1. CRIPTOGRAFIE SI CERTIFICATE DIGITALE

1.1. Concepte

1.1.1. Probleme de securitate intr-o transmisiune de date si rezolvarea lor

Consideram scenariul in care doua parti incearca sa-si transmita date una alteia, si problemele care pot afecta acest dialog.

Problema	Avem nevoie de	Rezolvare
Un atacator intra in posesia informatiei transmise intre cele doua parti ("eavesdropping")	Confidentialitatea informatiei	Criptarea informatiei
Una dintre parti se prezinta ca fiind altcineva (uzurpare de identitate – "impersonation")	Dovedirea identitatii interlocutorilor	Autentificarea partilor prezente in dialog
Un atacator modifica continutul mesajelor (chiar daca nu le intelege!) in drumul lor intre sursa si destinatie ("tampering")	Detectarea modificarilor aduse mesajului original	Trimiterea unei informatii de control semnate digital impreuna cu mesajul
Expeditorul neaga, in mod mincinos, ca a trimis un mesaj	Imposibilitatea ca expeditorul sa repudieze mesajul	Folosirea de semnaturi si certificate digitale

Cel mai avantajos scenariu pentru un atacator este acela de Man-in-the-middle (MITM). In acest scenariu, atacatorul se interpune intre cele doua parti ce comunica si poate captura si/sau altera mesajele schimbate de catre acestea, posibil fara ca partile sa fie constiente de acest lucru. Vom vedea in continuare cum criptografia ne ajuta sa anihilam avantajul pe care i-l confera atacatorului o astfel de pozitie privilegiata.

1.1.2. Criptare si decriptare

1.1.2.1. Concepte fundamentale

Criptarea este procesul de transformare a informatiei cu scopul de a nu fi inteleasa decat de catre destinatar. Aceasta presupune existenta unui element secret, cunoscut numai de catre cele doua parti ce comunica si inaccesibil potentialului atacator. La origini, acest element era algoritmul folosit pentru criptare. S-a dovedit insa ca, odata ce algoritmul secret a fost determinat de catre atacator, toate comunicatiile trecute si prezente sunt compromise, iar schimbarea algoritmului este costisitoare. De aceea, in ziua de astazi, algoritmii sunt publici, dar sunt parametrizati prin existenta uneia sau a doua chei (simple numere), care devin noul element secret. Expeditorul cripteaza informatia folosind cheia sa si numai cel ce cunoaste cheia pereche o poate decripta (in mod normal, acesta este doar destinatarul). Acest mod de operare permite schimbarea mai usoara a elementului secret, ducand in final la o mai buna securitate a informatiei.

In functie de relatia dintre cele doua chei (cea folosita la criptare si la decriptare) impartim algoritmii criptografici in:

- **algoritmi cu cheie simetrica** – sunt aceia in care cheia de criptare este aceeasi cu cea de decriptare
- **algoritmi cu cheie asimetrica** – cheia de decriptare difera de cea folosita pentru criptare

1.1.2.2. Algoritmi simetrici

In cazul algoritmilor simetrici, expeditorul si destinatarul cunosc o cheie secreta comuna – asa-numitul "shared secret". Expeditorul cripteaza informatia folosind aceasta cheie, iar destinatarul va folosi aceeasi cheie (numeric vorbind) pentru a decripta informatia.

Dezavantajul acestei clase de algoritmi este ca cheia secreta trebuie sa ajunga cumva in posesia celor doua parti care comunica – si numai a lor! – inaintea debutului comunicatiei, chiar si in conditiile in care cele doua nu au mai comunicat anterior (spre exemplu, cazul unui browser care se conecteaza pentru prima data la un server web, conexiunea intre ele trebuind criptata). Exista cel putin trei mecanisme de distributie a cheii:

1. cheia este cunoscuta anterior comunicatiei de catre cele doua parti – "pre-shared secret". Acest lucru se realizeaza prin interventie administrativa – spre exemplu, prin configurarea manuala a aceleiasi chei pe clientul si pe serverul care vor comunica
2. cheia este construita prin concursul ambelor parti, prin schimb de date, fara a o transmite insa ca atare pe canalul de comunicatie (ex: algoritmul Diffie-Hellman). Datele schimbate intre cele doua parti sunt insuficiente potentialului atacator pentru a reconstitui cheia. Transferul datelor confidentiale are loc abia dupa incheierea procedurii de generare a cheii comune
3. cheia simetrica este generata pe loc, in mod aleator, de catre una dintre parti, criptata folosind un algoritm asimetric si transmisa celeilalte parti, urmand ca apoi comunicatia sa se desfasoare folosind cheia simetrica

Un alt dezavantaj al algoritmilor simetrici este ca nu sunt potriviti pentru securizarea comunicatiei in cadrul unui grup:

- daca definim o singura cheie pentru intregul grup:
  - oricine poate decripta mesajele trimise de alt membru al grupului (asadar nu pot exista conversatii private intre perechi de membri)
  - mesajele nu sunt autentificate - oricine poate genera mesaje in numele altcuiva sau poate sa repudieze trimiterea unui mesaj propriu
- daca definim cate o cheie pentru fiecare pereche de membri ai grupului, scalabilitatea este compromisa

Dezavantajele mentionate sunt compensate de un avantaj major: algoritmii simetrici sunt rapizi, fiind potriviti pentru criptarea/decriptarea de cantitati mari de informatie. De aceea protocoalele de retea care transporta date criptate ii prefera in detrimentul celor asimetrici (care sunt mai lenti, insa au alte puncte forte, fiind indispensabili cand vine vorba de autentificare).

Iata cateva exemplu de algoritmi simetrici:

- **DES** – un algoritm vechi si vulnerabil, de evitat
- **3DES** – un algoritm care rezolva problema dimensiunii reduse a cheii din DES aplicandu-l pe acesta de 3 ori pentru fiecare bloc de date. In acest fel s-a realizat cresterea nivelului de securitate fara a gandi un algoritm complet nou
- **Blowfish** – un algoritm public gandit ca inlocuitor pentru DES; a fost folosit mult in criptarea datelor din retele Wireless (ca parte a WEP)
- **IDEA** – algoritm initial proprietar (devenit intre timp public), gandit de asemenea ca inlocuitor pentru DES
- **RC**{2,4,5,6} – o serie de algoritmi foarte rapizi, dar proprietari
- **AES** (Rijndael) – cel mai folosit algoritm simetric al zilelor noastre, ales prin concurs de catre NIST

1.1.2.3. Algoritmi asimetrici

Iata principalele proprietati ale acestei clase de algoritmi:

- fiecare dintre cele doua parti comunicante dispune de o cheie publica si una privata. Ele se genereaza astfel incat sa se afle intr-o anumita relatie matematica (ce depinde de algoritmul folosit)
- cheia publica se distribuie partenerilor de comunicatie; cea privata trebuie sa ramana numai in posesia entitatii pentru care a fost generata. Publicarea sa duce la compromiterea comunicatiei
- cheia publica se poate deduce din cea privata, dar nu invers
- **datele criptate cu cheia publica pot fi decriptate doar cu cea privata,** iar in cazul unora dintre algoritmi (ex:RSA) relatia functioneaza si invers. De aceea, destinatarul isi face cunoscuta cheia

publica, si oricine doreste sa-i trimita date le va cripta folosind aceasta cheie. In acest fel, singurul care le poate decripta este destinatarul (rol de asigurare a confidentialitatii)
- un alt mod de utilizare, in cazul RSA: daca expeditorul cripteaza date folosind cheia sa privata, oricine are cheia publica le poate decripta, insa stie sigur ca au provenit de la expeditor (rol de autentificare)
- dezavantajul acestui tip de algoritmi este ca sunt mult mai lenti decat cei cu cheie simetrica, de aceea in general ei sunt folositi numai pentru stabilirea unei chei simetrice de comunicatie intre cele doua parti, ulterior criptarea facandu-se cu un algoritm ce foloseste aceasta cheie simetrica
- exemple de algoritmi:
  - RSA, folosit pentru criptare/decriptare, autentificare si semnaturi digitale
  - ElGamal – folosit pentru criptare/decriptare, de obicei in conjunctie cu DSA
- criptografia cu cheie publica cuprinde si alti algoritmi cu cheie asimetrica, care insa nu sunt folositi pentru asigurarea confidentialitatii. Un exemplu este DSA:
  - gandit numai pt semnatura digitala
  - functioneaza in conjunctie cu SHA-1
  - este mai rapid ca RSA la generarea cheilor si a semnaturii, insa mai lent la verificarea de semnatura

Criptarea informatiei rezolva problema confidentialitatii, insa nu protejeaza impotriva atacurilor in care informatia este modificata/corupta in tranzit si nici impotriva celor de uzurpare a identitatii. Nici una dintre cele doua parti care comunica nu are de unde sa stie daca cel care ii trimite (sau catre care trimite) date este cel ce se pretinde a fi.

1.1.2.4. Chei simetrice vs asimetrice

O cheie simetrica si una asimetrica de aceeasi lungime (in biti) ofera un nivel de securitate foarte diferit. Spre exemplu, o cheie simetrica de 128 de biti este considerata a avea o rezistenta criptografica echivalenta cu cea a unei chei asimetrice de 3072 de biti, iar o cheie simetrica de 256 de biti echivaleaza cu una asimetrica de 15000 biti sau mai mult! (vezi tabelul alaturat)

Simetric	Asimetric
80	1024
112	2048
128	3072
256	15360

Cheile asimetrice sunt mai lungi deoarece presupun gasirea unor numere prime mari aflate intr-o anumita relatie – altfel spus, spatiul de chei este rarefiat: nu orice numar poate fi cheie asimetrica. In schimb, in cazul cheilor simetrice, aproape orice numar poate fi folosit pe post de cheie. Lungimea superioara a cheilor asimetrice este una dintre cauzele care fac algoritmii asimetrici mai lenti si deci mai costisitori.

1.1.3. Verificarea integritatii informatiei

1.1.3.1. Principii

Verificarea integritatii informatiei se realizeaza prin transmiterea, impreuna cu mesajul (informatia utila), a unei informatii de control calculate pe baza mesajului. Este de dorit ca informatia de control sa aiba dimensiuni cat mai reduse, pentru o cat mai buna eficienta a canalului de comunicatie. Destinatarul receptioneaza mesajul si informatia de control; el va efectua propriul calcul al informatiei de control pe baza mesajului si va confrunta rezultatul cu informatia de control primita impreuna cu mesajul. Identitatea celor doua atesta integritatea mesajului primit.

Atunci cand alegem o combinatie de resurse criptografice ce serveste unui anumit scop, trebuie sa ne punem in pozitia atacatorului si sa constientizam ce elemente ii sunt necesare acestuia pentru a ne anihila protectia. Daca am incerca sa folosim pentru detectia integritatii informatiei un algoritm simplu (sa spunem, o suma de control), singurul lucru de care ar avea nevoie atacatorul pentru a falsifica un mesaj interceptat este algoritmul folosit; cu alte cuvinte, ne aflam in scenariul – eliminat acum mult timp – al algoritmului folosit pe post de element secret.

De aceea este necesar ca informatia de control sa fie generata nu numai pe baza mesajului (pe care atacatorul il receptioneaza oricum) ci si pe baza unui element secret, care sa puna atacatorul in imposibilitatea de a calcula o informatie de control corecta pentru un mesaj falsificat. Cu alte cuvinte, informatia de control trebuie sa fie *autentificata*.

Nota: *informatia de control neprotejata (calculata strict pe baza mesajului) este deseori denumita MIC - Message Integrity Code.*

1.1.3.2. Algoritmi MAC (Message Authentication Code)

Numit si "message tag", MAC-ul este o informatie de control care, adaugata mesajului transmis, permite autentificarea si verificarea integritatii acestuia. MAC-ul se obtine aplicand mesajului un algoritm criptografic avand ca parametru o cheie secreta, rezultand o informatie de control calculata de asa natura incat o schimbare cat de mica a mesajului sa duca la schimbari (pe cat posibil majore) ale MAC-ului:

MAC = f (mesaj, cheie)

Cheia secreta este cunoscuta doar de catre expeditor si destinatar. Destinatarul calculeaza si el MAC-ul mesajului primit folosind aceeasi cheie si, daca MAC-ul calculat corespunde cu cel receptionat, deduce ca 1) mesajul nu a fost alterat in tranzit si 2) a fost generat de catre expeditor, fiindca doar acesta se afla in posesia cheii secrete.

Atentie! *Criptarea mesajului si prezenta MAC-ului in mesaj sunt aspecte independente! E posibil ca mesajul sa fie transmis in clar (necriptat), alaturandu-i-se MAC-ul. In acest fel destinatarul poate verifica integritatea si autenticitatea datelor, insa confidentialitatea acestora NU este asigurata.*

Ca algoritmi pentru generarea MAC-ului se pot folosi algoritmi de criptare simetrici (ex: DES, AES) sau algoritmi de hashing parametrizati cu cheie (ex: MD5, SHA-1).

Observatie: cheia secreta fiind aceeasi pentru expeditor si destinatar, algoritmii MAC sufera de aceleasi neajunsuri legate de distributia cheii ca si algoritmii simetrici.

1.1.3.3. Algoritmi de hashing

Un algoritm de hashing (numit si "one-way hash" sau "message digest") este o functie prin intermediul careia orice input este transformat, in mod ireversibil, intr-un sir de biti de lungime fixa (asa-numitul "hash"). Algoritmul genereaza asadar un fel de suma de control, insa una cu proprietati speciale, dupa cum urmeaza:

- hash-ul are lungime fixa, indiferent de lungimea datelor de intrare
- acelasi input duce de fiecare data la acelasi hash
- nu exista nici un fel de indicii care sa ajute la deducerea input-ului pornind de la hash - daca se schimba un bit din input, se schimba multi biti din hash (jumatate, in medie)
- pentru un mesaj de intrare dat, este extrem de greu de gasit un al doilea mesaj, diferit de primul, care sa genereze acelasi hash (obs: iar gasirea sa se faca mai repede decat simplul brute-force). O extensie a acestei conditii este rezistenta la coliziuni: sa nu existe nici o posibilitate de micsorare a efortului necesar gasirii a doua mesaje care au acelasi hash

Avand in vedere aceste proprietati, algoritmii de hashing sunt folositi in general pentru verificarea identitatii a doua seturi de date. In cazul transmisiilor de date, destinatarul doreste sa se asigure ca datele primite sunt identice cu cele trimise. Pentru aceasta, expeditorul trimite impreuna cu mesajul hash-ul acestuia. Destinatarul

calculeaza si el hash-ul mesajului si, daca este identic cu cel primit de la expeditor, deduce ca mesajul nu a fost alterat. Intalnim acest mod de validare, de exemplu, in cazul arhivelor software descarcate de pe internet. De obicei, autorul sau firma distribuitoare pune la dispozitia utilizatorilor o suma de control (MD5 sau SHA-1) ce reprezinta hash-ul datelor si care permite utilizatorului sa faca verificarea integritatii arhivei.

O alta utilizare des intalnita a algoritmilor de hashing este validarea parolelor la autentificarea utilizatorilor. De exemplu, in Unix-uri, in baza de date cu parole (/etc/shadow) sunt stocate numai hash-urile parolelor (astfel incat, chiar daca un atacator intra in posesia acestui fisier, sa nu aiba o modalitate de a afla parolele). Cand un utilizator doreste sa se autentifice, functia de autentifcare a sistemului de operare calculeaza hash-ul parolei introduse si il compara cu cel memorat in baza de date. Daca cele doua hash-uri coincid, se deduce ca utilizatorul a batut parola corecta. (remarca: in cazul parolelor, algoritmul de hashing este parametrizat prin intermediul unui "salt" (sare) – o secventa de biti random, aleasa pe loc; in caz contrar, aceeasi parola ar genera acelasi hash de fiecare data, pe orice sistem ar fi setata)

O a treia utilizare a algoritmilor de hashing se bazeaza pe proprietatea acestora de a crea un output "aleatorizat" (proprietatile 3 si 4 de mai sus). Algoritmii de hashing sunt folositi pentru crearea unor chei sigure pornind de la un input al utilizatorului. De exemplu, la memorarea intr-un fisier a unei chei private, utilizatorului i se solicita o parola ce va fi folosita apoi pentru generarea, prin hashing, a unei chei de criptare a fisierului.

Date fiind proprietatile lor, algoritmii de hashing pot fi folositi pentru generarea informatiei de control folosita pentru verificarea integritatii mesajului. Ei nu pot fi insa folositi ca atare: neexistand nici un element secret in comunicatia dintre cele doua parti, daca un atacator intercepteaza datele si le inlocuieste, recalculand in acelasi timp si hash-ul, destinatarul nu are cum sa-si dea seama de aceasta alterare. Este necesara introducerea unui element secret in calculul hash-ului, care sa puna atacatorul in imposibilitatea de a genera un hash valid pentru un mesaj alterat.

Rezolvarea vine sub forma asa-numitelor "keyed hashes" – algoritmi de hashing parametrizati prin folosirea unei chei. In acest fel, algoritmul de hashing poate fi folosit pentru generarea unui MAC, procedeul purtand denumirea de HMAC (Hash-based Message Authentication Code). De remarcat insa ca nu este inca rezolvata problema non-repudierii – oricine se afla in posesia cheii secrete poate genera MAC-ul mesajului, inclusiv destinatarul.

Cei mai cunoscuti/folositi algoritmi de hashing din ziua de astazi sunt:

- **MD5** (Message Digest 5) – genereaza un hash de 128 biti. Este considerat in ziua de astazi ca fiind nesigur din punct de vedere criptografic si se recomanda a nu se folosi ca parte a unui nou produs ce utilizeaza resurse criptografice. Exista si variante mai vechi ale algoritmului (ex: MD4) dar sunt nesigure
- **SHA-1** (Secure Hash Algorithm) – genereaza un hash de 160 de biti. Desi mai sigur decat MD5 (fie si prin lungimea hash-ului generat), si in SHA-1 au fost descoperite vulnerabilitati care, desi inca nu sunt fatale, indreptatesc migrarea catre algoritmi mai siguri
- **SHA-2** – reprezinta o suita de patru algoritmi (SHA-224, SHA-256, SHA-384, SHA-512), denumirile reflectand lungimile hash-urilor generate
- **SHA-3** – este inca in dezvoltare in momentul scrierii acestui material. Competitia pentru noul algoritm SHA-3 este inca deschisa, estimandu-se ca alegerea castigatorului va avea loc in 2012
- **RIPEMD**-160 – genereaza hash de 160 de biti. Este un algoritm de hashing mai putin folosit decat seria MD sau SHA. Exista si variante RIPEMD care genereaza hash de 128, 256 si 320 de biti

1.1.3.4. Semnaturi digitale

Precum s-a vazut in sectiunea anterioara, hash-ul ajuta la verificarea integritatii datelor din mesaj, insa ramane urmatoarea problema: cum poate fi sigur destinatarul ca hash-ul este cel original, calculat de catre expeditor?

MAC-ul este o posibila solutie, insa presupune ca cele doua parti sa impartaseasca un element secret comun ("shared secret"), ceea ce nu intotdeauna este practic sau posibil.

O semnatura digitala reprezinta echivalentul algoritmilor MAC realizat cu algoritmi asimetrici, folosind keyed hashes. Expeditorul calculeaza hash-ul mesajului si apoi il cripteaza folosind cheia sa *privata*. In acest fel, destinatarul nu poate decripta hash-ul decat cu cheia publica a expeditorului, asigurandu-se astfel de identitatea acestuia (daca mesajul a putut fi decriptat cu cheia publica a expeditorului inseamna ca a fost creat cu cea privata). Dupa decriptarea hash-ului, destinatarul calculeaza si el hash-ul mesajului si il compara cu cel decriptat. Daca se potrivesc, mesajul nu a fost alterat. Astfel, o semnatura digitala indeplineste acelasi rol ca un MAC.

Asadar, semnatura digitala reprezinta hash-ul mesajului original criptat cu cheia privata a expeditorului. Scopul semnaturii digitale este sa valideze mesajul insotit, avand in acelasi timp proprietatea ca numai expeditorul o poate genera. In acest fel, un atacator aflat in postura de man-in-the-middle, chiar daca ar putea modifica mesajul, nu ar putea genera semnatura corecta pentru acesta deoarece nu poseda cheia privata a expeditorului.

Observatie: *Diferenta dintre semnaturi digitale si MAC-uri este ca primele folosesc ca parametru al algoritmului o cheie asimetrica (cea privata), pe cand cele din urma folosesc o cheie simetrica (shared secret).*

Semnaturile digitale pot rezolva atat problema integritatii, cat si pe cea a autentificarii si a non-repudierii. Mai ramane insa o ultima problema: cum se poate asigura destinatarul ca se afla in posesia cheii publice corecte? Un atacator aflat in postura de man-in-the-middle ar putea purta doua conversatii independente cu cele doua parti, prezentandu-se fiecareia drept cealalta, avand astfel acces la informatie dupa bunul sau plac.

1.1.4. Autentificare

1.1.4.1. Certificate digitale, CA, PKI

In acest punct, este clar ca cele doua parti care intentioneaza sa comunice nu au cum sa fie 100% sigure de nimic daca nu incep de la un prim element care sa fie "trusted" (de incredere). Cineva trebuie sa-l asigure pe destinatar ca expeditorul este cel care se pretinde a fi, si invers. Aceasta terta entitate poarta numele de CA (Certification Authority) si este cea in care ambele parti au incredere.

CA-urile confirma identitatea cuiva care vrea sa transmita/primeasca date prin emiterea sau semnarea unui certificat digital, care in esenta leaga o identitate de o cheie publica (confirma faptul ca acea cheie apartine entitatii specificate). Este veriga lipsa in scenariul de pana acum: daca partea A primeste de la B un certificat digital eliberat de un CA in care A are incredere si in care se specifica faptul ca cheia publica a lui B este chiar aceea, A va fi acum sigur ca dialogheaza cu cine trebuie. Exemplu: un student vine la sala sa dea examenul final. Nu putem sti daca este cine pretinde a fi pana nu ne arata buletinul, in care sunt puse in corespondenta numele sau cu aspectul fizic (poza), iar in aceasta corespondenta avem incredere deoarece este confirmata de o a treia entitate in care avem incredere (Politia Romana).

Un CA este o firma de incredere care semneaza certificatele digitale ale altor firme/persoane, confirmand astfel faptul ca cheile publice continute in acele certificate corespund entitatilor respective. Firma/persoana beneficiara a certificatului depune o cerere de semnare de certificat (sub forma unui fisier CSR – Certificate Signing Request), iar CA face verificarile necesare si apoi semneaza certificatul folosind cheia sa privata si il returneaza beneficiarului.

Sa presupunem ca un client si un server incearca sa comunice. In prima faza, clientul va contacta serverul, cerandu-i sa-si dovedeasca identitatea. Serverul ii va trimite certificatul sau, semnat de un CA in care clientul

are incredere. Clientul are certificatul lui CA, asadar se afla in posesia cheii publice, si in consecinta este capabil sa verifice autenticitatea certificatului serverului.

CA-urile formeaza ierarhii. Exista asa-numitele **root CA** – autoritati al caror certificat este auto-semnat (*self-signed*) si care la randul lor semneaza certificatele altor CA-uri; acestea la randul lor pot semna certificatele altor CA-uri sau ale beneficiarilor directi (servere/clienti…) etc. In acest fel, relatia de incredere intr-un CA se extinde de-a lungul unui lant, iar a verifica validitatea unui certificat poate insemna parcurgerea lantului de certificate (*certificate chain*) pana se ajunge la un root CA.

Dar de unde stim ca putem avea incredere intr-un root CA, sau cine/ce dicteaza acest lucru? Exista multe softuri care permit emiterea si semnarea de certificate digitale, astfel ca este simplu de creat un certificat auto-semnat. Diferenta o face faptul ca sistemele de operare vin preconfigurate cu un set de "trusted root certificates" – certificate ale unor root CA care sunt considerate de incredere. Orice lant de certificate a carui verificare duce in final la un astfel de root CA va fi considerat valid. Exemple de root CA: Verisign, Thawte, Geotrust.

La modul general, este denumit **PKI (Public Key Infrastructure)** sistemul prin intermediul caruia se poate face analiza, confirmarea si dovedirea identitatii electronice a utilizatorilor/computerelor prin punerea lor in corespondenta cu chei publice. Sistemul include structura ierarhizata a CA-urilor impreuna cu functiile oferite de catre acestea: emiterea si revocarea certificatelor, verificarea identitatii celor care le solicita (persoana fizica/ juridica in cauza), posibilitatea de determinare a validitatii unui certificat deja emis de catre participantii la o transmisiune de date si autentificarea lor reciproca.

Nota: *alternativa la acest sistem este o solutie distribuita numita "web of trust", in care toti utilizatorii au certificate self-signed si care garanteaza unul pentru celalalt, fiecare utilizator cumuland un scor ce rezulta din cantitatea voturilor primite si de scorul votantilor insisi.*

1.1.4.2. Cum se obtine si ce contine un certificat

Un certificat leaga o cheie publica de o identitate (un asa-numit DN – Distinguished Name), asa cum buletinul leaga o persoana fizica de identitatea sa. Certificatul este emis de catre un CA in care participantii la schimbul de date au incredere.

Pentru a obtine un certificat semnat de un CA solicitantul trebuie sa genereze un CSR (Certificate Signing Request). Iata cum arata un CSR salvat intr-un fisier:

```
-----BEGIN CERTIFICATE REQUEST-----
MIIBNTCB4AIBADB7MQswCQYDVQQGEwJSTzESMBAGA1UECBMJQnVjdXJlc3RpMRQw
EgYDVQQKEwtJbmZvQWNhZGVteTEcMBoGA1UEAxMTd3d3LmluZm9hY2FkZW15Lm5l
dDEkMCIGCSqGSIb3DQEJARYVaW9udXRAaW5mb2FjYWRlbXkubmV0MFwwDQYJKoZI
hvcNAQEBBQADSwAwSAJBAMpx0pCPgFcusBxFe1dHguCAvZM25moXWkHv1Zw3Z0zi
pq2F8huCj3V0YhjMxhVD0cVmz1AKNJU6Lvp5ggzN08UCAwEAAaAAMA0GCSqGSIb3
DQEBBQUAA0EAOU6Wq2IjCVxME/wFLEYWY3lvOhuAo0r40hCp1C12P5Q8gvjSj8uK
yrLm30s8aCThpxk4GqWSjmXYTblw8wBwCg==
-----END CERTIFICATE REQUEST-----
```

…si varianta sa decriptata:

```
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: C=RO, L=Bucuresti, O=InfoAcademy,
CN=www.infoacademy.net/emailAddress=ionut@infoacademy.net
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
```

```
            RSA Public Key: (512 bit)
                Modulus (512 bit):
                    00:ca:71:d2:90:8f:80:57:2e:b0:1c:45:7b:57:47:
                    82:e0:80:bd:93:36:e6:6a:17:5a:41:ef:d5:9c:37:
                    67:4c:e2:a6:ad:85:f2:1b:82:8f:75:74:62:18:cc:
                    c6:15:43:d1:c5:66:cf:50:0a:34:95:3a:2e:fa:79:
                    82:0c:cd:d3:c5
                Exponent: 65537 (0x10001)
        Attributes:
        Requested Extensions:
            Netscape Cert Type:
                SSL Server
            X509v3 Subject Alternative Name:
                DNS:www.infoacademy.net, IP Address:11.22.33.44
    Signature Algorithm: sha1WithRSAEncryption
        39:4e:96:ab:62:23:09:5c:4c:13:fc:05:2c:46:16:63:79:6f:
        3a:1b:80:a3:4a:f8:d2:10:a9:d4:2d:76:3f:94:3c:82:f8:d2:
        8f:cb:8a:ca:b2:e6:df:4b:3c:68:24:e1:a7:19:38:1a:a5:92:
        8e:65:d8:4d:b9:70:f3:00:70:0a
```

In cerere sunt prezente:

- identitatea solicitantului (asa numitul *Distinguished Name* - DN), ce contine:
  - CN – Common Name, numele entitatii solicitante. Atunci cand certificatul este folosit, de exemplu, pentru a dovedi identitatea unui server web, CN de pe certificat trebuie sa corespunda cu numele cu care a fost accesat serverul (ex: *www.infoacademy.net* in cazul nostru)
  - C – tara (country)
  - L – Localitate
  - O – organizatie
- cheia publica a solicitantului
- eventuale extensii dorite de catre solicitant. In exemplul de mai sus, acesta doreste ca certificatul sa fie valid indiferent daca serverul sau este accesat folosind IP-ul sau numele DNS
- semnatura digitala a solicitantului aplicata certificatului (efectuata cu cheia privata a solicitantului)

Odata cererea semnata de catre CA, iata cum arata certificatul (in versiune decriptata):

```
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
            a0:8f:20:94:2b:81:eb:06
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=RO, L=Apoldu de Jos, O=Organizatia comunala pentru informatizare,
OU=Departamentul Certificate Digitale, CN=Apoldu CA
        Validity
            Not Before: Feb 22 11:05:53 2007 GMT
            Not After : Mar 24 11:05:53 2007 GMT
        Subject: C=RO, L=Bucuresti, O=InfoAcademy,
             CN=www.infoacademy.net/emailAddress=ionut@infoacademy.net
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (512 bit)
                Modulus (512 bit):
                    00:ca:71:d2:90:8f:80:57:2e:b0:1c:45:7b:57:47:
                    82:e0:80:bd:93:36:e6:6a:17:5a:41:ef:d5:9c:37:
                    67:4c:e2:a6:ad:85:f2:1b:82:8f:75:74:62:18:cc:
                    c6:15:43:d1:c5:66:cf:50:0a:34:95:3a:2e:fa:79:
                    82:0c:cd:d3:c5
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            Netscape Cert Type:
                SSL Server
            X509v3 Subject Alternative Name:
```

```
                DNS:www.infoacademy.net, IP Address:11.22.33.44
X509v3 Basic Constraints:
                CA:FALSE
    Signature Algorithm: sha1WithRSAEncryption
        2c:93:cf:e8:93:5b:5d:82:78:b7:bb:b2:7b:86:bc:8f:bb:2d:
        c7:2b:80:bb:1b:6f:82:4f:c8:88:dc:7e:e7:10:ee:7d:83:f9:
        53:70:d0:7e:5d:e4:e6:de:3c:cd:64:26:36:e8:14:e6:5b:7b:
        89:c6:34:8c:2d:4e:4e:49:48:9c
```

Observam aparitia catorva elemente noi:

- campul *Issuer* ce descrie semnatarul certificatului; tot a lui este si semnatura din final.
- campul *Serial Number* – este seria certificatului, folosita pentru a-l putea identifica in mod unic (spre exemplu, daca certificatul este revocat)
- campul *Validity* – specificat perioada de valabilitate a certificatului

CA-ul poate pastra extensiile cerute in CSR, le poate elimina total sau partial si poate introduce extensii proprii.

1.1.4.3. Extensii de certificat

Un CSR poate solicita una sau mai multe extensii, pe care CA-ul le poate apoi aproba sau nu. Fiecare extensie are un nume, caruia ii poate corespunde o valoare sau o secventa (o lista de sub-extensii). In cel de-al doilea caz, extensia “parinte” actioneaza ca un fel de director care contine la randul sau alte extensii.

O extensie poate fi desemnata ca fiind sau nu critica. Atunci cand o extensie critica nu este recunoscuta de catre software-ul ce valideaza un certificat, acesta din urma va fi considerat invalid.

Prezentam aici cateva extensii utile:

- **basicConstraints** – este o extensie de tip secventa; cea mai importanta componenta a sa este **CA**, ce poate avea valoarea true sau false. Aceasta componenta indica daca certificatul va putea fi sau nu folosit pentru a semna la randul sau alte certificate. Pentru un certificat al unui CA, valoarea trebuie sa fie true; pentru unul de server, valoarea va fi false
- **keyUsage** – este de asemenea o extensie de tip secventa. Daca este prezenta in certificat, ea trebuie desemnata ca critical. Valoarea sa este o succesiune de biti, fiecare dintre ei corespunzand unei utilizari posibile a cheii cuprinse in certificat. Bitii setati determina operatiile pentru care poate fi folosit certificatul. Lista cuprinde valorile: digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment, keyAgreement, keyCertSign, cRLSign, encipherOnly si decipherOnly. Spre exemplu, pentru certificatul unui CA avem nevoie de keyCertSign si cRLSign, iar pentru un certificat de server este necesar keyEncipherment.
- **extKeyUsage** (extended key usage) – extensie de tip secventa, care specifica utilizarile permise ale cheii. Printre variantele posibile se numara serverAuth, clientAuth, codeSigning, emailProtection
- **subjectAltName** (alternative name) – extensia are ca valoare o insiruire de nume suplimentare. Aceasta extensie permite, spre exemplu, accesarea unui server folosind diferite nume DNS (ex: o inregistrare A si un CNAME care pointeaza catre el) sau chiar adrese IP

1.1.5. Formate uzuale pentru stocarea informatiilor criptografice

Certificatele digitale, CSR-urile si cheile private sunt in general memorate in fisiere – iata cateva scenarii care dicteaza aceasta necesitate:

- la generarea unei chei private, CSR sau certificat, informatia generata este plasata intr-un fisier in scopul distribuirii/folosirii ulterioare
- un fisier cu un format standardizat este o solutie buna pentru transferul informatiilor de natura criptografica intre aplicatii (ex: exportarea/crearea unui certificat si importarea sa in alta aplicatie)

- multe dintre aplicatiile care folosesc resurse criptografice le primesc sub forma de fisiere (ex: pentru a-si autentifica/cripta comunicatiile folosind SSL/TLS, un server are nevoie de un certificat digital si de cheia privata corespunzatoare, plasate de obicei in doua fisiere).

Exista doua modalitati de reprezentare posibile pentru un certificat/CSR/cheie privata in cadrul unui fisier:

- binar – continutul criptografic este reprezentat sub forma unei succesiuni de octeti, fara vreo preocupare pentru felul in care arata fisierul intr-un viewer de text. Aceasta abordare corespunde codarii de tip DER
- ASCII – continutul criptografic este codat BASE64 astfel incat continutul fisierului sa conste exclusiv din caractere printabile. Fisierul debuteaza cu o linie de forma -----BEGIN *tip resursa*----- si se incheie cu o linie de forma -----END *tip_resursa*----- , unde *tip_resursa* poate fi CERTIFICATE REQUEST, RSA PRIVATE KEY, CERTIFICATE etc. Aceasta abordare este cea a codarii PEM

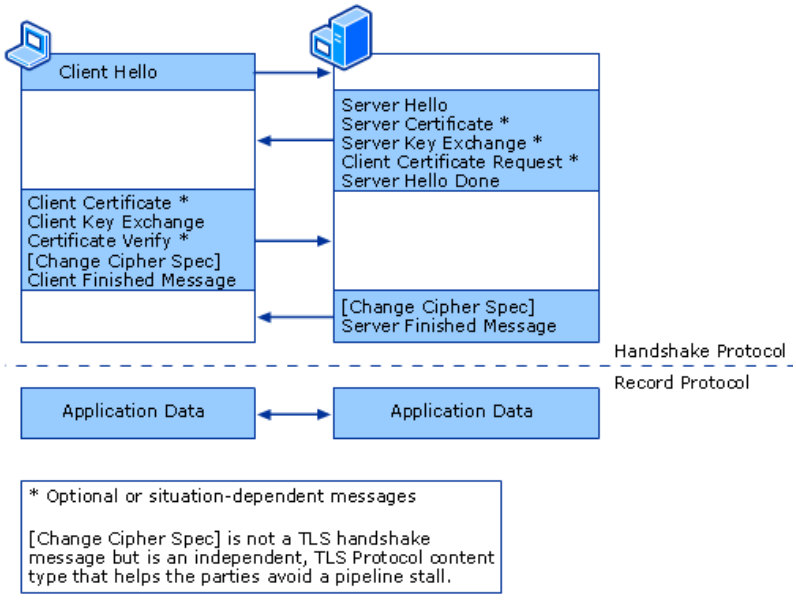
Atentie! A nu se confunda codarea folosita in cadrul fisierului cu criptarea sa! Codarea este usor reversibila si nu este folosita in scopul protectiei informatiei. Atunci cand informatia este confidentiala, continutul poate fi criptat folosind un algoritm simetric.

In practica se intalnesc mai multe formate standardizate de fisier in care “circula” resursele criptografice:

- .**pem** – sunt fisiere in care resursele sunt codate PEM. Pot contine chei, CSR-uri sau certificate si pot fi sau nu criptate. Un singur fisier poate contine multiple resurse (ex: mai multe certificate, sau o cheie si un certificat)
- .**der** – sunt fisiere binare in format DER, ce pot contine un singur certificat/cheie, folosite in general in lumea Java
- .**csr** – contine un Certificate Signing Request
- .**crt** – folosit pentru a memora certificate digitale (in format PEM sau DER), folosit preponderent in Unix si Linux
- .**cer** – asemanator cu CRT dar folosit preponderent in lumea Microsoft
- .**key** – folosit pentru a memora chei asimetrice (privata/publica)
- .**pfx**, .**pkcs12** sau .**p12** – un fisier in format PKCS12, care contine un certificat si cheia privata corespondenta
- .**crl** – folosit pentru a memora un Certificate Revocation List (vezi mai jos capitolul despre CA-uri)

1.2. Protocolul SSL/TLS

Protocolul SSL (Secure Sockets Layer) a fost initial gandit de catre inginerii Netscape pentru a securiza comunicatia intre doua aplicatii. Acest protocol functioneaza deasupra nivelului transport, folosindu-se de serviciile TCP, si ofera servicii de autentificare, criptare si verificare a integritatii aplicatiilor aflate la nivelul superior in stiva TCP/IP.



SSL a evoluat ca protocol proprietar pana la versiunea 3, cand pe baza sa a fost elaborat un standard IETF. Standardul specifica protocolul TLS (Transport Layer Security), versiunea curenta a acestui protocol fiind 1.2. SSLv3 si TLSv1 sunt foarte asemanatoare, insa nu pe deplin compatibile.

In protocolul SSL, dialogul intre client si server debuteaza cu etapa de “handshake”, in care

acestia negociaza parametrii de securizare a comunicatiei si se autentifica: serverul se autentifica intotdeauna, clientul numai daca serverul i-o cere. Autentificarea foloseste criptare cu algoritmi asimetrici si se realizeaza de obicei pe baza de certificate digitale schimbate de cele doua parti. Scopul acestei prime parti este stabilirea unui element secret comun, din care va fi derivata o cheie simetrica. Odata cheia aflata in posesia ambelor parti, restul comunicatiei va fi criptat folosind un algoritm simetric ce foloseste aceasta cheie, deoarece criptarea intregii comunicatii folosind algoritmi asimetrici ar fi prea costisitoare.

Iata o descriere a etapelor handshake-ului:

- negocierea (comunicatia este deocamdata necriptata si neautentificata)
  - clientul trimite un mesaj ClientHello in care indica versiunea de TLS maxima suportata, lista de algoritmi criptografici suportati si un numar generat aleator
  - serverul alege cea mai buna suita de algoritmi criptografici pe care o are in comun cu clientul si raspunde printr-un mesaj ServerHello, care contine la randul sau un alt numar generat aleator
  - in acest fel au fost stabiliti viitorii parametri de securizare a canalului de comunicatie: mecanismul prin care cele doua parti vor ajunge in posesia cheii simetrice comune (“key exchange method”), modalitatea de criptare a datelor (“bulk encryption method”) si modalitatea de verificare a integritatii datelor (MAC-ul)
- autentificare
  - serverul trimite clientului certificatul sau, insotit eventual de lista certificatelor CA-urilor care garanteaza pentru acesta
  - clientul verifica certificatul serverului si, in caz de validitate, ii trimite acestuia un mesaj care contine PreMasterSecret (generat si el aleator) sau cheia sa publica
  - (optional) clientul trimite certificatul sau pentru a se autentifica fata de server
  - serverul si clientul folosesc cele doua numere aleatoare si eventualul PreMasterSecret pentru a genera un element secret comun (asa-numitul “master secret”). Acesta este folosit ca baza in generarea oricaror chei de comunicatie ulterioare
  - pana in acest punct comunicatia a fost criptata cu algoritmi asimetrici; de aici incolo se comuta pe simetric
- incheierea handshake-ului
  - clientul trimite serverului un mesaj ChangeCipherSpec care indica faptul ca, incepand din acel moment, dialogul va fi autentificat si criptat folosind parametrii negociati anterior
  - clientul trimite un mesaj ClientFinished, autentificat si criptat, care contine un hash al tuturor mesajelor trimise/primite pana atunci. Informatia este divizata in doua parti, fiecare fiind criptata cu cate un algoritm de hashing (MD5 si SHA1) si efectuandu-se apoi un XOR intre cele doua rezultate. Scopul este protectia impotriva situatiei in care unul dintre algoritmii de hashing se dovedeste vulnerabil
  - serverul raspunde cu un mesaj ChangeCipherSpec construit pe aceleasi principii
- dialogul continua in mod criptat & autentificat

Observam 3 etape distincte:

- etapa de negociere, fara criptare/autentificare
- etapa de autentificare, ce foloseste algoritmi asimetrici
- etapa de transfer de date, ce foloseste algoritmi simetrici.

1.3. OpenSSL

1.3.1. Prezentare si capabilitati

OpenSSL este numele celei mai folosite biblioteci de functii criptografice prezente sub Unix/Linux. Accesul la aceste resurse este asigurat de utilitarul numit chiar *openssl*. Formatul general al liniei de comanda este:

```
openssl comanda optiuni
```

Fiecare comanda dispune de propriul sau set de optiuni, o mica parte dintre optiuni fiind comune tuturor comenzilor. O mica parte dintre aceste comenzi isi pot citi optiunile dintr-un fisier de configurare (vezi mai jos).

Iata principalele comenzi de interes openssl:

- **dgst** – pentru folosirea algoritmilor de digest
- **enc** – pentru algoritmi simetrici
- **genrsa** – pentru generare de chei RSA
- **rsa** – pt vizualizare/manipulare chei RSA
- **dsaparam** si **dsa** – pentru generarea parametrilor si respectiv managementul cheilor DSA
- **dhparam** – pentru generarea parametrilor Diffie-Hellman
- **req** – pentru managementul cererilor de semnare de certificat (CSR – Certifcate Signing Request)
- **x509** – pentru managementul certificatelor (inclusiv semnare)
- **verify** – verificare de lanturi de certificate (certificate chains)
- **passwd** – generare de hash-uri de parole
- **ca** – pentru a actiona ca CA (Certification Authority). OpenSSL va mentine o evidenta a certificatelor emise si revocate
- **s_client** – foarte util in debugging. Joaca rolul de client SSL/TLS si permite conectarea la un server SSL/TLS, afisand pe ecran detalii despre conexiune (certificatul primit de la server, parametrii conexiunii) si varianta decriptata a datelor traficate si permitand astfel interactiunea intre utilizator si server. Ne putem astfel conecta la un server de mail securizat cu TLS (de exemplu) din linia de comanda si sa dam comenzi ca si cum ne-am afla intr-o sesiune telnet.

Iata cateva optiuni des folosite si disponibile in cazul multora dintre comenzile openssl:

- in mod normal, comanda openssl citeste din stdin si afiseaza in stdout; se poate modifica acest lucru cu optiunile **–in** si **–out** urmate de calea catre fisierul folosit ca sursa/destinatie a datelor.
- cheile si certificatele sunt stocate in fisiere sub forma unei succesiuni de numere. OpenSSL ofera posibilitatea interpretarii unui astfel de fisier si a afisarii continutului intr-un format lizibil, folosind optiunea **–text**. Prin folosirea suplimentara a optiunii **–noout** se suprima varianta numerica.

Cateva dintre comenzile openssl (ca, req si x509) folosesc un fisier de configurare, ce poate fi specificat ca optiune in linia de comanda. Optiunile corespunzatoare lor se gasesc in sectiuni ale fisierului de configurare ce poarta numele comenzii (ex: [x509]).

1.3.2. Criptare si decriptare folosind algoritmi simetrici

Criptarea si decriptarea se realizeaza folosind comanda **enc** urmata de optiuni care specifica operatia dorita (criptare sau decriptare) si algoritmul folosit. In cazul criptarii, datele sunt citite implicit din standard input; atunci cand dorim preluarea lor dintr-un fisier este necesara folosirea optiunii **-in**. Asemanator, in cazul decriptarii, rezultatul produs este afisat implicit pe ecran si pentru salvarea sa intr-un fisier trebuie folosita optiunea **-out**.

Dupa ce dam comanda de criptare ni se va solicita o parola; ea este folosita pentru generarea cheii simetrice de criptare necesara algoritmului.La decriptare ni se va cere aceeasi parola, deoarece pentru a decripta este folosita aceeasi cheie.

```
# criptarea lui fisier1 si plasarea rezultatului in fisier1.criptat
openssl enc –aes256 –in fisier1 –out fisier1.criptat
```

```
# operatiunea inversa, cu afisare pe ecran
openssl enc –d –aes256 –in fisier1.criptat

# decriptare cu salvare intr-un fisier
openssl enc -d -aes256 -in fisier1.criptat -out fisier1.decriptat
```

1.3.3. Verificarea integritatii

OpenSSL ofera resurse precum:

- **algoritmi de hashing neparametrizati** – putem genera hash-uri MDx, SHAx, RIPEMD-xxx etc. Openssl ofera in acest scop comanda **dgst** si, in plus, comenzi ce poarta numele diversilor algoritmi de hashing suportati. Putem alege dintre doua procedee de generare a unui hash:
  - folosim openssl dgst urmat de o optiune ce specifica algoritmul dorit. Daca informatia provine dintr-un fisier, acesta va fi precizat cu ajutorul optiunii -in
  - folosim openssl urmat de comanda cu numele algoritmului de hasing, caz in care eventualul fisier se va specifica sub forma de argument al comenzii (vezi exemplele)

```
# generare de hash MD5, in cele doua variante
openssl dgst –md5 fisier.text
openssl md5 fisier.text

# generare de hash SHA-1, in cele doua variante
openssl dgst –sha1 fisier.txt
openssl sha1 fisier.txt

# generare de hash SHA256, in ambele variante
openssl dgst -sha256 fisier.txt
openssl sha256 fisier.txt # incepand cu openssl 1.0.0!
```

Nota: *algoritmii de hashing din seria SHA-2 sunt suportati incepand cu openssl 0.9.8d.*

- **generare de hash-uri pentru parole** – facilitate utila pentru popularea automatizata a fisierelor cu useri/parole - fie cele de sistem (passwd/shadow), fie in cazul serverelor care folosesc autentificare din fisier (ex: Apache, Squid). Particularitatea fisierelor de acest fel este ca la generarea hash-ului se foloseste un asa-numit “salt” (sare) – o secventa numerica ce parametrizeaza algoritmul, astfel incat, chiar daca doi utilizatori au ales intamplator aceeasi parola, hash-ul ei sa nu dezvaluie acest lucru. Salt-ul este memorat in fisier impreuna cu hash-ul, astfel incat parola sa poata fi verificata la autentificarea utilizatorilor.

```
# optiunea -1 inseamna folosirea algoritmului MD5;exista si alti algoritmi posibili (DES,SHA-1..)
openssl passwd -1 –salt cetqVYAw parolamea

# rezultat: $1$cetqVYAw$8wq7MXlNfG7gNm6WUujH21 (format: $1$salt$hash – vezi /etc/shadow)
```

1.3.4. Algoritmi asimetrici

1.3.4.1. Principii generale

In cazul algoritmilor asimetrici fiecare dintre entitatile implicate in comunicatie dispune de o pereche de chei. Spre deosebire de cazul simetric, cand puteam alege la intamplare o cheie, aici cele doua chei trebuie sa se supuna unor restrictii si se afle intr-o anumita relatie matematica una cu alta, si de aceea vorbim de *generarea* perechii de chei inainte de orice alta operatie.

Cheile generate sunt de obicei salvate in fisiere. Ceea ce se salveaza in fisier este cheia privata, deoarece cea publica se poate deduce pe baza celei private. Putem alege ca fisierul rezultat sa fie criptat; operatia de criptare nu are nicio legatura cu algoritmul de generare a cheilor, ea fiind folosita doar pentru protejarea informatiei din fisier. Criptarea va folosi un algoritm simetric; aceasta va determina solicitarea unei parole la generarea fisierului ce contine cheia privata, aceeasi parola trebuind furnizata la orice citire ulterioara a cheii private.

Nota: *criptarea fisierului ce contine cheia privata poate ridica probleme in cazul serverelor, atunci cand serverul porneste automat la bootare (adica non-interactiv) si nu dispune de un parametru de configurare prin care sa i se comunice parola necesara citirii cheii private.*

Comenzile (si operatiile) openssl difera in functie de algoritmul folosit:

- **chei RSA** (Rivest, Shamir, Adleman) - pot fi folosite pentru:
  - asigurarea integritatii prin semnatura digitala (criptand hash-ul cu cheia privata)
  - asigurarea confidentialitatii (criptand mesajul cu cheia publica a destinatarului)
  - autentificare (criptand mesajul cu cheia privata a expeditorului)
- **chei DSA** (Digital Signature Algorithm) – pot fi folosite doar pentru semnaturi digitale

1.3.4.2. Generarea si vizualizarea cheilor RSA

Comanda openssl folosita este **genrsa**. Sintaxa sa generala este:

```
openssl genrsa -out fisier.key <-des|-des3|-idea> nr_biti_cheie
```

Optiunile -des/-des3/-idea sunt facultative, ele specificand algoritmul folosit pentru criptarea fisierului ce va contine cheia privata (fisier.key in exemplul de mai sus). Lipsa lor determina crearea unui fisier necriptat.

Exemple:

```
# crearea unei chei private de 2048 de biti; fisierul va fi criptat folosind algoritmul 3DES
openssl genrsa -out cheie.key -des3 2048
# dupa generarea cheii se solicita parola!

# generarea unei chei private de 1024 de biti, fara criptare
openssl genrsa -out cheie.key 1024
```

Pentru vizualizarea parametrilor unei chei existente se foloseste comanda openssl **rsa**, urmata in general de numele fisierului in care se afla cheia privata. Daca cheia privata a fost criptata, la citirea sa se va solicita parola. Iata cateva exemple de operatii utile:

```
# afisarea detaliata a parametrilor cheii
openssl rsa -in cheie.key –text -noout

# afisarea cheii publice pe baza celei private
openssl rsa -in cheie.key -pubout

# generarea unui fisier cheie privata criptat pe baza unuia necriptat
openssl rsa -in cheie.key -des3 -out cheie_enc.key
```

1.3.4.3. Generarea si vizualizarea cheilor DSA

Cheile DSA pot fi folosite numai pentru semnatura digitala. In cazul lor este necesara generarea prealabila a unui set de parametri si abia apoi, pe baza lor, a cheilor. Iata comenzile necesare pentru generare si vizualizare:

```
# generare parametri DSA
openssl dsaparam 1024 -out params.dsa

# generare cheie DSA
openssl gendsa –out cheie.dsa params.dsa

# ...sau acelasi lucru dintr-o linie; sunt inclusi in fisier si parametrii DSA
openssl dsaparam –genkey –out cheiedsa.key

# afisarea detaliata a parametrilor cheii
openssl dsa -in cheie.key –text -noout

# afisarea cheii publice
openssl dsa -in cheie.key -pubout
```

1.3.5. Lucrul cu certificate digitale

1.3.5.1. Etape

Generarea unui certificat digital presupune 3 etape:

1. generarea perechii de chei – conform paragrafelor anterioare
2. generarea unui CSR (Certificate Signing Request) ce include identitatea solicitantului si cheia sa publica. Se realizeaza in openssl cu comanda **req**
3. semnarea CSR-ului de catre un CA, rezultand certificatul digital. Se realizeaza in openssl cu comanda **x509** sau **ca**, in functie de caz (vezi mai jos)

1.3.5.2. Generarea si vizualizarea unui CSR

Se realizeaza cu comanda openssl req, care permite atat generarea cat si vizualizarea de CSR:

- in cazul generarii este necesara specificarea optiunii **-new** (astfel incat openssl sa inteleaga ca trebuie sa creeze un CSR, nu sa citeasca unul deja existent). Daca se doreste crearea unui certificat self-signed, se poate adauga optiunea -x509 (vezi mai jos sectiunea despre crearea de certificate)
- in cazul vizualizarii este in general necesara optiunea -in pentru a indica fisierul in care se afla CSR-ul

```
# creare CSR nou, folosind o cheie privata existenta in fisierul cheie.key
openssl req -new -key cheie.key -out cerere.csr

# crearea CSR nou cu comanda combinata, care genereaza pe loc si cheia privata (optiunea –newkey)
openssl req -new -newkey rsa:1024 -keyout cheie.key -out cerere.csr

# vizualizare detalii CSR existent
opensl req –in cerere.csr –text –noout
```

1.3.5.3. Crearea si vizualizarea unui certificat digital

Un certificat digital poate fi:

- self-signed – este cazul certificatelor pentru root CA-uri, care garanteaza pentru ele insele; pot fi folosite de asemenea pentru securizarea comunicatiilor, in masura in care exista o modalitate ca interlocutorul sa declare ca trusted un astfel de certificat
- semnat de catre un CA – cazul majoritatii certificatelor

Iata exemple cu cateva operatii uzuale:

```
# semnare CSR si generare certificat self-signed
openssl x509 –req –in cerere.csr –signkey cheie_semnare.key –out certificat.crt

# comanda combinata pentru generarea unui certificat self-signed (optiunea –x509);
# atentie, comanda openssl folosita este req!
openssl req –new –x509 –key cheie.key –out certificat.crt

# comanda combinata pentru generarea cheilor SI certificatului self-signed;
# optiunea -nodes solicita ca fisierul ce contine cheia privata sa nu fie criptat
openssl req -new -x509 -newkey rsa:1024 -keyout cheie.key -nodes -out certificat.crt

# vizualizarea detaliilor unui certificat digital existent
openssl x509 -in certificat.crt -text -noout

# conversie certificat in CSR (de exemplu pentru prelungirea unui certificat)
openssl x509 –x509toreq –in certificat.crt –out certificat.csr

# semnarea unui CSR folosind o alta cheie privata decat cea corespunzatoare certificatului
openssl x509 -req -in cerere.csr -CA certificatCA.crt -CAkey cheieCA.key -CAcreateserial
                                                -CAserial serial.txt -days 30 -out certificat.crt
```

1.3.6. Verificarea unui certificat digital

Verificarea unui certificat digital presupune mai multe operatii:

- se construieste ierarhia de CA-uri pornind de la certificatul dat. Se verifica prezenta fiecarui certificat in parte
- se verifica fiecare certificat din ierarhie, sub diverse aspecte:
  - validitate – certificatul nu este expirat, semnatura se potriveste cu continutul etc
  - scop – extensiile certificatului permit folosirea sa in scopul semnarii de alte certificate?
  - grad de incredere - este certificatul declarat ca trusted in aplicatia care il valideaza? In general, certificatele declarate explicit ca trusted sunt cele ale root CA-urilor, insa aplicatiile permit in general definirea de exceptii

Pentru a putea valida un certificat, openssl (si orice alt software) are nevoie de urmatoarele:

- lista de certificate ale CA-urilor ce formeaza ierarhia de deasupra certificatului validat. O aplicatie nu poate obtine aceste certificate pe cont propriu, ci are doua variante prin care poate intra in posesia lor:
  - certificatele sunt disponibile local (pre-configurate administrativ, spre exemplu instalate explicit in aplicatie sau in sistemul de operare). Este de obicei cazul certificatelor pentru root CA-uri, care sunt in acelasi timp si declarate ca trusted
  - certificatele sunt primite de la server impreuna cu certificatul serverului. Este in general cazul CA-urilor intermediare
- o lista de certificate trusted. Simpla prezenta a ierarhiei de certificate nu are nicio valoare daca varful ierarhiei nu este un CA in care aplicatia are incredere. De aceea sistemele de operare sau/si aplicatiile dispun de liste de root CA-uri preconfigurate si actualizabile

Pentru a valida certificate, openssl foloseste comanda **verify**, ce primeste urmatoarele optiuni:

- **-CAfile** */cale/catre/fisier* – specifica calea catre fisierul ce contine o lista de certificate de CA-uri trusted
- **-untrusted** */cale/catre/fisier* – specifica calea catre un fisier in care se gasesc, concatenate, toate certificatele CA-urilor intermediare

```
openssl verify -CAfile trustedCA.crt -untrusted listaCAintermediar.crt certificat.crt
```

1.3.7. Fisierul de configurare openssl

O parte dintre comenzile openssl – mai exact x509, req si ca, care sunt mai bogate in optiuni – se pot folosi de un fisier de configurare pentru a obtine o parte dintre parametrii necesari. Fisierul este denumit de obicei openssl.cnf, locatia sa diferind insa de la o distributie Linux la alta, insa numele si locatia sa pot fi usor specificate folosind optiuni openssl.

Fisierul este format dintr-un ansamblu de directive de configurare de forma *parametru=valoare*, ce pot fi grupate in sectiuni. Fiecare sectiune debuteaza cu o linie ce contine numele sectiunii intre paranteze drepte (ex: [ca_default]). Valoarea unei directive poate fi fie un scalar, fie numele unei alte sectiuni. Iata un scurt exemplu:

```
[ca]
default_ca = myca # refera numele altei sectiuni

[myca]
dir = /etc/CA
new_certs_dir = $dir/certificates
```

In acest exemplu avem doua sectiuni ([ca] si [myca]). Parametrul de configurare *default_ca* refera sectiunea [myca]. In aceasta din urma, valoarea parametrului *new_certs_dir* este construita folosindu-se de cea a parametrului *dir*. Constructia $dir este un asa-numit “macro”.

Nota: *o directiva trebuie sa fi fost deja definita pentru a fi folosita ca macro in cadrul valorii alteia. Definitia trebuie sa fi avut loc fie in cadrul aceleiasi sectiuni, fie in contextul global (“radacina” fisierului).*

Fiecareia dintre cele 3 comenzi openssl ce beneficiaza de fisierul de configurare ii poate corespunde in fisier o sectiune cu numele sau ([ca],[req] sau [x509]), directivele cuprinse in acea sectiune aplicandu-i-se automat si furnizand valori default pentru diferitele optiuni care nu sunt specificate in linia de comanda. Aceasta abordare este foarte utila deoarece aceste comenzi sunt deseori invocate in mod repetat cu seturi de parametri identici sau foarte asemanatori: spre exemplu, putem avea nevoie ca la semnarea mai multor certificate sa folosim aceeasi cheie privata de CA, aceeasi perioada de validitate, acelasi algoritm de hashing etc. In loc sa scriem de fiecare data aceste lucruri explicit in linia de comanda, ele vor fi preluate din sectiunea corespunzatoare a fisierului de configurare, singurul eventual element repetitiv fiind optiunea care specifica locatia fisierului de configurare dorit.

Locatia fisierului de configurare poate fi stabilita in urmatoarele moduri:

- daca nu se specifica explicit si nu este modificata la compilare, locatia din oficiu folosita este */usr/local/ssl/lib/openssl.cnf*
- poate fi specificata explicit:
  - cu optiunea -config a comenzii openssl, urmata de calea catre fisierul dorit
  - setand variabila de mediu OPENSSL_CONF. Aceasta ultima varianta este mai comoda in cazul comenzilor openssl multiple care folosesc acelasi fisier de configurare

```
# specificarea fisierului de configurare dorit folosind optiunea -config
openssl ca -in cert.csr -config /etc/openssl.cnf

# alternativ, folosind variabila de mediu
export OPENSL_CONF=/etc/openssl.cnf
openssl ca -in cert.csr
```

1.3.8. Operarea ca CA

Atunci cand openssl este folosit pentru a crea un CA este nevoie de o minima infrastructura, constand din:

- cheia privata a CA-ului. Trebuie protejata la maximum, deoarece compromiterea ei atrage dupa sine compromiterea tuturor certificatelor emise de acel CA
- certificatul digital al CA-ului. In functie de tipul de CA, poate fi:
  - self-signed, pentru un root CA
  - semnat de un alt CA, pentru un intermediate CA
- o baza de date cu certificatele deja emise
- o baza de date cu certificatele revocate
- o evidenta a numerelor seriale de certificate, astfel incat sa nu se emita doua certificate cu acelasi serial number

Toate acestea beneficiaza de directive de configurare in *openssl.conf*, astfel incat nu este nevoie de specificarea lor la fiecare operatie care implica CA-ul.

Iata un exemplu de fisier de configurare openssl:

```
[ ca ]
# daca in comanda openssl ca nu se specifica sectiunea de configurare dorita, se aplica
# cea specificata ca valoare a directivei default_ca
default_ca = theCA

[ theCA ]
dir = /etc/CA                           # directorul de referinta, folosit in celelalte directive
private_key = $dir/cheieCA.pem          # cheia privata a CA-ului
certificate = $dir/certCA.pem           # certificatul CA-ului
database = $dir/index.txt               # lista de numere seriale ale certificatelor emise
new_certs_dir = $dir/certificate        # locul in care sunt plasate certificatele emise
serial = $dir/serial.txt                # fisierul care memoreaza urmatorul serial number

default_days = 1                        # perioada de validitate a certificatului
default_md = sha1                       # algoritmul de hashing folosit

policy = default_policy                 # vezi mai jos explicatia
x509_extensions = default_extensions    # extensii introduse

[ default_policy ]
commonName              = supplied
stateOrProvinceName     = supplied
countryName             = supplied
organizationName        = supplied

[ default_extensions ]
basicConstraints = CA:false
```

Policy-ul reprezinta setul de restrictii introduse asupra elementelor componente ale DN-ului (Distinguished Name) ce reprezinta identitatea solicitantului. Fiecare element dispune de o directiva de configurare corespondenta, valoarea putand fi:

- supplied – campul respectiv trebuie sa fie prezent in DN
- optional – campul poate fi prezent insa nu este obligatoriu
- match – valoarea campului trebuie sa corespunda cu a campului corespunzator din certificatul CA

Atentie! *Orice camp absent in policy dar prezent in CSR va fi sters automat, lipsind astfel din certificatul final!*

Pentru a semna certificate digitale se foloseste comanda **openssl ca** urmata de optiuni precum:

- **-config** *cale* – specifica calea catre fisierul de configurare dorit

- **-name** *sectiune* – specifica numele sectiunii din fisierul de configurare care furnizeaza valorile default pentru CA. Daca nu este specificat numele, se foloseste valoarea directivei default_ca din sectiunea [ca]
- **-in** *fisier.csr* – indica calea catre fisierul ce contine CSR-ul
- **-out** *fisier.crt* – specifica locatia si numele cu care va fi salvat certificatul. Daca aceasta optiune lipseste, certificatul va fi afisat pe ecran (insa o copie a sa va fi plasata oricum in new_certs_dir)

```
openssl ca -in fisier.csr -out fisier.crt -config /etc/openssl.cnf -name theCA
```

Nota: *in scopul simplificarii operatiilor de creare a unui CA si de emitere de certificate, openssl dispune de doua script-uri – CA.pl (script Perl) si CA.sh (script de shell bash). Acestea sacrifica o parte a flexibilitatii insa usureaza mult operatiile amintite.*

1.4. BIBLIOGRAFIE

- How SSL works - http://www.definityhealth.com/marketing/how_ssl_works.html
- Introduction to Public-Key Cryptography - http://docs.sun.com/source/816-6154-10/contents.htm
- Introduction to SSL - http://docs.sun.com/source/816-6156-10/contents.htm
- Cartea *Network Security with OpenSSL*, in editura O'Reilly
- Introducere in protocolul SSL – apache.org: http://httpd.apache.org/docs/2.0/ssl/ssl_intro.html
- Cryptography Tutorial: http://www.cryptostuff.com/crypto/index.php
- Recomandari pentru lungimi de chei: www.keylength.com
- OpenSSL HOWTO: http://www.madboa.com/geek/openssl/
- Cryptography Tutorials: http://www.herongyang.com/Cryptography/index.html
- Configurare extensii de certificate in OpenSSL: http://www.openssl.org/docs/apps/x509v3_config.html
- Formate de fisiere criptografice uzuale: http://www.gtopia.org/blog/2010/02/der-vs-crt-vs-cer-vs-pem-certificates/
- Conversie online intre diferite tipuri de certificate: https://www.sslshopper.com/ssl-converter.html