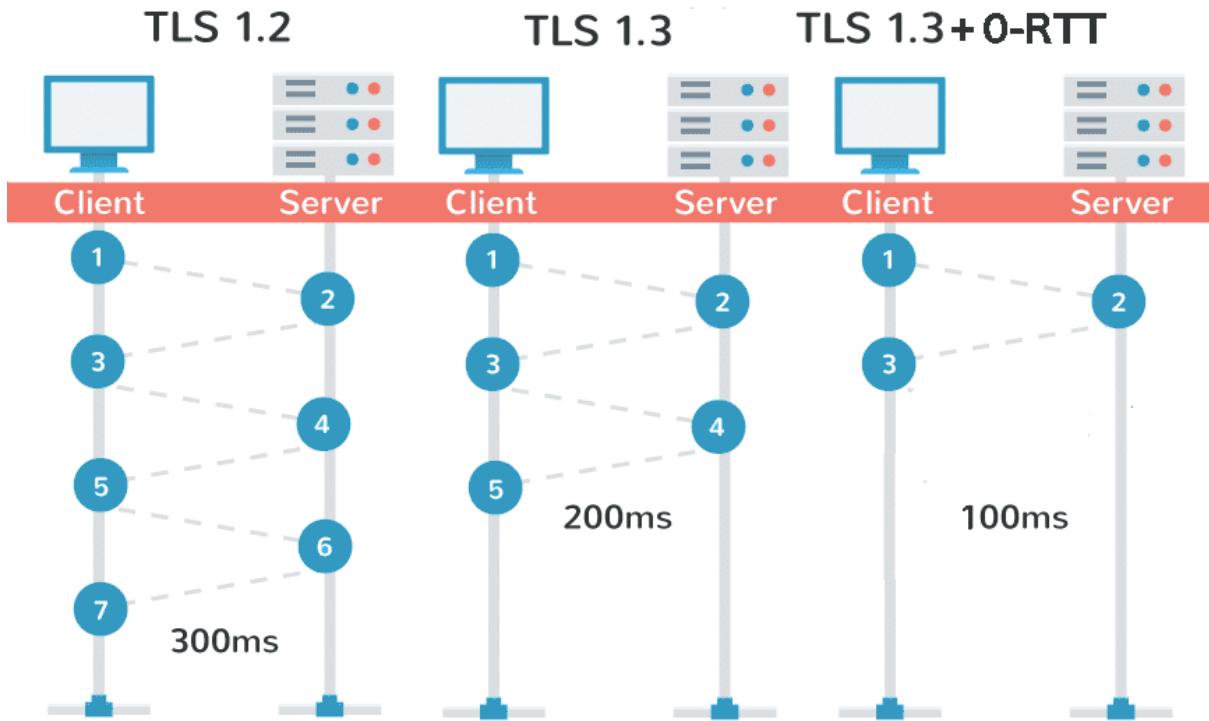
Una parte interessante è quella delle chiavi effimere: una chiave effimera è un tipo di chiave ottenuto con una procedura tale → perfect forward secrecy. Praticamente siamo legati ai concetti di forward security. La così detta chiave effimera è legata al fatto che in questo key exchange... la parte interessante è capire per cosa viene utilizzata questa chiave pubblica. Se noi in uno scambio di chiavi ha l'obiettivo di generare chiavi effimere, allora queste chiavi sono usate per garanzie di autenticità, ma non vengono usate per obiettivi di confidenzialità. Il client genera una chiave e la cifra usando la chiave pubblica contenuta nel certificato del server. La chiave che scambiamo però non garantisce forward security → un avversario nel futuro, essendo usata per decifrare le chiavi simmetriche usate nel passato gli stiamo permettendo di decifrare messaggi nel passato, accedendo alle chiavi usate in passato. Magari l'attaccante sta facendo il dump di tutto il traffico cifrato e solamente

dopo n mesi è riuscito a fregarci la chiave. Questo è in grado di decifrare tutto quello che è stato scritto fino a quel momento che magari l'attaccante ha rubato? Un attacco del futuro mi compromette i dati nel passato? Se uso una chiave a lungo termine non posso andare a garantire forward secrecy

Uso queste chiavi a long term solamente per l'autenticità, perché mi interessa solamente per coloro che devono verificare l'autenticità del dato. Devo andare a generare delle chiavi effimere di norma per andare a gestire la confidenzialità dei dati. Diffie Hellman viene definito come effimero (ECDHE) quando client e server generano i contributi pubblici che sono nuovi ogni volta a run time. E' la fase dello scambio di chiavi che deve essere effimero. Potrei realizzare lo scambio di chiavi effimero anche con RSA. Noi non usiamo le public key inserite nei certificati, mi raccomando. Abbiamo anche altri dettagli tecnici. All'interno di una sessione di scambio di dati che usa la stessa chiave simmetrica, posso usare anche delle tecniche di key rotation, cancellando la chiave simmetrica precedente. Questo in modo di avere chiavi che cambiano anche all'interno di una stessa sessione, in modo che se un attaccante mi scopre l'ultima chiave di una sessione sono ancora protetto all'interno della stessa sessione.

Nel client hello posso anche andare a mandare un session ID e serve per andare a identificare una chiave simmetrica che è stata definita in passato.

Al momento TLS 1.2 e 1.3 vivranno in parallelo, questo perché 1.3 è diverso e pensato diversamente. 1.3 per altro funziona solamente su scambio di chiavi basati su Diffie Hellman. La parte fondamentale di TLS 1.3 è che cerca di migliorare le performance e sia a tre scambi invece che a 4. C'è una modalità aggiuntiva che si chiama 0-RTT, in cui praticamente andiamo ad evitare l'hand shake. Se c'è già stato in passato il client può mandare un payload direttamente e non abbiamo alcun tipo di risposta dal server. Un singolo messaggio. Questa cosa può essere attaccata da reply attack ovviamente.



Attività di default solamente per certi sottoinsiemi di determinati contesti applicativi.
L'idea è quando non provochiamo problemi di sicurezza → se usassimo http e basta ci starebbe per delle richieste di sola lettura. In tutte le operazioni **idempotenti**, che non modificano lo stato del server o della logica applicativa. Sicuramente le *get* se è stata sviluppata bene.

[18-05-2023]

Presentazione dei progetti disponibili. Praticamente lo prenoti, hai una settimana di recesso e hai 1 mese per andarlo a preparare. Possiamo anche andare a proporre argomenti extra. Se fai proof of concept magari lavora con delle slides direttamente per andare a spiegare meglio il tutto.

Protocollo di Singlas

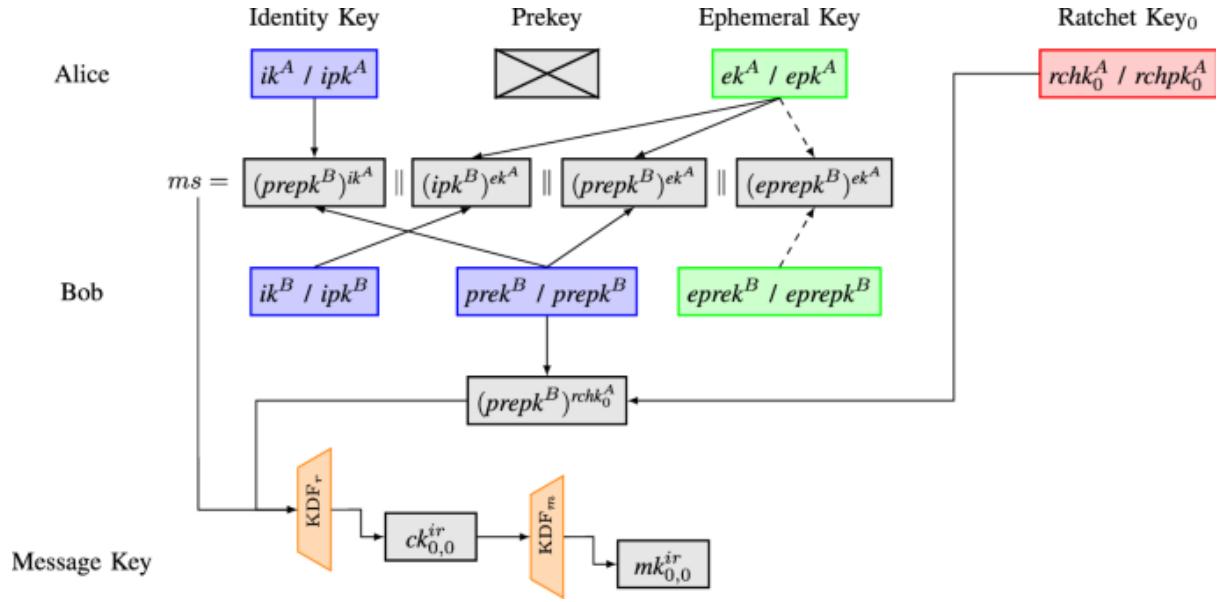
L'applicazione da praticamente il nome al protocollo essenzialmente. E' molto importante perché definisce alcune tecniche che hanno influenzato anche tanti altri contesti. Signals non è fatto per un sistema federato e c'è una unica autorità che gestisce l'infrastruttura per l'inoltro dei messaggi e abbiamo una unica autorità che gestisce chi è autorizzato a comunicare su signals. Uno dei problemi principali è che non abbiamo una unica autorità che sa l'elenco completo di chi può accedere al sistema. Si decide quindi di re-centralizzare il server e si mira ad avere garanzie molto più avanzate rispetto ad altri sistemi.

Immaginiamo un protocollo di scambio di chiavi in cui Alice e Bob conoscono le chiavi pubbliche l'uno dell'altro. Il sistema è sempre quello, per cui ho il calcolo di Diffie Hellman. Non so come Alice e Bob hanno ottenuto queste chiavi pubbliche, ma qui se ho uno schema autenticato assumo che loro abbiano le chiavi autentiche e bona. La distribuzione delle chiavi non la vediamo, perché signals non la gestisce, al massimo checka la verifica delle chiavi. E' un approccio tofu. Vedremo i signals in maniera più complessa è che l'uso di DH con firme digitali non è l'unico modo per avere scambi di chiavi autenticate. Abbiamo un approccio differente non usato nel web. Questo schema praticamente mi permette di avere una chiave che non mi lega direttamente ad un determinato messaggio che ho cifrato con la mia chiave. Questo schema è DH based authenticated key agreement.

Assumiamo di conoscere le chiavi pubbliche rispettivamente. Schema applicabile solamente se la chiave pubblica è legata al protocollo di DH e siano long term. Potrei anche usare questa roba in web, ma dovrebbero avere delle chiavi pubbliche compliant. Sicuramente non lo uso con chiave pubblica con RSA → non è compatibile con una pletora di schemi usati comunemente. Praticamente si scambiano i contributi pubblici effimeri. Uso poi una key derivation function che prende come input la concatenazione dei due segreti di DH calcolati in modo indipendente. Quello che viene trasferito sono due segreti già calcolati essenzialmente.

L'idea è quella di usare una singola coppia di chiavi per un unico scopo. Nei protocolli se volessi avere crittografia end to end potremmo andare a usare intermediari. Il sistema per funzionare in maniera corretta deve assumere che i server comunque facciano qualcosa e soddisfino alcuni protocolli specifici.

Paradigma tofu per cui le chiavi pubbliche lungo termini sono scambiate e il server potrebbe spoofarle, ma tofu approccia che questo primo scambio venga fatto con successo. Useremo delle tecniche per gestire le chiavi in modo da avere forward secrecy in tutti i modi, in maniera per altro embeddata e senza l'intervento dell'utente.



Immaginiamo poi di voler far comunicare Bob con Alice, che però è offline e quindi il server deve poter andare ad ospitare i messaggi che non sono recapitati.

Assumiamo il server che sia sano. Praticamente devo usare tecniche di aggiornamento e rotazione delle chiavi dopo il primo contatto con il server per mantenere forward secrecy. X3DH + Double ratchet algorithm sono i due metodi per andare ad implementare tutto questo. Vedremo che vengono usate effettivamente anche delle firme digitali e questo protocollo di scambio di chiavi è basato su EDSA. La variante di signal genera il nonce in maniera ibrida e usa un approccio deterministico e ci mette dentro anche dei dati random.

X3DH

Basato su 3 macro operazioni, non è un handshake che possiamo andare a risolvere in una tornata lineare sola. Qui abbiamo abbiammo 3 macro fasi indipendenti fra di loro perché andiamo a gestire tutto quanto in maniera disaccoppiato.

Immaginiamo che il destinatario si registra a signals. Successivamente un mittente vuole andare a iniziare una nuova comunicazione. Qua Alice non è che deve aspettare che Bob diventi online per mandare roba cifrata. Alice una volta che vuole mandare roba a Bob, anche senza avere mai conosciuto Bob prima si può fare automaticamente e quando Bob si risveglia, si connette al server, scopre messaggi cifrati e dovrà derivare in maniera disaccoppiata la chiave asimmetrica. Il server fa da repo pubbliche di chiavi pubbliche.

In fase di registrazione si invia al server la sua chiave pubblica e la si associa alla sua identità in maniera univoca. Però a livello tecnico dobbiamo aggiungere altre informazioni per avere garanzie avanzate. Aggiungiamo anche una signed pre key.

Pre key lo diamo perché sono chiavi caricate in anticipo, ma usato per cifrare informazioni solamente nel futuro. Il termine signed indica il fatto che quello che carichiamo è firmata dalla chiave di lungo termine associata ad una determinata identità. Ovviamente è al di fuori del protocollo la verifica dell'identità (solitamente fatto con sms).

Proviamo a mandare dei messaggi: il server fornisce ad A l'identità, la chiave pubblica, one time pre key (nonostante sia una chiave simmetrica). Appena il server fornisce queste informazione si riducono le informazioni salvate sul server. X3DH è una estensione di un protocollo id scamibio di chiavi DH only. Poi genero 3 + 1 segreti di DH: abbiamo la possibilità di andare ad usare una one time pre key. Il protocollo esiste in due varianti: (la one time key è una mitigazione pre prevenire in maniera forte attacchi handshake). Cioè voglio imporre di usare una chiave una sola volta, perché se la sto ricevendo più volte sto andando a subire un replay attack! Questo mi permette di avere una situazione in cui in qualunque caso non si possa avere una chiave simmetrica che è stata usata nel passato.



Leggiti direttamente la documentazione di signals per capirci qualcosa