2. AUTHENTICATION

Magnus Jensen

- *Tabel løsning - ubetinget sikkerhed*
- *Beregningstungt sikkerhed*
- *Secret key*
- *Algoritmer*
- *Sikkerhed*
- *Eksempel: CBC-MAC*
- *Public Key*
- *Algoritmer*
- *Sikkerhed*
- *Fordele ved public-key*
- *RSA Implementering*
- *RSA PROBLEM*
- *Hashfunktioner*
- *RSA & Hash*
- *Replay Attack*
- *NONCE*

AUTHENTICATION

Så når vi snakker om authentication, så er det ideen om, at vi vil kunne **bekræfte identiteten** for den vi kommunikere med.

Tabel løsning - ubetinget sikkerhed

En løsning der giver **ubetinget sikkerhed**, hvor det simpelt ikke er muligt at for en fremmed at udnytte systemet.

Sender og modtager **aftaler** en MAC for hver mulig besked på forhånd.

> *MAC = Message Authentication Code*

En MAC er en **message authentication code**, og er unik og uafhængig af beskeden.

Når man **sender en besked**, sender man så den tilhørende MAC, så modtager kan tjekke at de passe sammen.

> *(en tilfældig tabel)* | *MESSAGE* | *MAC* | |:--|:--| | *HEY* | *143* |

Eftersom tabellen er hemmeligt aftalt, vil kun de to kunne kommunikere sammen.

Derved kan samme besked dog kun sendes en gang, men en fremmed vil aldrig kunne gætte en MAC til en besked, der da ingen sammenhæng er.

Beregningstungt sikkerhed

I praksis er **ubetinget sikkerhed ikke brugbart**; hvorfor vi bruger sikkerhed baseret på beregningstunge algoritmer.

Vi kigger derfor på to tilgange til dette.

Secret key

Et **secret key system** er hvor parterne på forhånd har **aftalt en nøgle**, som de bruger til at godkende hinandens beskeder.

Algoritmer

Der er tre algoritme:

> - `G = k` *laver en nøgle*
> - `MACk(m) = c` *laver en MAC for* ***m*** *under nøglen* ***k***
> - `Vk(m, c) = true/false` *verificere MAC'en for beskeden*

Sikkerhed

I sådan et secret key system, så skal det **aldrig** være muligt for en fremmed at **lave en gyldig MAC**.

Det kunne han f.eks, hvis han **ud fra nogle beskeder og deres MACs gennemskuer nøglen.**

Derfor bruges nøgler af 128bit længde, da det er for meget for moderne computere.

> *Secret Key længde = 128 bit*

Eksempel: CBC-MAC

Lad os tage et eksempel! **Cipher Block Chaining Mac**, er en algoritme til at lave MAC's med.

Den laver en MAC ved dele beskeden op i dele, og så maccer en del ad gangen.

> *C0 = IV Ci = AESk(Mi ⊕ Ci-1)*

Vi kan altså se, at en C er afhængig af den **udregnet før**.

Sammen med beskeden sender vi så den **sidst udregnede C**.

Modtager kan så tjekke ved at lave samme udregning, og det virker fordi kun de to har nøglen.

Public Key

Et **public key system** er hvor hver har en privat key de holder for sig selv, og en public key som andre må kende til.

Man kan da signere en besked med sin private nøgle, og andre kan tjekke den er fra en, ved at bruge ens public key.

Algoritmer

Der er tre algoritme:

> - `G = (pk, sk)` *laver to nøgler*
> - `SIGNsk(m) = c` *laver en signatur for* ***m*** *under nøglen* ***sk***
> - `Vpk(m, c) = true/false` *verificere signaturen for beskeden*

Sikkerhed

I sådan et public key system, så skal det **aldrig** være muligt for en fremmed at **lave en falsk men gyldig signatur**.

Der **findes algoritmer** der kan udregne en secrete key ud fra en public key, derfor bruges meget store nøgler på **minimum 2048 bit**.

> *Public Key længde = 2048 bit*

Fordele ved public-key

Når man bruger en MAC, kan B ved modtagelse af m fra A; se om den **virkelig var fra A** (givet at kun A og B kender MAC); men **B kan ikke bevise** for nogle andre at det var A der sende beskeden, den kunne ligeså godt have været generet af B.

Her er formålet ved at bruge **public key systemet**; for **alle kan have A's pk**; men da kun A har SKa; er det kun A der kan signere en besked m. Så sender A en signeret besked til B, kan C der har SKa verificere at A lavede m og ikke B, selvom det er B som nu er i besidelse af den.

RSA Implementering

RSA laver nøglerne:

> *Public key: (e, n) Secret key: (d, n)*

Og algoritmerne:

> - `SIGNsk(m): m^d % n`
> - `Vpk(m, s): (s^e % n) == m`

RSA PROBLEM

Et problem er, at en **fremmed** kan bare **vælge et tal** og sige det er en **signatur** - og udføre noget af *v* herpå, for at få en *m*.

> $m = s^e \ \% \ n$

Så kan den fremmede sende (m, s), det er dog muligt *m* ikke betyder noget. Men det er stadig et problem.

> *send (m, s)*

... men det vender vi tilbage til

Hashfunktioner

En hashfunktion producere et unikt fingerprint.

En hash funktion har følgende egenskaber:

>

- *Tager input af alle længder*
- *Giver output af fikseret længde*
- *Hurtig som secret key*
- *Svært beregningsproblem, til sjælens kollision*

En **kollision betyder** at vi ikke vil have:

$x != y, h(x) == h(y)$

Af hash funktioner kan nævnes:

- *MD5*
- *SHA-1*
- *SHA-256*

RSA & Hash

Ved at bruge hash i RSA kan man stoppe førnævnte problem.

Vi laver to nye algoritmer:

- `SIGN2(m): SIGN(h(m))`
- `V2(m, s): V(h(m), s)`

Fordi hashen typisk er mindre end en besked, er algoritmerne først og fremmest **hurtigere nu**.

Men dernæst **løser de problemer** fordi, at en signatur nu **afhænger af en hashet** udgave af en besked, at man derved ved overstående trick ikke **får *m* men hashen af *m***.

Replay Attack

At bruge signature beviser bare, at vi **på et tidspunkt godkendte en besked**; men en fremmed kan i teorien stjæle beskeden og signaturen og **sende den igen og igen**. Dette hedder et replay attack og kunne **f.eks bruges mod en bank**.

Der er flere måder at forhindre dem på

> - *Huske hver besked*

Men det er dumt

> - *Man kunne bruge counters*
> - *Man kunne bruge timestamps*

Men det er alt sammen noget **hvor man skal huske**..

NONCE

En ordentlig metode er, at en modtager vælger et nonce og sender til senderen. Dertil sender senderen sin besked plus en MAC udregnet over det nonce og beskeden. Det forhindre replays, da nonces kun bliver brugt en gang.