AT Computing
Arnhemsestraatweg 33
6881 ND Velp
The Netherlands

Keys en Certificaten

Versie: 1c

Inhoudsopgave

1 Keys: symmetrisch en public/private

Een key is een getal. En een tekstboodschap is een reeks van bytes, dus ook getallen. **Encryptie** betekent dat je die tekst en die key samen in een bepaalde "molen" stopt, en de uitkomst is dan de versleutelde (geëncrypte) tekst. In die molen wordt een stappenplan doorlopen, waarbij de uitkomst van de ene stap weer een rol speelt in de volgende stap. Heel globaal kun je denken aan Booleaanse XOR-operaties tussen de bitjes van de tekst-bytes (als lange string achter elkaar gezet) en de bitjes van de key (als kortere string). De key schuift steeds een stapje op langs de tekst, de XOR wordt herhaald, weer een stapje opschuiven, weer XOR, enz. enz.

Er zijn twee soorten keys: **symmetrische** keys en **public/private** keys.

Bij een symmetrische key dient hetzelfde getal voor encryptie én voor decryptie (ontsleutelen). Beide partijen -de partij die encrypt, en de partij die decrypt- moeten de key kennen, en 'm geheim houden. Zo'n key noemt men een **shared secret**.

Bij public/private keys heb je twee getallen; die vormen een **keypair**. Wat met het ene getal/key is geëncrypt, kun je decrypten met het andere. Dat werkt beide kanten op. Eén van die twee getallen krijgt de rol van "private key", en blijft streng geheim. Het andere getal wordt de "public key"; iedereen krijgt dat te zien.

Met public/private keys kun je meerdere doelen bereiken:

- Doel: vertrouwelijkheid (confidentiality)
  Stel dat Alice een geheime boodschap aan Bob wil sturen. Ze moet de boodschap dan encrypten met Bob's public key. Voor decrypten is Bob's private key nodig; alleen Bob zelf heeft die, en die houdt hij geheim. Dus alleen Bob zal de boodschap kunnen lezen.
  Als Bob wil antwoorden aan Alice dan heeft hij Alice's public key nodig. Merk op: bij deze communicatie tussen **twee** partijen spelen dus **twee** key**paren** een rol; elke partij heeft z'n eigen paar. Dat zijn samen **vier** keys.
- Doelen: authentication en non-repudiation
  Stel dat Alice een boodschap wil publiceren. Die kan ze encrypten met haar eigen private key. Voor decrypten is haar public key nodig, en iedereen kent die, want die heeft ze op haar website gezet, of in haar e-mail signature, of op een **public key server**, enz. Als het decrypten daarmee lukt dan is bewezen dat de boodschap van Alice zelf afkomstig moet zijn, alsof haar handtekening eronder staat. Certificaten gebruiken dit 'authentication'-principe.
  De ontvanger weet nu zeker dat de boodschap van Alice afkomstig is, want alleen zij (Alice) bezit de key waarmee de boodschap geëncrypt was. Maar om diezelfde reden kan Alice ook niet meer ontkennen dat het háár boodschap is. Dit laatste heet "non-repudiation"; in goed Nederlands zou je zeggen: "niet meer te verloochenen".

Daarnaast krijg je bij elke vorm van encryptie nog een controle op **integriteit** van het transport cadeau:

als onderweg met een geëncrypte boodschap is gerommeld dan gaat de decryptie (waarschijnlijk) niet meer goed.

Wanneer je te maken hebt met public/private keypairs dan moet je één basisprincipe centraal stellen: “Je private key geef je **nooit** uit handen”. Nooit, nooit, nooit…. Hij mag **nooit** jouw lokale/persoonlijke machine verlaten. Hij zit dus ook nooit in een certificaat of in een andere verpakking die tijdens een communicatie wordt verstuurd. En als een nieuw keypair gemaakt moet worden, maar je doet dat niet zelf, dan moet je goed nadenken hoe je het private gedeelte naar jouw lokale machine toehaalt, en hoe je zeker weet dat geen kopie achterblijft bij de maker.

Als je niet zelf de administrator bent van de machine waarop je private key staat, dan moet je ook nog apart nadenken over hoe je zorgt dat die administrator niet bij jouw key kan komen. Meestal gebeurt dat door die key zelf weer te encrypten met een apart password of een **passphrase**. Daar komen we nog op terug.

Nog een beetje jargon: de oorspronkelijke tekst wordt “cleartext” genoemd, en versleuteld heet het de “ciphertext”.

2 Hashing (digesting) versus encryption

Dit zijn twee technieken die vaak met elkaar verward worden. Bij hashing wordt op de originele tekst een berekening losgelaten waar uiteindelijk een relatief klein getal uitrolt: het 'hashgetal'. Als in de originele tekst een wijziging wordt aangebracht dan levert dezelfde berekening daarna, met grote waarschijnlijkheid, een ander hashgetal op. De mate van waarschijnlijkheid hangt af van de gebruikte berekening: het hash-algoritme. Bekende hash-berekeningen zijn MD5 en de SHA-familie. De oudere technieken **parity** en **CRC** (checksums) zijn ook hashes, maar wiskundig veel minder verfijnd.

Dus als iemand een bestand (tekst, software-distributie o.i.d.) verspreidt en tegelijk de hash publiceert, bijv. op een website, dan kan een ontvanger de hash opnieuw uitrekenen en vergelijken met de gepubliceerde, originele hash. Zo controleer je of met dat bestand is gerommeld. Het bereikte doel is: **integrity**.

Hash wordt ook wel **digest** genoemd, naar het Engelse 'spijsvertering': er gaat een groot bord eten in, en er komt een klein poepje uit. Bij encryptie is het de bedoeling om de originele tekst ooit te reconstrueren; bij hashing lukt dat uiteraard nooit meer.

Het doel van een hash is dus om te controleren of met een tekst is gerommeld. Het gaat hierbij niet per se om geëncrypte tekst. Open Source programmacode is een goed voorbeeld van cleartext materiaal waar door de makers vaak een hash bij wordt gepubliceerd. Hashing en encryptie komen natuurlijk ook in combinatie voor, maar je moet beseffen dat het twee afzonderlijke mechanismen zijn.

In dit kader past ook de **digitale handtekening**. In het algemeen wordt hiermee bedoeld: van de (te ondertekenen) document-tekst wordt een digest/hashgetal berekend, en die hash wordt geëncrypt met iemands private key. Dat wordt aan het document toegevoegd als ondertekening. Degene die controleert, dus het hashgetal opnieuw heeft uitgerekend, krijgt dan niet te maken met de situatie dat de orginele hash elders is gepubliceerd, maar hij moet de public key van de ondertekenaar te pakken zien te krijgen, en daarna vindt hij de originele hash in het document zelf. Als het om veel verschillende documenten gaat, dus veel verschillende hashes, is controle via één centraal gepubliceerde public key natuurlijk gemakkelijker te organiseren dan het apart publiceren van al die originele hashgetallen.

Het "kraken" van een hash is iets heel anders dan het kraken van een encryptie. Bij een hash zoekt men naar technieken om in een originele cleartext een gewenste wijziging aan te brengen, en daarbij nog dummy bytes toe te voegen, zodat uiteindelijk de totale file weer hetzelfde hashgetal oplevert als voorheen. Men noemt dat niet "kraken" van de hash, maar "genereren van een hash collision". Met MD5 is dat inmiddels gelukt, en met sommige korte leden van de SHA-familie ook.

3 | Wat is een certificaat

Een certificaat is als een paspoort: een bundel identiteitsgegevens, verpakt en uitgegeven door een autoriteit die garandeert dat de gegevens correct zijn. Als een onbekende tegen jou zegt: “Ik ben Piet”, dan weet je niet of hij de waarheid spreekt. Als hij een paspoort overlegt en zegt: “Ik ben Piet en dit is mijn identiteitsbewijs” dan weet je al wat meer, maar nog steeds niet genoeg. Je moet **twee** controles doen:

- Je controleert of het paspoort inderdaad van ‘de autoriteit’ komt, dus dat het geen vervalst exemplaar is: je bekijkt de echtheidskenmerken van het document.
- Daarna controleer je de koppeling tussen het paspoort en de persoon die voor je staat: je vergelijkt foto, handtekening, vingerafdruk, enz.

Nemen we als voorbeeld een https-webserver. Met jouw browser contacteer jij die server, en je krijgt zijn certificaat. Daarin staan onder meer zijn domeinnaam en zijn public key. Maar het feit dat de server met dat certificaat zegt: “Ik ben `www.ATComputing.nl` en dit certificaat is mijn identiteitsbewijs” is niet genoeg. Daar horen nog de twee extra controles bij.

De eerste controle, of het certificaat vervalst is, is ingewikkeld. Dat gaan we nog uitgebreid bespreken. Dan vertellen we ook wie die ‘autoriteit’ voor certificaten is. De tweede controle, voor koppeling tussen het certificaat en “de server die zich legitimeert”, is daarna gemakkelijk: jij (client) kiest een willekeurige tekststring en encrypt die met de public key uit het certificaat. Dat stuur je naar de server met wie je verbinding hebt. Want hij (en hij alleen!) zou de bijbehorende private key moeten hebben. Stuurt die server jouw encrypted string correct gedecrypt terug, dan weet je dat dat goed zit: koppeling tussen **identiteitsbewijs** en **identiteit zelf** is OK. Deze manier van controleren heet **challenge-response**. Die naam spreekt voor zich[1] .

Nu moet je (c.q. je browser) nog controleren of het certificaat, waarin dus o.m. de domeinnaam en de public key van de server staan, inderdaad geleverd is door ‘de autoriteit’, dus dat het niet vervalst is. De ‘autoriteit’ heeft bij het uitgeven een hash berekend over belangrijke delen van het certificaat, die hash daarna geëncrypt met **zijn** private key, en dat aan het certificaat toegevoegd. Dit hebben we eerder in dit verhaal een ‘digitale handtekening’ genoemd. De browser haalt de public key van ‘de autoriteit’ op, decrypt daarmee de hash in het certificaat, en controleert de berekening van die hash. Klopt dat, dan is dat certificaat inderdaad geleverd door ‘de autoriteit’. Het ophalen van de public key van ‘de autoriteit’ heeft nog wel wat voeten in aarde; daar komen we nog op terug.

In technische termen zegt men: een certificaat is *gesigneerd* door een *trusted third party*. Signeren=ondertekenen, maar dat komt dus neer op encrypten met een private key. *Trusted third party* betekent: een tussenpersoon die zowel door de client als ook door de server vertrouwd wordt. Je kunt de rol van deze **trusted third party** vergelijken met de overheid die paspoorten uitgeeft, maar ook met

[1] Realiseer je dat die server-naam `www.ATComputing.nl` nog door DNS naar een IP-nummer moet worden vertaald. Als je dat **secure** wilt hebben dan kom je bij DNSSEC. Want als jouw aanvaller het antwoord op een DNS-query kan vervalsen, én een vals certificaat op jou loslaat, dan gaat de challenge-response goed.

een notaris bij een juridische transactie, of met banken in het betalingsverkeer. In het jargon noemen we die tussenpersoon de CA ("Certificate Authority").

> Een certificaat is: de public key van een webserver (of mailserver, of …), plus identiteits-gegevens van die server, inclusief domeinnaam enz., waarbij bepaalde delen zijn geëncrypt met de private key van een autoriteit die als *trusted third party* fungeert.
>
> De bedoeling is dat die *trusted third party* (CA) gecontroleerd heeft dat domeinnaam en de andere gegevens met elkaar overeenkomen. De CA voegt ook nog allerlei administratieve details toe, zoals z'n eigen naam, een serienummer, een geldigheidstermijn (begin+einde), enz.

Als je dit heel minimalistisch bekijkt dan gebeurt hier eigenlijk het volgende: een server-beheerder genereert een keypair, en stuurt de public key en zijn domeinnaam naar een CA. Dat heet een 'Certificate signing request' (CSR), een soort bestelformulier. Waarschijnlijk kenden die beheerder en die CA elkaar tot op dat moment niet eens. De CA vraagt bijvoorbeeld via de **whois** service bij de registrar van de domeinnaam na wie de contactpersoon voor dat geregistreerde domein is, stuurt emails naar de betreffende mailadressen, en verwacht antwoorden.

Een andere controlemethode is dat de CA aan de beheerder vraagt om een bepaald brokje informatie ergens tussen de website-files te plaatsen, en met een `wget/curl`-achtige techniek controleert of de beheerder dat inderdaad klaarspeelt. Ook kan de CA eisen dat een speciaal `TXT` of `CNAME` record wordt toegevoegd aan de DNS-server van het domein, en met DNS-queries controleren of dat lukt.

Als de controle naar tevredenheid is dan signeert de CA de aangeleverde gegevens van het CSR, en daarmee is het certificaat klaar.

Bovengenoemde controles zijn allemaal 'op afstand'. Maar het kan ook veel verder gaan: KvK controle, openbare telefoonnummers nabellen, bedrijfsbezoek, enz. en dat beïnvloedt natuurlijk het prijskaartje. Er is zelfs een officiële classificatie van certificaten, gebaseerd op de diepgang van de controle (DV **Domain Validation Certificate**[2] ,

OV **Organization Validation Certificate**, EV **Extended Validation Certificate**). Browsers kunnen de klasse van een ontvangen certificaat zichtbaar maken door het tonen van alleen een groen slotje, of door naast het slotje de bedrijfsnaam te vermelden. Dat laatste betekent dat de CA ook het verband tussen domeinnaam en bedrijfsnaam heeft gecontroleerd.

De browser kan verkeer naar de webserver encrypten als hij de public key van die webserver heeft. Die staat in het certificaat, maar de browser wil eerst de geëncrypte hash in dat certificaat controleren voordat hij het vertrouwt. Voor dát decrypten heeft de browser de public key van de CA nodig. Dat hadden we hierboven al genoemd, maar dat had wat meer voeten in aarde.

De public keys van heel veel CA's zijn in elke browser ingebouwd. Dus als de gebruiker zijn eigen browser vertrouwt, en als de server-beheerder zijn certificaat inderdaad gekocht heeft bij een van die *browser-built-in* CA's, dan is de cirkel nu rond.

Die *browser-built-in* CA's worden ook **root CA's** genoemd. Overigens: als je die lijst van root CA's bekijkt dan zakt de moed je in de schoenen. Er zitten degelijke commerciële partijen bij (bijv. Veritas/Verisign) en heel veel nationale overheden, waaronder de *Staat der Nederlanden*. Of je al die overheden even betrouwbaar vindt moet je maar voor jezelf bepalen, want ze hebben beslist niet allemaal een even goede reputatie. En als zo'n overheid misschien zelf wel betrouwbaar is dan moet je maar hopen dat elke onderaannemer, die certificaten verzorgt namens zo'n overheid, zijn zaakjes op

[2] De bekende gratis certificaten van het "Let's Encrypt" consortium zijn van het DV type. Hun controle bestaat uitsluitend uit de constatering dat het (door hen verschafte) script waarmee het certificaat wordt aangevraagd, in staat is om een (ook door hen verschaft) bestandje op de website te plaatsen.

orde heeft. De grootste rel (tot nu toe) op dát gebied is de DigiNotar-affaire geweest: niet in een bananenrepubliek of schurkenstaat, maar in ons eigen land.

Sommige CA's maken het aanvragen van een certificaat heel gemakkelijk via een web-interface dat ook wel even het keypaar voor de klant kan genereren. Eerder in dit artikel hadden we al gezegd dat als je je keypaar niet zelf genereert, je zeker moet weten dat bij de maker geen kopie van je private key achterblijft. In principe is een CA die zo'n dienst aanbiedt niet te vertrouwen.

Een jargonterm: een **Public Key Infrastructure** (**PKI**) is het geheel van policies en andere organisatievormen voor het maken, publiceren, zo nodig terugtrekken, enz. enz. van keys. Dus precies wat de naam zegt: een "infrastructuur" rondom het werken met "public keys".

4 | Certificate chain

Soms vindt de beheerder van een server die root-CA's te duur, of de validation policy van zo'n CA te moeilijk, enz. Er zijn genoeg tussenhandelaar-CA's met gunstiger voorwaarden. In jargon heten die "intermediate CA's", of "Registration Authorities" (RA's). Maar hún public key zit niet in de browsers ingebouwd.

Als de server-beheerder een certificaat koopt van zo'n intermediate CA, dan levert die niet alleen een certificaat, maar ook een z.g. Certificate Chain. Dat is een apart certificaat waar de public key van die CA in zit, met een hash die is gesigneerd (geëncrypt) met de private-key van een *hogere* CA. Bij elke verbindingsopbouw moet de server nu dus zijn (eigen) Certificate sturen, plus de Certificate Chain. Dat kan in de vorm van twee aparte files, maar ook gecombineerd in één file. De browser moet daarna de public key van die *hogere* CA te pakken zien te krijgen.

Als die *hogere* CA, met z'n public key, in het browser-built-in lijstje zit, dan hebben we weliswaar een extra stap overhead, maar de cirkel is toch weer rond. En als die *hogere* CA niet in de browser bekend is, dan moet in de chain-file ook zíjn public key staan, gesigneerd met de private key van een nóg hogere CA, enz… Die reeks (chain) moet altijd eindigen bij een van de root CA's.

5 | Root CA's

Browser leveranciers zijn héél terughoudend in het accepteren van root-CA's in hun "built-in" lijst. Er zijn ook allerlei voorschriften, van overheden en van organisaties van accountants e.d., verenigd in het **CA/Browser Forum**, over hoe een root-CA de toegang tot zijn keypair moet afschermen. Dat gaat heel ver: genereren en opslaan moet in een "hardware security module", die moet worden bewaard in een kluis met 24 uurs cameratoezicht en uiteraard zonder netwerk-toegang, met uitgebreide procedures en protocollen voor als een mens daar in de buurt mag komen, enz.

Daarom beginnen die root-CA's meestal met het oprichten van een eigen intermediate CA. De "echte" root private key wordt gebruikt om een of meer intermediate certificaten te signeren. Dat gebeurt tijdens een "key signing ceremony", onder toezicht van accountants, andere getuigen, camera's enz., en nog tijdens de ceremonie gaat die root private key weer de kluis in. Zo ontstaat dus meteen al de eerste schakel van een chain.

Een zwakke plek in dit verhaal is dat browser leveranciers te maken krijgen met overheden van landen waar ze zaken willen doen. Als zo'n overheid een eigen root CA naar voren schuift dan kun je niet veel eisen stellen aan procedures en protocollen. Een overheid kan zelfs toegang hebben tot het root-keypair van die CA, en daarmee dus valse certificaten voor willekeurige domeinnamen genereren. Bedenk dat een browser een certificaat als geldig aanneemt als de chain daarvan eindigt bij één van de "built-in" root-CA's, dus willekeurig welke. Als de providers in zo'n land ook nog worden verplicht om mee te werken aan "man in the middle" afluisterpraktijken, bijvoorbeeld via DNS-manipulatie, dan blijft niet veel privacy over.

Soms kom je bij een certificaat de aanduiding `G2`, `G3`, enz. tegen. Dat is een nummering die een CA gebruikt om de generatie van zijn certificaten te onderscheiden. Als een CA bijvoorbeeld een nieuwe private key gaat gebruiken, of een ander hashing mechanisme, dan wordt dit generatienummer opgehoogd zodat het verificatie-proces de oude en nieuwe certificaten uit elkaar kan houden totdat de geldigheid van alle oude certificaten verlopen is.

6 | File formaat .pem

Een certificaat wordt geleverd in de vorm van een `.pem`-file[3] .

Dat staat voor “Privacy Enhanced Mail”. Het is een format dat oorspronkelijk is ontworpen voor mail-communicatie, hoewel Internet-mail tegenwoordig een ander formaat gebruikt. Een erfenis uit die mail-oorsprong is dat in de file alleen tekstbytes (7-bit ASCII) mogen staan.

Maar encryptie-algoritmen produceren ook non-tekst bytes. Daarom is een `.pem`-file als laatste stap omgezet naar z.g. **base64**-encoding. Dat is identiek aan de manier waarop image-files (`.jpeg`, `.gif` enz.) voor e-mail attachments worden omgezet naar volledig leesbare tekstbytes. Merk op dat **base64**-encoding dus geen veiligheid toevoegt: encoding is een soort vertaling, maar het is geen encryption. Je hebt wel een decoding-commando zoals `base64` (standaard in Linux aanwezig) nodig om de inhoud van zo’n `.pem`-file te bekijken. De geëncrypte gedeelten zijn dan nog steeds geëncrypt, dus nog steeds geen leesbare tekst.

Als je meer wilt weten over de interne opbouw van certificaten dan kun je de Wikipedia-pagina’s over **Public Key Certificate** en over **X.509** opvragen.

In Microsoft-omgevingen kom je de filenaam-extensie `.pfx` tegen. Dat is een ‘transportverpakking’ waarin zowel een certificaat als ook de bijbehorende private key zitten. Doel hiervan is om die combinatie gemakkelijk te kunnen distribueren (export, import) over bijv. een server farm. Zo’n `.pfx` file is encrypted met een password, maar omdat er een private key in zit moet je er toch heel zorgvuldig mee omgaan.

[3] Naast `.pem`-formaat bestaan nog andere formaten, bijv. `.crt` en `.cer`, of `ca-bundle` (voor de chain file).

7 | Certificate revocation

Certificaten hebben een eindige geldigheidsduur, en de einddatum staat in het certificaat. Maar soms moet een certificaat al eerder ongeldig worden verklaard, bijvoorbeeld als de private key van de website uitgelekt is. De situatie is nog erger als een CA in de fout is gegaan: haar private key kan via een inbraak zijn gestolen, of hackers hadden toegang tot de systemen van de CA en genereerden daar zelf certificaten (zoals bij DigiNotar is gebeurd) of het wordt op een andere manier duidelijk dat die CA niet veilig werkt. In dat geval worden alle certificaten die door zo'n CA zijn uitgegeven ongeldig verklaard. De vakterm is "revoked" (herroepen).

Het is de bedoeling dat een browser die een certificaat ontvangt apart controleert of dat certificaat revoked is. In het certificaat staat de URL van een **Certificate Revocation List** (CRL), en de browser gebruikt het **Online Certificate Status Protocol** (OCSP) om die lijst te raadplegen.

Een zwak punt in deze situatie treedt op als de OCSP responder server niet reageert. De meeste browsers tonen de website dan maar gewoon alsof niets aan de hand is.

Bij de DigiNotar inbraak werd ten onrechte een certificaat op naam van `*.google.com` gegenereerd. De "aanvrager", dus bezitter van de bijbehorende private key, was een hacker. Ook andere CA's maakten vergelijkbare fouten. Als reactie daarop is toen een nieuw type DNS-record ingevoerd: het **Certificate Authority Authorization** (CAA) record. Hierin zegt een domeineigenaar welke CA hij heeft gekozen voor het uitgeven van zijn certificaat.

Dat CAA record wordt gebruikt door een CA die van een klant een Certificate Signing Request (bestelling) voor een certificaat ontvangt. De CA is verplicht om te controleren of de klant een CAA record in zijn DNS heeft, hoewel dat hebben zelf niet verplicht is. Maar áls de klant er een heeft dan controleert de CA of hij (de CA) daarin staat genoemd.

8 | Session key

In het certificaat zit dus de public key van de webserver. Als de browser de hash uit dat certificaat heeft gedecrypt (met behulp van de public key van 'de autoriteit'), en de hashberekening heeft gecontroleerd, dan vertrouwt hij (browser) die public key van de webserver ook. Daarmee kan hij verkeer vertrouwelijk naar de server sturen. Maar dat werkt maar één kant op. In de inleiding van dit verhaal hadden we gezien dat voor verkeer in omgekeerde richting de ontvangende partij (dat is dan dus de browser) een eigen keypair zou moeten hebben.

Op zichzelf zou de browser zo'n keypair kunnen maken, en het public deel daarvan naar de server sturen. De ontwerpers van HTTPS hebben een iets andere weg gekozen[4] :

server en browser kiezen in overleg een **symmetrisch** encryptiealgoritme. Daarmee bouwen ze een symmetrische key, en die gaat dienst doen als 'shared secret'. Tijdens deze selectieprocedure speelt de client de voortrekkersrol, want hij kan al veilig encrypted verkeer naar de server sturen; andersom is aanvankelijk nog niet mogelijk.

Als de symmetrische key eenmaal gemaakt is dan wordt die tijdens de rest van de sessie gebruikt om alle dataverkeer, in beide richtingen, te versleutelen. Daarom heet dat een **session key**. Bij afsluiten van de sessie vervalt die. Voor deze aanpak is mede gekozen omdat versleutelen van data met een symmetrische key veel minder rekenwerk kost dan met een public/private keypair.

[4] Strikt genomen is dit geen keus van HTTPS maar van TLS, het **Transport Layer Security** protocol. De implementatie daarvan wordt vaak SSL genoemd, zoals bijvoorbeeld in OpenSSL. Deze naam SSL staat voor **Secure Socket Layer**. De eerste generatie van het protocol heette zo; latere generaties is men TLS gaan noemen.

9 | Webserver configuratie

In de configuratie van een webserver (bijv. Apache) spelen dus drie files een rol:

1) De private key (file) van de server. Die is streng geheim en mag de server-computer nooit verlaten. Maar het serverproces moet er wel op disk bij kunnen. De server stuurt zijn public key (zit in het certificaat) naar de client, en ontvangt vervolgens verkeer van die client dat daarmee is geëncrypt. Om dat te decrypten heeft de server zijn private key nodig. Dit speelt een rol aan het begin van de sessie, totdat in gezamenlijk overleg de (symmetrische) session key is bepaald.

2) De Certificate file. Die bevat (onder meer) de domeinnaam en de public key van de server. De server stuurt deze file naar de browser/client bij het tot stand komen van de verbinding. De browser heeft die nodig voor het encrypten van informatie die hij naar de server gaat sturen. Maar die public key-file zelf is gesigneerd door een CA. Dat wil zeggen: ze bevat een hash die is geëncrypt met de private key van die CA. De browser heeft de public key van de CA nodig om dat te decrypten.

3) Als die public key van de CA **niet** is ingebouwd in de browser dan moet de server de Certificate Chain file meesturen. Daarin staat de public key van de CA, met een hash die op zijn beurt is gesigneerd (geëncrypt) met de private key van een hogere CA. Dus de browser heeft nu weer de public key van die hogere CA nodig. En als die niet is ingebouwd in de browser dan moet de Certificate Chain file ‘m meeleveren, maar dan wel weer gesigneerd door een nóg hogere CA, enz. enz. Nieuwe versies van Apache kunnen werken met een gecombineerde Certificate + Certificate Chain file.

10 | Self-signed certificates

Duur inkopen van een certificaat is niet altijd nodig. De server-beheerder kan zichzelf tot Certificate Authority uitroepen, een eigen keypair maken, daarmee een z.g. *self-signed*-certificate produceren en dat in zijn website installeren. Als een browser zo'n *self-signed*-certificate ontvangt dan krijgt de browsende gebruiker een pop-up scherm, waarmee hij dat certificaat kan goedkeuren en "handmatig" toevoegen aan de browser-built-in verzameling.

Browsende gebruikers moeten natuurlijk opgevoed worden zodat ze niet té lichtvaardig zo'n certificaat goedkeuren en in hun browser installeren.

Self-signed certificates worden spottend ook wel **Snake Oil certificates** genoemd. De naam komt van het "geneesmiddel" dat vroeger door rondreizende kwakzalvers werd verkocht op kermissen en jaarmarkten, en dat zou helpen tegen alle kwalen, van hoofdpijn tot eksterogen. Het is vergelijkbaar met wat we in Nederland kennen als "Haarlemmer Olie".

11 | Tweezijdige authenticatie

Aan het certificaat kan de browser/client zien of hij met de juiste server te maken heeft. Maar soms wil de server ook weten of hij met de juiste browser/client-partij te maken heeft. Dat heet *Client Authenticatie*. Daarvoor moet je een certificaat in omgekeerde richting organiseren: van browser naar server.

Een probleem is natuurlijk dat de gemiddelde browser-gebruiker geen duur certificaat bij een commerciële CA wil kopen. De server-beheerder zal hier dus voor moeten opdraaien, bijvoorbeeld door self-signed certificaten te maken en uit te delen. Dat is alleen praktisch in een besloten omgeving, waar de client-apparatuur van tevoren kan worden geprepareerd. Denk bijvoorbeeld aan werknemers die mobiel of vanuit huis niet-openbare webpagina's bij hun bedrijf moeten raadplegen.

De beheerder zal aan de server-kant moeten regelen dat die client-certificaten goed herkend worden. Bij Apache krijg je dan te maken met directives zoals SSLCACertificateFile en een aantal andere SSLCA…-directives. In deze directive-namen staan de letters CA voor **client authentication**. Aan de browser-kant komt daarbij nog de drempel van het (correct) installeren van zo'n certificaat.

12 | SSH en Diffie-Hellman

Wanneer je communiceert met SSH wil je ook vertrouwelijkheid, maar daar komen in eerste instantie geen certificaten aan te pas. Bij SSH is dus geen garantie dat je met de juiste server praat (maar lees ook de rest van deze paragraaf). SSH verzekert alleen maar dat je data onderweg niet kunnen worden afgeluisterd.

SSH is gebaseerd op een symmetrische encryptie, dus een ‘shared secret’. Stel dat je met je client naar een (ssh-)server gaat waar je nooit eerder bent geweest. Dus van tevoren kon je ook geen ‘shared secret’ afspreken. Het shared secret moet nu worden gecommuniceerd terwijl de communicatie nog niet is geëncrypt, dus op dat moment misschien al wordt afgeluisterd. Dit lijkt een kip-en-ei probleem.

De heren Diffie en Hellman hebben daar toch een methode voor gevonden (de Britse geheime dienst bleek deze methode vele jaren eerder al te kennen). Gebaseerd op een slim stuk wiskunde kunnen client en server een aantal brokstukken van informatie uitwisselen, terwijl ze worden afgeluisterd, en daarna kan elke partij die ontvangen brokstukken combineren met stukjes niet-uitgewisselde informatie, en zo toch zijn kopie van een shared secret samenstellen.

Bij een ssh-verbinding wordt de Diffie-Hellman techniek gebruikt om, vlak nadat de verbinding tot stand komt, een symmetrische **session key** te genereren. Vervolgens wordt alle verkeer daarmee geëncrypt.

Als een beheerder een ssh-server inricht dan moet hij op die server public/private keyparen maken. Die zijn dus niet bedoeld voor het dataverkeer, want dat gaat met een symmetrische key. De public/private keyparen helpen de client om te controleren dat hij de “echte” server aan de lijn heeft. Dat gaat als volgt:

Als een gebruiker met een ssh-client voor de eerste keer een bepaalde server bezoekt, dan vraagt die client aan zijn gebruiker of die zeker weet dat hij de juiste server aan de lijn heeft. Want SSH werkt niet met certificaten van een bepaalde autoriteit, dus de server kan zich niet legitimeren. Maar als de gebruiker bij dat eerste bezoek de gestelde vraag bevestigt, dan onthoudt de client de public key van die server in z’n ‘known_hosts’ verzameling. Bij elk volgend bezoek gebruikt de client de onthouden public key voor een challenge-response controle. De gebruiker wordt gewaarschuwd als die controle mislukt. Want dat betekent dat de server een andere private key is gaan gebruiken. Dat gebeurt als die server opnieuw geïnstalleerd is, maar kan ook gebeuren omdat de bedoelde server het zwijgen is opgelegd (bijv. met een DDos aanval), en je in plaats daarvan met een ‘valse’ server contact krijgt. Met die challenge-response controle heb je nog een beetje zekerheid dat je met de juiste server praat.

Ook een ssh-gebruiker (client) kan op zijn lokale computer een keypair maken. Het public deel daarvan kopieert hij naar naar de ‘authorized_keys’ verzameling van zijn login-account op een ssh-server waar hij vaak naar toe moet. Bij inloggen doet die server dan een challenge-response controle om zeker te weten dat de binnenkomende gebruiker/client op zijn lokale computer de bijbehorende private key heeft. Dat is veiliger en sneller dan een ‘gewoon’ login-password.

13 | Man-in-the-Middle attacks

De Diffy-Hellman techniek is kwetsbaar voor **man in the middle** aanvallen door een (proxy-)server die ergens in het pad tussen client en beoogde server in zit. Herinner je nog even dat Diffy-Hellman session keys van het symmetrische type zijn.

Laten we die drie partijen C (client), M (middle man) en S (server) noemen. Als C en S beginnen met het uitwisselen van de brokstukken informatie dan vangt M dat verkeer van beide kanten af. Zoals we hebben verteld bij de uitleg van Diffy-Hellman is op dit moment het verkeer nog niet geëncrypt, dus het is gewoon af te luisteren. Naar beide kanten toe antwoordt M met zelfgemaakte brokken informatie, die uiteraard wel compatibel zijn met het Diffy-Hellman algoritme. Zo ontstaan **twee** afzonderlijke session keys: een tussen C en M, en een tussen M en S. Data die bij M binnenkomen langs de ene kant kan hij decrypten met die ene session key, en moet hij weer met de andere session key encrypten voordat hij ze naar de uiteindelijke bestemming doorstuurt.

Ssh-verkeer begint met een Diffie-Hellman exchange over een niet-geëncrypte verbinding, en is dus in principe kwetsbaar voor zo'n MitM attack. Als de ssh-gebruiker vanaf C inlogt bij S door een password te typen, dan kan dat gelezen worden door M, en zijn alle deuren open. Echter, als de gebruiker vanaf C inlogt via de 'authorized_keys' techniek dan verwerkt het ssh-protocol de session key als data mee in de challenge-response boodschappen waarmee dat inloggen wordt gecontroleerd. Die boodschappen worden geëncrypt met het public/private keypair van de gebruiker, en daar heeft M geen greep op. Op dat moment valt M alsnog door de mand omdat C en S ontdekken dat ze niet dezelfde session key hebben.

Ook HTTPS/TLS verbindingen genereren een session key voor hun dataverkeer, maar daar is een belangrijk verschil: de client heeft uit het certificaat al de public key van de server. Daarmee kan hij vanaf het allereerste moment reeds het verkeer naar de server encrypten, dus nog voordat de session key gebruiksklaar is. Encrypten in de andere richting kan nog niet vanaf het allereerste begin. Toch is dit al voldoende om de middle man buiten spel te zetten. want die kan het verkeer in elk geval in die ene richting niet vanaf het begin afluisteren.

Overigens: er zijn ook tools beschikbaar om in een HTTPS/TLS context te werken met een Man-in-the-Middle proxy. Maar die vereisen dat de client speciaal wordt geprepareerd om een certificaat van de proxy (dus diens public key) te accepteren alsof het het certificaat van de eigenlijk bedoelde server is. Dat prepareren kan gebeuren door de leverancier van die client, bijvoorbeeld als dat een special purpose softwareprodukt is. Het kan ook doordat de IT-afdeling van je bedrijf de built-in lijst van root-CA's uitbreidt voordat je de client kunt/mag gebruiken. En het kan doordat je op een of andere manier wordt verleid om in je client een fout certificaat te accepteren.

Geavanceerde HTTPS-afluistertools, zoals bijvoorbeeld overheden gebruiken om hun burgers te controleren, werken ook ongeveer op deze MitM manier. Zo'n overheid zorgt dat in de browser-built-in lijst een CA wordt opgenomen waarvan zij de private key kent. Als dan een URL door de afluister-proxy passeert, wordt daar on the fly een certificaat voor die website gegenereerd. Feitelijk is het certificaat dus vals. Uiteraard wordt niet gekeken naar een eventueel CAA-record in de DNS, zoals

we besproken hebben in de paragraaf over ‘Certificate revocation’. Maar doordat het in de browser matcht op een van root-CA’s is de browser toch tevreden.

14 | Key agent

Eerder in dit verhaal hebben we gezegd dat een private key streng geheim moet blijven. Hij mag nooit jouw eigen machine verlaten. Maar als je een machine gebruikt die niet van jezelf is, hoe scherm je dan de inhoud van je private key file af voor de administrator van die machine?

De software waarmee je een keypaar genereert heeft meestal de mogelijkheid om de private key te versleutelen met een **pass phrase**. Maar dat betekent wel dat iedere keer als die private key dienst moet doen, jij opnieuw die passphrase moet intypen. Dat is onhandig.

Met name `ssh` heeft daarvoor een mooie oplossing: de **ssh-agent**. Dat is een apart programma dat wordt gestart als jij inlogt. Het vraagt aan jou de passphrase van je private key, ontsleutelt de key, en bewaart die in zijn geheugen. Als binnen jouw sessie een ander programma de key nodig heeft dan zal dat programma eerst kijken of het jouw **agent**-programma kan aanspreken. Lukt dat, dan hoef je niet meer zelf (opnieuw) in actie te komen met je pass phrase. Pass phrase intypen is nu nog maar een keer per sessie nodig. Zo'n ssh-agent wordt wel een 'personal daemon' genoemd. Variaties op dit ssh-agent thema vind je ook in andere omgevingen waar private keys worden gebruikt.

15 | Web of trust

Naast certificaten bestaat nog een andere manier waarop een partij een soort garantie kan krijgen dat een ontvangen public key inderdaad afkomstig is van de juiste verzender. Die manier werkt zonder ‘authority’ en heet **Web of Trust** (WoT). Dit wordt o.a. gebruikt in GPG (**GNU Privacy Guard**) waarmee je bestanden kunt encrypten om ze bijv. als email-attachment te versturen. GPG is een implementatie van de OpenPGP standaard, waarbij PGP staat voor ‘Pretty Good Privacy’.

Aan het begin van dit verhaal hebben we gezien dat twee partijen die elkaar berichten willen sturen elk een eigen keypaar moeten hebben. Als jij berichten ontvangt dan heeft de andere partij die geëncrypt met jouw public key. Voor decrypten gebruik je je eigen private key; die heb je zelf gemaakt dus die kun je vertrouwen. Maar als je berichten wil encrypten voor **verzending** dan heb je de public key van de ontvangende partij nodig. Die key moet je van tevoren ergens hebben opgehaald. Web of Trust is bedoeld om jou zekerheid te geven dat die public key inderdaad afkomstig is van de bedoelde andere partij, en dat er niet mee is gerommeld voordat jij ‘m in handen kreeg.

Bij het maken van een GPG-keypair wordt een identificatie van de maker -meestal zijn email adres- aan de public key gekoppeld. De maker kan daarna zijn public key op een **public key server** plaatsen, en iedereen die ‘m nodig heeft kan ‘m downloaden. Maar hoe weet de ontvanger dat die key niet vervalst is?

Voor dat doel kan de maker zijn public key laten **signeren** door mensen uit zijn kennissenkring. Dat kan op individuele basis worden geregeld, en soms worden ook **key signing parties** georganiseerd, bijvoorbeeld tijdens congressen. Dat betekent dus dat zo’n public PGP-key de digitale handtekeningen met zich meedraagt van andere mensen die daarmee zeggen: “ik kan je verzekeren dat deze public key inderdaad van mijn kennis Piet is”.

Als jij iemand een geëncrypt bericht wil sturen dan haal je zijn public key van een keyserver, of je vraagt of hij zijn key wil opsturen. Die key moet je daarna toevoegen aan jouw ‘sleutelbos’ (key ring); dat wordt **importeren** genoemd. Als je op dat moment die key al voldoende vertrouwt dan kun je vervolgens je gang gaan met het encrypten en versturen van je bericht.

Maar bij het importeren moet je ook een oordeel geven over hoe zorgvuldig die key-eigenaar, volgens jou, zijn vrienden kiest. Anders gezegd: je moet een oordeel geven over hoe zorgvuldig hij, volgens jou, de identiteit controleert van een derde persoon die hem een key-signing verzoek voorlegt. Dat gaat op een schaal met een aantal keuzen zoals ‘ik weet het niet’, ‘hij is een allemansvriend’, tot ‘ik vertrouw hem blindelings’. Dat oordeel gaat pas later een rol spelen, als je nog meer andere keys gaat ontvangen. Het blijft vertrouwelijk binnen jouw sleutelbos. Na verloop van tijd bevat je GPG-sleutelbos dus meerdere sleutels, met bij elke sleutel een oordeel van jou over de eigenaar van die sleutel.

Stel dat jij later weer een nieuwe sleutel ontvangt en importeert. Op dat moment wordt gekeken of mede-ondertekenaars van die nieuwe sleutel al in jouw sleutelbos aanwezig zijn. Wordt inderdaad zo iemand gevonden, en heb jij van die persoon indertijd genoteerd dat je ‘m blindelings vertrouwt, dan wordt de nieuwe sleutel ook meteen geaccepteerd. De nieuwe sleutel wordt ook geaccepteerd als er al

drie andere sleutels aan je bos zitten waarvan je de eigenaren als ‘matig vertrouwd’ hebt gelabeld. En zo zijn er nog meer combinaties die het accepteren van de nieuwe sleutel laten afhangen van wat jij al hebt geoordeeld over degenen die die nieuwe sleutel mede hebben ondertekend, voor zover de eigen sleutels van die ondertekenaars al eerder in jouw sleutelbos zijn opgenomen. Web-of-Trust implementeert dus een soort “de vrienden van mijn vrienden zijn ook míjn vrienden”.