Provideri Criptografici

4.

- Java Cryptography Architecture (JCA) oferă un framework extensibil pentru providerii criptografici și un set de API-uri pentru accesarea diferitelor primitive criptografice.
    - Aplicațiile care folosesc JCA trebuie doar să ceară un anumit algoritm, fără a fi nevoie să folosească direct un provider specific.
    - Framework-ul se va ocupa să găsească providerul care oferă acel algoritm.
- Cryptographic Service Provider (CSP) este un pachet ce include implementarea unui set de servicii sau algoritmi criptografici.
    - Fiecare provider va anunța către JCA ce servicii și algoritmi implementează.
    - JCA va gestiona un registru cu providerii și algoritmii implementați de aceștia.
    - În acest registru, providerii vor fi ordonați în ordinea preferinței.
    - Dacă doi provideri implementează același algoritm, va fi selectat cel cu preferința cea mai mare (număr de ordine cel mai mic).
- Service Provider Interface (SPI) este o interfață comună care trebuie respectată de toți providerii care implementează un anumit algoritm.
    - Mai exact este o clasă abstractă care este implementată de către providerii ce oferă acel algoritm.

5.

- JCA Engines oferă unul dintre următoarele servicii:
    - Operații criptografice (criptare, decriptare, semnare, verificare semnătura, hash)
    - Generare sau conversie a materialelor criptografice (chei sau parametrii algoritmilor)
    - Gestionare și stocare a obiectelor criptografice (chei, semnături digitale)
- Clasele engine decuplează codul clientului de implementarea algoritmilor, de aceea nu pot fi instanțiate în mod direct.
- În schimb ele oferă o metodă statică de tip factory numită getInstance().
- Prin această metodă se va cere implementarea unui serviciu în mod indirect.
- Avem aici 3 formate ale lui getInstance(). Cel mai utilizat este primul, în care se specifică doar numele algoritmului iar JCA va identifica providerul cu cea mai mare prioritate care oferă acel algoritm.
- În al doilea format, se cere și un anumit provider specificandu-se numele providerului în format string.
- Al treilea da ca argument o instanță a providerului dorit.
- Toate pot arunca NoSuchAlgorithmException dacă nu se găsește algoritmul, iar a doua mai poate arunca NoSuchProviderException, dacă nu găsește providerul respectiv.

6.

- Clasa MessageDigest este folosită pentru a obține o funcție de hash.

- Avem aici un exemplu. Întâi se obţine o instanţă a clasei MessageDigest specificând algoritmul SHA-256 atunci când apelăm metoda factory getInstance(). Apoi pentru datele reprezentate ca un array de bytes aplicăm metoda digest() şi obţinem hash-ul.
- Există două metode prin care îi putem da datele: dacă avem date de dimensiuni mari, putem sa îi dăm bucăţi folosind metoda update(), iar la final apelăm digest() fără nici un argument.
- Dacă avem date mici sau de dimensiune fixă putem apela direct digest() cu datele ca argument.

7.

- Clasa Signature oferă o interfaţă pentru algoritmii de semnături digitale bazaţi pe criptare asimetrică.
- Numele unui algoritm este de obicei <digest>with<encryption>, unde digest este numele unui algoritm de hashing iar encryption este numele un algoritm de criptare asimetric.
- Avem aici un exemplu de creare a unei semnături digitale. Întâi se obţine instanţa cu getInstance si numele algoritmului SHA256withRSA. Apoi se setează cheia privată cu care se va realiza semnătura prin metoda initSign(). Apoi se dau datele prin metoda update(). În final se realizează semnătura prin metoda sign().
- Apoi avem un exemplu de verificare a semnăturii digitale. Întâi se obţine instanţa, apoi se configurează cheia publică cu care se va verifică semnătura prin metoda initVerify(). Apoi se dau datele cu metoda update(). În final se verifică semnătura cu metoda verify().

8.

- Clasa Cipher oferă o interfaţă comună pentru criptare şi decriptare.
- Aici avem un exemplu de criptare. În primul rând obţinem o instanţă Cipher care foloseşte algoritmul de criptare AES, modul de operare CBC şi padding de tipul PKCS#5.
- Apoi generăm un Initialization Vector random şi îl punem într-un obiect de tipul IvParameterSpec.
- Apoi iniţializăm Cipher pentru criptare folosind metoda init() cu flag-ul ENCRYPT_MODE, o cheie de criptare şi un IV.
- Apoi dăm bucăţi de date pentru criptare metodei update() care returnează rezultate intermediare - returnează null dacă datele sunt prea scurte pentru un bloc.
- Se obţine ultimul bloc prin apelarea metodei doFinal().
- Ciphertextul întreg se obţine din concatenarea rezultatelor intermediare cu blocul final.

9.

- Pentru decriptare, obţinem instanţa Cipher şi iniţializăm Cipher-ul folosind init() cu flag-ul DECRYPT_MODE, cheia şi IV-ul primit.
- Îi dăm bucăţi din ciphertext cu metoda update() şi apelam doFinal() pentru a obţine ultima bucată de date.
- Plaintext-ul final este obţinut prin concatenarea rezultatelor intermediare cu ultima bucată de date.

10.

- Clasa Mac oferă o interfață comună pentru algoritmii de timp Message Authentication Code.
- Avem aici un exemplu de utilizare. În primul rând obţinem o instanţă Mac folosind metoda getInstance() în care cerem implementarea algoritmului HMAC care foloseşte SHA-256 ca funcţie de hash.
- Apoi este iniţializată cu metoda init() şi cheia secretă.
- Apoi apelăm metoda doFinal() pentru a obține valoarea MAC-ului pe baza datelor. De asemenea am putea să dăm bucăţi de date prin metoda update() şi în final să apelăm doFinal().

11.

- Clasa KeyGenerator este folosită pentru a genera chei simetrice (folosite pentru algoritmi de criptare simetrici sau algoritmi MAC).
- Este mai bună decât SecureRandom pentru că se realizează verificări adiţionale pentru chei weak şi poate seta bytes de paritate atunci când este necesar (de exemplu pentru DES).
- De asemenea, poate folosi hardware criptografic dacă este disponibil.
- Avem aici două exemple de utilizare. În primul exemplu obţinem o instanţă KeyGenerator pentru algoritmul HmacSha256 şi apoi generăm cheia cu metoda generateKey. Ne va da direct o cheie pe 256 biţi pentru acel algoritm.
- În al doilea exemplu, obținem o instanță KeyGenerator pentru algoritmul AES. Apoi îi dăm dimensiunea cheii - 256, deoarece AES poate suporta 3 dimensiuni de chei. În final generam cheia.

12.

- Clasa KeyPairGenerator este folosită pentru a genera perechi de chei publice şi private (pentru algoritmi asimetrici).
- Avem un exemplu de utilizare. Se obţine instanţa KeyPairGenerator pentru algoritmul RSA. Apoi îi dăm lungimea cheilor. Apoi obţinem un KeyPair folosind metoda generateKeyPair. În final putem obţine cheile cu metodele getPrivate şi getPublic.

13.

- Să vedem câțiva provideri pe care îi putem folosi pe Android.
- Harmony’s Crypto Provider este un provider JCA cu un set restrâns de funcţionalităţi care este inclus în biblioteca de runtime Java. Include doar 4 algoritmi, pentru SecureRandom, KeyFactory, MessageDigest şi Signature.
- Android’s Bouncy Castle Provider este un provider JCA cu un set foarte mare de algoritmi şi servicii, care face parte din Bouncy Castle Crypto API. Avem mulţi algoritmi pentru fiecare clasă în parte: Cipher, KeyGenerator, Mac, MessageDigest, SecretKeyFactory, Signature, CerticateFactory, etc.
- AndroidOpenSSL Provider este implementat în cod nativ din motive de performanţă. Are destul de multe funcţionalităţii, acoperind marea majoritate a algoritmilor şi serviciilor

oferite de Bouncy Castle. Practic implementarea foloseşte JNI pentru a accesa biblioteca OpenSSL. Acest provider este cel preferat în mod implicit, are prioritatea cea mai mare, 1.

SSL/TLS

15.

- Android-ul oferă provideri criptografici pentru calculul hash-urilor, MAC-urilor, pentru criptare, etc. Aceştia pot fi folosiţi pentru asigurarea comunicaţiei securizate.
- Totuşi, pentru a evita introducerea unor vulnerabilităţi, este mai bine să folosim protocoale e securitate standardizate, care a fost specificate şi folosite pentru asigurarea comunicaţiei securizate prin reţea.
- Cele mai folosite protocoale de securitate folosite sunt TLS şi predecesorul său SSL.

16.

- Acestea sunt protocoale pentru comunicaţia securizată punct-la-punct, care oferă autentificare, confidenţialitatea şi integritatea mesajelor trimise între două entităţi care comunică peste TCP/IP.
- Ele folosesc o combinaţie între criptarea simetrică şi asimetrică pentru a asigura confidenţialitate şi integritate şi se bazează pe certificate pentru autentificare.

17.

- De obicei clientul care doreşte să comunice cu un server prin SSL, iniţiază comunicaţia prin trimiterea versiunii de SSL suportate şi a unei liste de cipher suites.
- Un cipher suite este un set de algoritmi folosiţi pentru calculul cheii, autentificare, protecţia integrităţii şi criptare.
- Clientul şi serverul vor negocia un cipher suite care este suportat de amândoi.

18.

- După aceea, se vor autentifica (vor verifica identitatea celuilalt) prin certificate.
- În general, doar serverul se autentifică la client. Totuşi, SSL permite şi autentificarea clienţilor.
- Dacă autentificarea are loc cu succes, ei calculează o cheie simetrică partajată care va fi folosită pentru securizarea comunicaţiei.
- În continuare, comunicaţia va fi securizată folosind algoritmul de criptare şi cheia negociată.

19.

- Un certificat bazat pe cheie publică este folosit pentru asocierea unei identităţi cu acea cheie publică.
- SSL foloseşte certificate X.509 pentru autentificare.

- Aceste certificate includ un număr mare de câmpuri: algoritmul de semnare, entitatea care l-a generat (issuer), validitatea, subiectul, etc.
- Un subiect este reprezentat de un set de atribute incluzând common name (CN), locaţia şi organizaţia, care împreună formează distinguished name (DN).
- Issuer-ul este descris prin niste atribute similare.
- Aici aveţi o parte din certificatul lui Google.

20.

- Dacă clientul SSL comunică cu un număr mic de servere, poate fi configurat cu un set de certificate server de încredere (numite şi trust anchors).
- Aceste certificate pot fi self-signed.
- Un server este de încredere dacă certificatul lui face parte din acest set.
- Avantajul este că putem avea control strict asupra serverelor de încredere.
- Dezavantajul este că e mai greu de actualizat cheia serverului şi certificatul, trebuie modificat/reconfigurat clientul.

21.

- O altă opţiune este folosirea unei autorităţi de certificare (CA) private pentru a semna certificatul serverului. Acest CA privat este folosit ca trust anchor. Clientul va avea încredere în orice certificat care este generat de acest CA.
- Avantajul este ca se poate actualiza uşor cheia serverului şi certificatul, fără actualizarea clientului.
- Dezavantajul este faptul că acest CA devine un sigle point of failure. Dacă CA-ul este compromis, atunci atacatorul poate genera certificate în care clienţii vor avea încredere în mod automat.

22.

- Un client SSL (browser web, client de mail) care nu ştie în avans care vor fi serverele ţintă, este de obicei configurat cu un set de trust anchors, care sunt CA-uri bine cunoscute, publice.
- Majoritatea browserelor web includ un set de mai mult de 100 de CA-uri (certificatele lor) ca trust anchors.

JSSE

24.

- Androidul suportă SSL/TLS prin implementarea Java Secure Sockets Extension (JSSE).
- API-ul JSSE este disponibil prin pachetele javax.net şi javax.net.ssl.
- Oferă:
    - Socketi SSL de tip client şi server
    - Socket factories
    - SSLEngine - care produce şi consumă stream-uri SSL

    - SSLContext - un context de socketi secure, care este folosit pentru a crea factories și engines.
    - Manageri de chei și factories pentru a-i crea
    - Manageri de încredere și factories pentru a-i crea
    - HttpsURLConnection - pentru conexiuni HTTPS

25.

- Această imagine include clasele JSSE și interacțiunea între ele.
- În JSSE, clasele care reprezintă capetele unei conexiuni sunt SSLSocket și SSLEngine
- Figura arată care sunt clasele principale folosite pentru crearea SSLSocket și SSLEngine.

26.

- Un SSLSocket este creat prin SSLSocketFactory, sau prin acceptarea unei conexiuni pe un SSLServerSocket.
- Un SSLServerSocket este creat prin SSLServerSocketFactory.
- Un SSLEngine este creat direct prin SSLContext și se bazează pe faptul că aplicația poate gestiona operațiile I/O.

27.

- Un SSLContext este obținut în două moduri:
    - Prin apelarea metodei getDefault() a SSLServerSocketFactory sau SSLSocketFactory - acesta oferă un context implicit inițializat cu KeyManager-ul implicit, TrustManager-ul implicit și un generator de numere aleatoare. Key material din keystore-ul și truststore-ul implicit sunt obținute din proprietățile de sistem.

28.

- Se poate obține o instață de SSLContext și prin apelarea metodei statice getInstance() a SSLContext. Apoi contextul este inițializat prin următorii parametri: un vector de obiecte KeyManager, un vector de obiecte TrustManager și un SecureRandom. Obiectele KeyManager și TrustManager pot fi obținute prin KeyManagerFactory și TrustManagerFactory. Aceste factories pot fi inițializate cu un KeyStore care conține key material.

29.

- După ce o conexiune a fost creată între un client și un server SSL, se crează un obiect SSLSession.
- SSLSession include informații de genul: identitățile capetelor conexiunii, cipher suite-ul negociat, etc.
- Fiecare conexiune SSL are asociat un SSLSession, iar un SSLSession poate fi folosit pentru mai multe conexiuni între aceleași entități.

30.

- JSSE va delega deciziile legate de încrederea în certificate clasei TrustManager iar selecţia cheilor pentru autentificare clasei KeyManager.
- Fiecare instanţă SSLSocket creată prin JSSE va avea acces la acele clase prin instanţa SSLContext asociată.
- TrustManager are un set de certificate de încredere generate de autorităţi de certificare, şi va lua deciziile pe baza lui. Dacă un certificat este emis de către o autoritate de certificare de încredere atunci acel certificat va fi considerat de încredere.

31.

- TrustManager-ul implicit din JSSE foloseşte trust store-ul de sistem, care include un set de certificate de CA comercile şi guvernamentale.
- Trust store-ul de sistem se găseşte în acest exemplu în /system/etc/security/cacerts.bks
- Aveţi aici un exemplu în care obţinem toate certificatele din trust store-ul de sistem.
- Întai se obţine o instanţă de TrustManagerFactory care se iniţializează (daca ii dam null, va lua automat trust store-ul default, de sistem). Apoi se obţine primul TrustManager, cel default.
- Apoi prin metoda getAcceptedIssuers() se obţine lista certificatelor din trust store-ul implicit. Iar pentru fiecare certificat în parte sunt afişate informaţii.

32.

- Până la Android 4.0, trust store-ul de sistem se afla într-un singur fişier /system/etc/security/cacerts.bks. Totuşi, partiţia de sistem este read-only, deci fişierul nu putea fi modificat (nici de către aplicaţiile de sistem).
- De la Android 4.0, în plus faţă de acest fişier avem două directoare adiţionale: /data/misc/keychain/cacerts-added şi /data/misc/keychain/cacerts-removed.
- Primul include certificate CA are au fost adăugate trust store-ului de sistem, şi a doua include certificate CA care au fost scoase din trust store-ul de sistem.
- Doar utilizatorul system poate adăuga sau scoate certificate CA din trust store-ul de sistem.
- Se pot adăuga trust anchors prin clasa TrustCertificateStore, care este accesibilă prin JCA KeyStore API.

33.

- Acesta este un exemplu de validare manuală a unui certificat de server folosind trust store-ul de sistem.
- În primul rând obţinem o instanţă a lui TrustManagerFactory. O iniţializăm cu trust store-ul de sistem prin folosirea metodei init cu argumentul null.
- Apoi obţinem vectorul de TrustManagers. Îl luam pe primul şi ii facem cast la X509TrustManager.
- Construim un lanţ de certificate incluzând certificatul severului şi alţi issuers intermediari.

- În final, validăm lanţul de certificate folosind metoda checkServerTrusted() a obiectului X509TrustManager.
- SSLSocket şi HttpsURLConnection efectuează această validare în mod automat.

34.

- HttpsURLConnection este metoda recomandată pentru conectarea la un server HTTPS. Foloseşte SSLSocketFactory-ul implicit pentru a crea socketi SSL.
- Atunci când avem nevoie de un trust store custom sau chei de autentificare, SSLSocketFactory-ul implicit poate fi înlocuit prin metoda statică setDefaultSSLSocketFactory() a lui HttpsURLConnection

35-36.

- Dacă vrem să folosim un trust store al nostru, trebuie să parcurgem paşii următori:
- Întai încărcăm trust store-ul nostru (ce conţine trust anchors) într-un obiect KeyStore.
- Apoi vom obţine o instanţă a lui TrustManagerFactory pe care apoi o iniţializăm cu trust store-ul nostru.
- Dacă avem nevoie de autentificarea clientului, încărcăm key material în obiectul KeyStore. Obţinem o instanţă de KeyManagerFactory şi o iniţializăm cu acel key store.
- Apoi obţinem o instanţă SSLContext pentru TLS şi o iniţializăm cu TrustManager şi KeyManager.
- Creăm un obiect URL şi HttpsURLConnection bazat pe acel URL.
- Asociem SSLSocketFactory (a lui SSLContext) la HttpsURLConnection.

37.

- Aici avem un exemplu mai specific, în care încărcăm propriul trust store.
- Putem genera trust store-ul în linia de comandă folosind Bouncy Castle şi openSSL.
- Apoi plasăm trust store-ul în res/raw.
- În primul rând vom obţine o instanţă de KeyStore în format Bouncy Castle, în care vom încărca trust store-ul nostru (va trebui specificată parola).
- Apoi obţinem o instanţă de TrustManagerFactory şi o iniţializăm cu trust store-ul nostru.
- Obţinem o instanţă de SSLContext şi o iniţializăm cu TrustManager-ul obţinut din factory. Observăm că KeyManager este null deoarece nu vrem să realizăm autentificarea clientului în acest exemplu.
- Apoi obţinem obiectul URL, HtttpsURLConnection şi asociem SSLSocketFactory din context la conexiune.

Provideri JSSE

39.

- În mod similar cu providerii criptografici JCA, avem provideri JSSE care implementează funcţionalitatea pentru clase engine definite de API-ul descris anterior.

- Funcţionalitatea implementată de provideri include: secure sockets, trust managers, key managers, etc. Aplicaţiile nu vor lucra în mod direct cu implementarea claselor ci cu clasele engine.
- În Android, avem doi provideri JSSE: Harmony JSSE care implementat în Java, şi AndroidOpenSSL, care este implementat în cod nativ, şi accesat prin JNI.

40.

- HarmonyJSSE este bazat pe socketi Java şi foloseşte clasele criptografice JCA pentru implementarea SSL.
- Oferă suport doar pentru SSLv3 şi TLSv1.
- Este considerat deprecated şi nu este dezvoltat în mod activ.

41.

- AndroidOpenSSL implementează majoritatea funcţionalităţii prin apeluri către biblioteca nativa OpenSSL (prin JNI).
- Oferă suport pentru TLSv1.1 şi TLSv1.2.
- De asemenea oferă suport pentru extensia de TLS numită Server Name Indication (SNI) - ceea ce însemnă că clienţii SSL pot specifica un hostname, în cazul în care serverul are mai multe hostname-uri virtuale.
- SNI este folosit în mod implicit atunci când se face o conexiune cu HttpsURLConnection.
- Amândoi providerii partajează acelaşi cod pentru TrustManager şi KeyManager, dar implementarea de socketi este diferită.