Laboratorul 4. Securitate în Java

Informația schimbată prin rețea este expusă la o serie de riscuri, putând fi interceptată, alterată sau impersonificată. Criptarea informației utilizând chei simetrice sau asimetrice, calcularea digest-urilor mesajelor şi semnarea lor, oferă soluții pentru schimbul privat al informației (codificarea mesajelor astfel încât să devină computational imposibilă descifrarea acestora) şi asigurarea integrității informației (detectarea mesajelor corupte sau modificate). În cadrul acestui capitol sunt prezentate soluțiile de securizare a informației distribuite în cadrul rețelelor de calculatoare folosind tehnologii Java.

1. Suportul arhitectural al securității oferit de Java

Modelul de securitate oferit de Java reprezintă unul dintre punctele sale forte, fiind ideal pentru dezvoltarea aplicațiilor ce rulează în medii distribuite. Modelul de securitate Java permite protejarea utilizatorilor în fața unor situații critice precum rularea locală a programelor descărcate din diverse surse. De exemplu, dacă ne gândim că applet-urile Java rulează local pe maşina utilizatorului în condițiile în care sursa de proveniență a acestora oferă adesea prea puțină încredere, modelul de securitate trebuie să fie capabil să protejeze utilizatorul în fața posibilității descărcării şi rulării accidentale a unor „viruşi".

Modelul de securitate oferit de Java se bazează pe conceptul de „sandbox", o definiție programabilă a limitelor de securitate disponibile unui program. Altfel spus, un program Java poate rula numai în interiorul propriului sandbox. El poate face orice atât timp cât se încadrează în limitele de securitate stabilite. De exemplu, limitele appleturilor Java includ implicit activități precum:

- Citirea sau scrierea pe discul local;
- Stabilirea unei conexiuni pe rețea cu orice stație, cu excepția stației de proveniență a respectivului applet;
- Crearea de noi procese;
- Încărcarea de noi biblioteci dinamice şi apelarea directă de metode native.

Prin limitarea anumitor acțiuni modelul de securitate Java protejează utilizatorul în fața ostilității codului rulat. Astfel, un applet ce se execută local şi

care poate fi descărcat din diverse surse de neîncredere ar putea transporta inclusiv cod de tip virus, însă modelul de securitate face imposibil acest lucru.

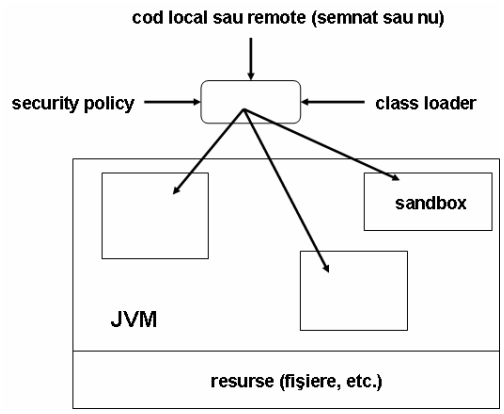

Figura 1. Modelul de securitate oferit de Java.

Componentele arhitecturii modelului de securitate Java ce sunt răspunzătoare de limitarea acțiunilor aplicațiilor sunt:

- Elementele de securitate construite direct la nivel de maşină virtuală (şi de limbaj);
- Arhitectura de încărcare a claselor;
- Verificatorul de încărcare a claselor;
- Managerul de securitate şi API-ul Java.

Două dintre componentele arhitecturale Java deosebit de importante pentru modelul de securitate oferit sunt *class loader* şi *security manager*.

Accesul unei aplicații Java la resursele sistemului cum ar fi ecranul, sistemul de fişiere, procese, fire de execuție, rețea sunt controlate de către managerul de securitate. Limitele de securitate ale unei aplicații Java pot fi controlate prin specificarea de către utilizator a unor noi clase bazate pe managerul implicit *java.lang.SecurityManager*. Definirea unui nou manager de securitate implică suprascrierea de metode declarate în clasa superioară ce au rolul de a decide permiterea efectuării unor operații specifice (precum scrierea pe disc). În cadrul mediului de execuție al aplicației un singur obiect de tip *SecurityManager* poate fi specificat la un moment dat, accesul la toate resursele sistemului pentru respectiva aplicație fiind controlat de metodele respectivului obiect.

Managerul de securitate implicit se poate schimba folosind metoda statică a clasei System, *setSecurityManager()*. Se poate chiar să specificăm că nu dorim să utilizăm deloc un manager prin apelul *System.setSecurityManager(null)* sau chiar

să utilizăm propriul manager de securitate, apelând *System.setSecurityManager(new MySecurityManager())*, unde *MySecurityManager* reprezintă o subclasă a clasei *SecurityManager*.

Un exemplu de clasă ce implementează un manager de securitate propriu care nu permite citirea fişierelor cu extensia *.java* este următorul:

```
class MySecurityManager extends SecurityManager {

   public void checkRead(String s) {

      if (s.endsWith(".java"))
         throw new SecurityException("Access to "+s+" file not allowed");
   }
}

public class MyApplication{

   public static void typeFile(String filename) throws Exception {

      int b;
      FileInputStream fin = new FileInputStream(filename);
      while( (b = fin.read()) != -1 )
         System.out.print( (char) b );
      fin.close();
   }

   public static void main(String args[]) {

      System.setSecurityManager(new MySecurityManager());
      try{
         typeFile(args[0]);
      } catch(Exception e) {
         System.out.println("Eroare citire fisier "+e );
      }
   }
}
```

O altă componentă importantă pentru modelul de securitate Java este *class loaderul*. Mecanismul de încărcare a claselor Java are şi rolul de a verifica dacă clasele încărcate îndeplinesc constrângeri de siguranță la nivelul limbajului. O verificare mai adecvată a securității claselor încărcate poate fi obținută prin înlocuirea class loaderului oferit de sistem. De altfel acest mecanism este folosit de exemplu în cazul încărcării appleturilor. Class loaderul pentru appleturi în Java ştie să încarce clase de pe alte staţii şi ştie să autentifice pachete jar semnate. Acest class loader utilizează spații de nume diferite pentru clasele locale şi pentru clase provenite de pe rețea, pentru a evita conflictul acestora.

Java permite utilizarea unui class-loader definit de utilizator, ce ar putea chiar forța şi mai mult caracteristicile de securitate oferite implicit. Orice class-loader definit de utilizator trebuie să extindă clasa abstractă *java.lang.ClassLoader* şi să redefinească metoda abtractă *loadClass()*. În general se recomandă ca metoda aceasta să implementeze cel puțin următoarele acțiuni:

1. Verifică dacă clasa indicată prin *className* este încărcată. Pentru a putea face acest lucru trebuie să se țină evidența claselor anterior încărcate.
2. Dacă clasa nu este încă încărcată, verifică dacă este clasă sistem. În caz că nu este, va încărca codul octeți (din sistemul local de fișiere de exemplu).
3. Apelezează metoda *defineClass()*pentru a prezenta codul octeți maşinii virtuale.

Exemplul următor ilustrează modalitatea de definire a unui nou class loader de către utilizator:

```
public class MyClassLoader extends ClassLoader {
   private Hashtable classes = new Hashtable();

   protected synchronized Class loadClass( String name, boolean resolve )
         throws ClassNotFoundException {

      Class cl = (Class) classes.get( name );
      if( cl == null ) {
         // Se verifica daca este clasa sistem
         return findSystemClass( name );
      }

      // se incarca codul octeti
      byte classBytes[] = loadClassBytes( name );
      if( classBytes == null ) throw new ClassNotFoundException( name );
      cl = defineClass( name, classBytes, o, classBytes.length );
      if( cl == null ) throw new ClassNotFoundException( name );
      classes.put( name, cl );
      if( resolve) resolveClass( cl );
   }

   private byte[] loadClassBytes( String name ) {

      String cname =name.replace('.','/')+".class";
      FileInputStream in = null;
      try{
         in = new FileInputStream( cname );
         ByteArrayOutputStream buffer = new ByteArrayOutputStream();
         int ch;
         while( ( ch = in.read()) != -1 )
            buffer.write( ch );
         return buffer.toByteArray();
      } catch( IOException e1) {
         if( in != null )
            try{
               in.close();
            } catch( IOException e2 ){}
               return null;
            }
      }
}
```

Clasa descrisă se utilizează precum în următorul exemplu:

```
ClassLoader loader = new MyClassLoader();
Class c = loader.loadClass("OClasa");
```

Java pune la dispoziție şi o serie de mecanisme predefinite de securitate ce operează direct la nivelul maşinii virtuale:

- Type-safe reference casting
- Structured memory access (no pointer arithmetic)
- Automatic garbage collection (can't explicitly free allocated memory)
- Array bounds checking
- Checking references for null

Orice apel către o referință a unui obiect Java impune declanşarea unor mecanisme de verificare implicite. De exemplu, o referință la un tip de date diferit rezultă într-o serie de verificări implicite din partea Java asupra corectitudinii cast-ului. Accesarea unui array implică verificarea implicită dacă indicele cerut se încadrează în limitele respectivului array. Orice astfel de încercare greşită rezultă în aruncarea unor excepții la nivelul maşinii virtuale.

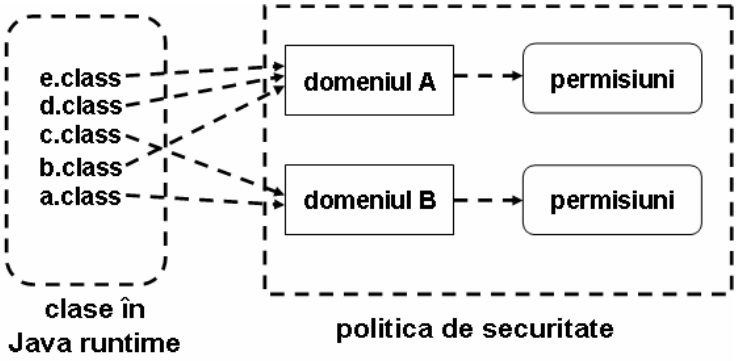

Figura 2. Maparea Clase -> Domenii -> Permisiuni.

Un alt concept al sistemului de securitate este reprezentat de *domeniul de protecție*. Domeniu de protecție este un mecanism pentru gruparea şi izolarea unui set de clase ale căror instanțe au impuse anumite permisiuni. Un domeniu de protecție poate fi limitat de către setul de obiecte ce sunt direct accesate de către un *principal*, o entitate din program căreia i se acordă anumite drepturi. Astfel este posibilă chiar separarea domeniilor de protecție, a interacțiunilor între ele, astfel încât orice interacțiune cu anumite rutine aparținând de diverse domenii de protecție să poată fi facută numai prin intermediul unor rutine sigure ale sistemului.

Domeniile de protecție se împart în două categorii: de sistem şi de aplicație. Orice resursă externă protejată, precum sistemul de fişiere, interfețele de rețea, tastatura sau monitorul, poate fi accesată numai prin domenii de protecție ale sistemului.

Domeniile de protecție sunt determinate de către politica de securitate folosită. Java menține întotdeauna intern o mapare între cod (clase şi instanțe) şi domeniile de protecție şi permisiunile curent acordate unor entități ale aplicației executate.

Un fir de execuție poate executa acțiuni doar în interiorul propriului domeniu de protecție sau în interiorul domeniului de protecție al sistemului (de exemplu o aplicație poate afişa un mesaj, acțiune ce necesită accesarea domeniului de protecție al sistemului). Ca regulă, permisiunile sunt acordate astfel:

- Setul de permisiuni acordate unui fir de execuție sunt considerate a fi intersecția tuturor permisiunilor corespunzătoare tuturor domeniilor de protecție traversate de către execuția sa.
- Atunci când este apelată metoda *doPrivileged* a clasei *AccessController* setul de permisiuni acordate firului de execuție curent va include şi o permisiune acordată de domeniul de protecție al codului.

Metoda *doPrivileged* acordă unei aplicații drepturi temporare la mai multe resurse decât sunt disponibile implicit acesteia. Acest aspect este necesar în câteva situații excepționale. De exemplu, o aplicație poate să nu aibă acces direct la fişierele sistemului conținând setul de fonturi folosite, dar ea trebuie să permită utilizatorului să aleagă dintre fonturile pe care acesta le poate folosi în scrierea unui document. În acest moment poate fi folosită metoda *doPrivileged* a domeniului de sistem. În timpul execuției unei aplicații Java, dacă este apelată o operație critică asupra unei resurse a sistemului (operație asupra unui fişier de exemplu), transparent pentru utilizator este apelată în fundal şi o metodă a clasei *AccessControler* ce evaluează dreptul de execuție al respectivei operații. În afara acestui comportament implicit, orice aplicație poate defini mecanisme suplimentare de protecție a resurselor interne acesteia. De exemplu, o aplicație bancară poate implementa mecanisme interne de protecție a unor resurse precum conturi bancare, depozite sau retrageri. Domeniul de protecție pentru astfel de aplicații este specificat de către dezvoltatorii acestora, prin intermediul unor resurse ajutătoare puse la dispoziție de Java (într-o secțiune ulterioară vom prezenta clasa *SignedObject* de exemplu).

În Java accesul la resurse ale sistemului este specificat prin intermediul unor clase speciale ce implementează *java.security.Permission*. Un exemplu de folosire a clasei *java.io.FilePermission* în scopul acordării dreptului de a citi fişierul de pe disc */tmp/abc* este următorul:

```
perm = new java.io.FilePermission("/tmp/abc", "read");
```

Dezvoltatorii de aplicații pot specifica noi permisiuni prin extinderea clasei abstracte de bază *java.security.Permission* sau extensii ale acesteia, precum *java.security.BasicPermission*. O metodă importantă ce trebuie implementată de către noile extensii de clase este *implies*. Această metodă este apelată pentru verificări de tipul „a implică b", adică dacă *a* are o anumită permisiune acordată atunci şi *b* are acordat dreptul respectiv.

Pentru crearea unei noi permisiuni este recomandată parcugerea câtorva paşi. Să presupunem că dorim crearea unei noi permisiuni în cadrul dezvoltării unei aplicații Java pentru „vizualizare TV". Primul pas va consta în crearea unei noi clase *com.abc.TVPermission*, ce extinde clasa abstractă *java.security.Permission* (sau o subclasă a acesteia).

```
public class TVPermission extends java.security.Permission
```

Clasa nou creată trebuie inclusă în pachetul furnizat împreună cu aplicația dezvoltată. Ulterior, fiecare utilizator ce doreşte să adauge această permisiune pentru anumite operații va trebui să adauge o intrare corespunzătoare în fişierul de politici. Un exemplu de astfel de intrare ce acordă dreptul codului provenind de la *http://java.sun.com* de a vizualiza canalul 5 este următorul:

```
grant codeBase  "http://java.sun.com/" {
   permission com.abc.TVPermission "channel-5", "watch";
}
```

În cadrul aplicației, atunci când dorim să verificăm valabilitatea unei aplicații de a executa un anumit cod folosim metoda *checkPermission* a clasei *AccessController* furnizând ca parametru obiectul de tip *com.abc.TVPermission*, precum în exemplul următor:

```
com.abc.TVPermission tvperm = new
       com.abc.TVPermission("channel-5", "watch");
AccessController.checkPermission(tvperm);
```

Politica de securitate a sistemului în cadrul unei aplicații Java, cuprinzând permisiunile disponibile codului provenind din diverse surse externe, este reprezentată de către o instanță a clasei *java.security.Policy*. Mai exact, o subclasă a acestei clase abstracte. În cadrul unei aplicații mai multe astfel de obiecte pot fi definite, dar o singură instanță poate fi în funcție în orice moment al ciclului de viață al aplicației. De altfel putem afla care este obiectul Policy curent activ prin apelarea metodei *getPolicy* şi putem oricând să modificăm politica prin apelarea metodei *setPolicy*.

Locația informației folosită de obiectul Policy este lăsată la latitudinea aplicației Java (pot fi folosite fişiere de configurație a politicii de securitate de tip text, fişiere binare serializate sau chiar baze de date). Implementarea de referință a politicii (cea folosită implicit dacă nu specificăm nimic la rularea unei aplicații

Java) foloseşte informaţia definită în fişiere statice de configurare. Un astfel de fişier de configurare a permisiunilor acordate entităţilor aplicaţiei este codat în format UTF-8.

Un fişier de configurare conţine în esenţă o listă de intrări. El poate conţine o intrare de tip „keystore" şi zero sau mai multe intrări de tip „grant". Un keystore reprezintă o bază de date de chei private şi de certificate digitale asociate acestora, precum lanţuri de certificate X.509 ce sunt folosite pentru autentificarea unor chei publice corespondente (detalii despre certificate sunt prezentate în subcapitolul următor). Keystore-ul specificat în fişierul de configurare este folosit pentru a verifica cheile publice ale semnatarilor specificaţi în intrările „grant". O intrare keystore trebuie specificată numai atunci când în intrările „grant" sunt specificaţi semnatari. Sintaxa unei astfel de intrări este următoarea:

```
keystore "some_keystore_url", "keystore_type";
```

În cadrul exemplului, *some_keystore_url* desemnează locaţia URL a fişierului keystore, iar *keystore_type* specifică tipul de keystore folosit (ultimul argument fiind opţional – implicit JKS).

Fiecare intrare de tip „grant" are următorul format:

```
grant [SignedBy "signer_names"] [, CodeBase "URL"]
      [, Principal [principal_class_name] "principal_name"]
      [, Principal [principal_class_name] "principal_name"] ... {
    permission permission_class_name [ "target_name" ]
               [, "action"] [, SignedBy "signer_names"];
    permission ...
};
```

De exemplu, următoarea politică acordă permisiunea *a.b.Foo* codului semnat de *George*:

```
grant signedBy "George" {
    permission a.b.Foo;
};
```

Următorul exemplu specifică acordarea permisiunii de citire fişierelor *.tmp* întregului cod (indiferent de semnatarul aplicaţiei sau de sursa de proveniență a acesteia):

```
grant {
   permission java.io.FilePermission ".tmp", "read";
};
```

Următorul exemplu specifică acordarea a două permisiuni codului ce este semnat de *Michel* şi care provine de la *http://java.sun.com*:

```
grant codeBase "http://java.sun.com/*", signedBy "Michel" {
```

```
    permission java.io.FilePermission "/tmp/*", "read";
    permission java.io.SocketPermission "*", "connect";
};
```

Java mai oferă câteva mecanisme suplimentare de protecţie internă. Să presupunem de exemplu un scenariu în care un furnizor de obiecte rulează într-un alt fir de execuţie decât consumatorul respectivelor obiecte. Pentru astfel de scenarii poate fi utilă folosirea unor obiecte de tip *GuardedObject* ce asigură protecţia accesului la resurse.

Furnizorul de resurse crează obiectul ce trebuie transmis şi crează şi un obiect *GuardedObject* ce încapsulează obiectul iniţial împreună cu un obiect *Guard* şi care este de fapt trimis consumatorului. Consumatorul nu poate obţine ulterior obiectul inclus în ceea ce primeşte decât dacă anumite verificări de securitate sunt satisfăcute. *Guard* reprezintă o interfaţă şi practic orice obiect o poate implementa. Singura metodă a acestei interfeţe este *checkGuard*. Metoda primeşte un obiect şi execută anumite verificări de securitate. Clasa *java.security.Permission* implementează această interfaţă.

De exemplu, să presupunem că dorim să putem cere unui fir de execuţie să deschidă un fişier *a/b/c.txt* pentru citire, dar nu putem avea încredere în cine execută respectiva cerere sau în ce împrejurări (de exemplu, în cazul aplicaţiilor distribuite, putem pune la dispoziţie o serie de operaţii unor utilizatori...). Într-un astfel de context putem folosi un obiect *GuardedObject* pentru a forţa verificări de control al accesului, după cum urmează:

```
FileInputStream f = new FileInputStream("/a/b/c.txt");
FilePermission p = new FilePermission("/a/b/c.txt", "read");
GuardedObject g = new GuardedObject(f, p);
```

În acest moment firul de execuţie poate transmite *g* către firul de execuţie consumator, pentru obţinerea accesului la *f* acesta fiind obligat să apeleze:

```
FileInputStream fis = (FileInputStream) g.getObject();
```

Metoda aceasta invocă metoda *checkGuard* a obiectului *p*, şi, doarece *p* este de tip *Permission*, apelul respectiv ajunge să execute:

```
SecurityManager sm = System.getSecurityManager();
if (sm != null) sm.checkPermission(this);
```

Aceasta asigură efectuarea unui control al permisiunilor corespunzător. În acest fel pot fi implementate diverse politici de control ce pot verifica informaţii de context, precum dreptul execuţiei unor operaţii în funcţie de ora curentă sau identitatea apelantului.

2. Suportul pentru securitatea comunicațiilor în Java

2.1. Certificate

Funcția principală a unui certificat este aceea de a asocia unei identități o cheie publică. Certificatele sunt publice şi conțin informație referitoare la subiectul certificatului (Entitatea care deține certificatul - numele, cu ce organizație e asociat acest nume, locația şi țara), cheia publică a certificatului, entitatea care a emis acest certificat şi semnatura acesteia. Asocierea cheie-identitate (certificatul) este recunoscută şi garantată de un terț, entitatea care semnează certificatul şi care este numită **Autoritate de Certificare**.

Un certificat poate fi obținut printr-o cerere făcută unei Autorități de Certificare. Cel care dorește să dețină un astfel de certificat va genera local o pereche de chei, publică şi privată şi va arăta cheia publică Autorității de Certificare care îi va creea şi semna un certificat conținând acea cheie publică. În principiu se crează local o cerere de certificat, de fapt un certificat nesemnat, care conține identitatea entității şi cheia publică, certificat care este trimis Autorității de Certificare spre semnare. Autoritatea de Certificare este responsabilă să verifice dacă identitatea din certificatul primit aparține celui care a trimis spre semnare certificatul. De obicei acest proces se face offline, find implicate şi chestiuni legislative. Autoritatea de Certificare asigură integritatea datelor din certificat prin calcularea unui hash peste întregul certificat şi semnarea hash-ului cu cheia privată a Autorității de Certificare (care deține şi ea un certificat şi o cheie privată asociată).

După cum se observă Autoritatea de Certificare trebuie să fie considerată de încredere pentru ca asocierea cheie publica - identitate prezentă în certificat să fie viabilă. Este uşor de imaginat că la nivel global nu se poate stabili o singură Autoritate de Certificare, atât din motive de încredere cât şi de scalabilitate. Astfel există un numar de Autorități de Certificare considerate rădăcină. Aceste autorități sunt cunoscute şi considerate de încredere, iar certificatele lor (deci şi cheia publică) sunt de obicei încorporate în aplicațiile care fac uz de autentificare prin certificate. Un certificat semnat de o astfel de Autentificare de Certificare este considerat (hardcodat) de încredere, însă de cele mai multe ori Autoritățile de Certificare rădăcină nu emit certificate direct utilizatorilor, ci altor Autorități de Certificare care pot fi responsabile cu anumite regiuni şi care, la rândul lor, ar putea emite certificate pentru alte Autorități de Certificare, construindu-se astfel câteva nivele până la Autoritățile care emit certificate utilizatorilor (a se vedea Figura 5.3).

Un certificat al unui client este verificat pe lanțul de semnături până ce se ajunge la o Autoritate de Certificare de încredere sau se stabileşte că un astfel de lanț nu există, caz în care autentificarea eşuază.

Întregul sistem cu entități care dețin/cer certificate şi Autorități de Certificare poartă numele de Infrastructură cu Chei Publice (Public Key Infrastructure - PKI).

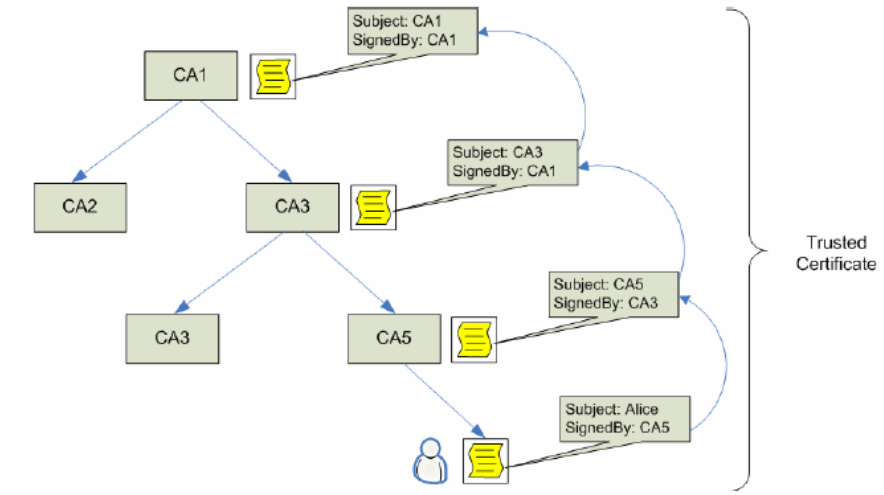

Figura 3. Autorități de Certificare.

Autentificarea poate fi cerută de ambele entități care doresc să comunice sau poate fi pretinsă numai de una dintre acestea. Principiul este următorul: entitatea care trebuie să se autentifice (A) prezintă certificatul deținut (de cele mai multe ori va prezenta un lanț de certificate până la o Autoritate de Certificare posibil cunoscută de cealaltă parte), cealaltă parte (B) verifică dacă certificatul este de încredere şi, dacă acesta este cazul, generează aleator un set de date şi cere ca acestea să fie semnate cu cheia privată pereche a cheii publice din certificatul prezentat. După ce primeşte datele, B le decriptează cu cheia publică (dezvaluită în certificatul arătat anterior) şi dacă obține aceeaşi valoare entitatea A (care a prezentat certificatul) este considerată deținătoarea de drept a acestuia, deci având identitatea din certificat.

Certificatele în formatul X.509 sunt un standard ITU-T pentru Infrastructura cu Chei Publice. Un certificat X.509 conține cel puțin următoarele date:

- **Versiunea**. Există trei versiuni de certificate X.509 v1, v2 şi v3.
- **Număr Serial**. Identifică certificatul şi trebuie să fie unic între certificatele semnate de Autoritatea de Certificare care emite şi certificatul curent (numărul serial şi numele emitentului identifică unic un certificat).
- **Algoritm de Semnare**.
- **Emitent**. Numele emitentului în format X.500 (în general o Autoritate de Certificare).

- **Perioada de Valabilitate**. Data când certificatul devine valabil şi data când acesta expiră.
- **Subiectul**. Numele celui care deține certificatul (poate fi în format X.500 sau într-un alt format - ex. URL).
- **Cheia Publica**.
- **Semnatura Emitentului**.

Numele în format X.500 conțin câteva câmpuri care oferă informații suplimentare despre entitatea respectivă. În format X.500 pot apărea următoarele atribute:

- Common Name CN – Numele deținătorului.
- Organization O – Organizația cu care este asociat numele.
- Organization Unit OU – Unitatea din cadrul organizației.
- Country C – Țara.

Certificatele X.509 v1 au apărut în 1988 şi au utilizat formatul X.500 pentru reprezentarea emitentului şi subiectului unui certificat. X.509 v2 a introdus conceptul de identificatori unici pentru subiect şi emitent pentru a se putea reutiliza aceste nume, însă această propunere a fost controversată şi cele mai multe recomandări referitoare la certificate sunt împotriva refolosirii numelor. Certificatele X.509 v2 au continuat să utilizeze formatul X.500 de reprezentare a numelor. Certificatele X.509 v3 sunt cele mai recente, făcându-şi apariția în anul 1996. Noutatea adusă este prezența extensiilor, care pot fi definite şi incluse în certificate (semnatura certificatului este calculată şi peste aceste extensii).

Extensiile permit ca un certificat v3 să cuprindă şi alte informații pe lângă identitatea deținătorului (ex. atribute ale deținătorului). O serie de extensii sunt deja definite şi acceptate ca atare (Utilizare Cheii – KeyUsage – specifică scopul în care poate fi utilizată cheia din certificat, Nume Alternative – AlternativeNames – pentru a asocia şi alte nume cu cheia publică din certificat etc.) altele pot fi definite de utilizatori. Aceste extensii pot fi marcate ca fiind critice sau necritice. O extensie critică ar trebui verificată şi utilizată/interpretată de aplicația care primeşte certificatul respectiv, însă aceasta depinde de convențiile şi modul de funcționare al aplicației. În plus certificatele în format v3 nu obligă reprezentarea în format X.500 a subiectului sau a emitentului putându-se utiliza nume generice.

În Java mecanismele de securitate sunt legate de noțiunea de *keystore*, fişiere conținând chei şi certificate. Un keystore poate conține două tipuri de intrări: certificate de încredere şi mapări cheie/certificat, fiecare astfel de intrare conținând o cheie privată şi certificatul corespunzător semnat cu cheia publică. Fiecare intrare din keystore este identificată printr-un un *alias*.

Certificatele care sunt folosite în cadrul setării unor sesiuni de comunicație sigure sunt citite din keystore-ul furnizat la rularea aplicației, iar la recepție

certificatele sunt verificate contra certificatelor conţinute în fişiere trustore. Instrumentul pentru crearea de keystore-uri şi trustore-uri este *keytool*. Paşii pentru crearea unui keystore şi a unui trustore sunt următorii: se porneşte cu crearea unui keystore, din acest keystore se exportă un certificat ce este importat într-un trustore, keystore-ul rămâne folosit local iar trustore-ul este furnizat aplicaţiei remote.

Crearea unei noi perechi de chei (publică şi privată) şi a unui certificat pentru un subiect *Duke*, folosind algoritmul DSA se poate face astfel:

```
> keytool -genkey -alias alias -keystore .keystore
    Enter keystore password: password
    What is your first and last name?
      [Unknown]: Duke
    What is the name of your organizational unit?
      [Unknown]: JavaSoft
    What is the name of your organization?
      [Unknown]: Sun
    What is the name of your City or Locality?
      [Unknown]: Cupertino
    What is the name of your State or Province?
      [Unknown]: CA
    What is the two-letter country code for this unit?
      [Unknown]: US
    Is <CN=Duke, OU=JavaSoft, O=Sun, L=Cupertino, ST=CA, C=US> correct?
      [no]: yes
```

Dacă se doreşte folosirea algoritmului RSA pentru creare se poate folosi:

```
> keytool -genkey -keyalg RSA -keysize 1024 -alias alias -keystore .keystore
```

Exportarea unui certificat din keystore se poate face astfel:

```
> keytool -storepass my-keystore-password -alias myalias -export -file
outfilename.cer
```

Importarea unui certificat dintr-un fişier se poate realiza folosind:

```
> keytool -storepass my-keystore-password -alias myalias -import -file
infilename.cer
```

Paşii de creare a unei perechi keystore-trustore de certificate semnate de un CA sunt astfel următorii:

```
Setarea CA-ului
-----------------------------------

Pasul 1. Se instalează openssl local (sursele sau binarele sunt disponibile
la www.openssl.org).

Pasul 3. Se generează o cheie privată (ca.key) şi o cerere de certificat
(ca.csr) pentru CA:

openssl req -new -newkey rsa:1024 -nodes -out ca.csr -keyout ca.key
```

Pasul 4. Se crează certificatul semnat de propriul CA ca.pem (în exemplu perioada de validitate a certificatului va fi de un an):

```
openssl x509 -trustout -signkey ca.key -days 365 -req –in ca.csr -out ca.pem
```

Pasul 5. Se importă certificatul CA-ului în keystore-ul conţinând certificate autorizate JDK:

```
keytool -import -keystore $JAVA_JOME/jre/lib/security/cacerts -file
ca.pem -alias my_ca
```

Pasul 6. Se crează un fişier ce va conţine numerele seriale ale CA-ului ca.srl, ca în exemplul:

```
echo "02" > ca.srl
```

Creare keystore şi trustore

Pasul 7. Se crează un keystore keystore.ks.

```
keytool -genkey -alias student -keyalg RSA -keysize 1024 -keystore
keystore.ks –storetype JKS
```

Pasul 8. Se crează o cerere de certificat keystore.csr ce este trimisă CA-ului.

```
keytool -certreq -keyalg RSA -alias student –file keystore.csr -keystore
keystore.ks
```

Fişierul keystore.csr trebuie modificat dintr-ul editor text. Se înlocuieşte textul „NEW CERTIFICATE REQUEST” cu „CERTIFICATE REQUEST”

Pasul 9. Semnarea cererii de către CA cu generarea fişierului keystore.crt:

```
openssl x509 -CA ca.pem -CAkey ca.key –Caserial ca.srl -req -in keystore.csr
–out keystore.crt -days 365
```

Pasul 10. Importarea certificatului semnat în keystore:

```
keytool -import -alias student –keystore keystore.ks -trustcacerts -file
keystore.crt
```

Execuţia comenzii va genera un mesaj „Certificate reply was installed in keystore”.

Pasul 11. Importarea certificatului CA în keystore:

```
keytool -import -alias my_ca –keystore keystore.ks -trustcacerts -file
ca.pem
```

Acest pas este necesar numai dacă se doreşte folosirea autentificării din partea clientului.

Pasul 12. Exportarea certificatului din keystore (opţiunea –rfc generează un certificat printabil) în keystore.cer:

```
keytool -export -rfc -alias student -file keystore.cer -keystore keystore.ks
-storepass JKS
```

```
Pasul 13. Importarea certificatului într-un trustore:

keytool -import -alias student -file keystore.cer -keystore trustore.ks
```

2.2. SSL

De multe ori se pune problema realizării unor comunicații sigure, confidențiale, astfel încât mesajele schimbate să nu poată fi descifrate. SSL (Secure Socket Layer) este un protocol de securitate care oferă aceste calități. Prin intermediul său se poate asigura confidențialitate, integritatea mesajelor şi autentificarea părților. SSL acționează peste un flux TCP şi oferă servicii nivelelor superioare. Protocolul HTTP poate fi utilizat peste SSL şi în acest caz este numit HTTPS (Secure Hyper Text Transfer Protocol). HTTPS funcționează pe acelaşi principiu cu HTTP, diferența constând în criptarea mesajelor schimbate între un server web şi un browser şi în posibilitatea autentificării atât a serverului cât şi a clientului. SSL este însă un protocol general care poate oferi servicii oricărui protocol de nivel aplicație.

Protocolul SSL este format din două etape: *handshake* şi *transfer*. În timpul handshake-ului clientul şi serverul stabilesc un set de algoritmi pentru realizarea criptării, stabilesc cheile care vor fi utilizate şi se autentifică. Doar autentificarea serverului este obligatorie, cea a clientului fiind opțională. După încheierea handshake-ului, în cea de-a doua fază, transferul, datele sunt sparte în blocuri de dimensiuni mai mici (opțional pot fi şi compresate) protejate pentru garantarea integrității şi criptate după care are loc transmisia propriu-zisă pe rețea.

EXEMPLUL 1. Pentru exemplificare, prezentăm modul de construcție a unei aplicații client-server ce folosesc SSL pentru schimbul de informații.

Un server SSL necesită specificarea unor certificate ce sunt prezentate clienților în procesul de autentificare. În Java certificatele sunt citite din fişiere de tip keystore ale căror locații trebuie să fie explicit specificate (nu există locație implicită).

Crearea unui server SSL este ilustrată în exemplul următor. După cum se poate observa, Java uşurează sarcina dezvoltatorului prin furnizarea clasei *SSLServerSocketFactory*. Crearea unui *ServerSocket* pregătit pentru comunicația folosind SSL se realizează apelând metoda *createServerSocket* a acestei clase ajutătoare. Modul de folosire a unui astfel de server este cel obişnuit, prin folosirea metodelor *accept*, *getInputStream* sau *getOutputStream*. Practic, în afară de modul diferit de construcție a instanței, nimic nu trădează că vorbim de o comunicație mai specială.

```
try {
   int port = 443;
   ServerSocketFactory ssocketFactory = SSLServerSocketFactory.getDefault();
   ServerSocket ssocket = ssocketFactory.createServerSocket(port);

   // acceptam noi conexiuni
   Socket socket = ssocket.accept();

   // crearea stream-urilor de date
   InputStream in = socket.getInputStream();
   OutputStream out = socket.getOutputStream();

   // diverse operații de transfer...

   // inchiderea conexiunii
   in.close();
   out.close();
} catch(IOException e) { }
```

La rularea aplicației server trebuie specificat şi numele fişierului *keystore* folosit. Specificarea fişierului keystore conținând certificatele de folosit se poate specifica folosind proprietatea *javax.net.ssl.keyStore*, rularea aplicației server fiind efectuată astfel:

```
> java -Djavax.net.ssl.keyStore=mySrvKeystore –
Djavax.net.ssl.keyStorePassword=123456 MyServer
```

Codul corespunzător aplicației client ce foloseşte SSL este prezentat în următorul exemplu. Din nou, folosim o clasă ajutătoare, *SSLSocketFactory*, ce ascunde detaliile de implementare ale unui socket ce „vorbeşte" protocolul SSL. De această dată însă vedem o serie de metode noi. După stabilirea conexiunii clientul trebuie să apeleze metoda *startHandshake* pentru a iniția primul pas în stabilirea conectivității. Fără apelarea aceastei metode o operație de transfer ar arunca exceptie. Operația se termină odată cu finalizarea handshake-ului (dacă autentificarea nu reuşeşte se aruncă excepție). Ulterior se poate continua cu transferul datelor în formă sigură, precum este prezentat în exemplu.

```
try {
   int port = 443;
   String hostname = "hostname";
   SocketFactory socketFactory = SSLSocketFactory.getDefault();
   Socket socket = socketFactory.createSocket(hostname, port);

   // conectarea la server – declansarea handshakeului
   socket.startHandshake();

   // regasirea lantului de certificate prezentate de server
   java.security.cert.Certificate[] serverCerts =
        socket.getSession().getPeerCertificates();

   // crearea streamurilor de comunicatie
   InputStream in = socket.getInputStream();
   OutputStream out = socket.getOutputStream();
```

```
   // diverse operatii de citire/scriere...

   // inchiderea socketilor
   in.close();
   out.close();
} catch(IOException e) { }
```

În cadrul exemplui se poate observa şi modalitatea prin care clientul poate obține informații privind identitatea prezentată în certificatul folosit de server în procesul de autentificare. Atunci când un client SSL se conectează la un server SSL primeşte un certificat de autentificare din partea serverului. Clientul trebuie să valideze certificatul prin compararea acestui cu un set de certificate conținute în propriul trustore. Fişierul trustore implicit este *<java-home>/lib/security/cacerts*. Dacă certificatul prezentat de server nu poate fi validat contra certificatelor din trustore, certificatul serverului trebuie să fie adăugat în trustore înainte ca conexiunea să poată fi stabilită. Un alt fişier trustore poate fi specificat folosit proprietatea *javax.net.ssl.trustStore* , rularea aplicației anterior prezentate făcându-se astfel:

```
> java -Djavax.net.ssl.trustStore=truststore -
Djavax.net.ssl.trustStorePassword=123456 MyApp
```

În cadrul listingului *example1* este prezentat un exemplu mai amplu de aplicație client/server ce foloseşte de această dată TLS ca protocol de autentificare. În cadrul exemplului se demonstrează şi modalitatea de construcție a unui manager de certificate propriu utilizatorului, manager ce este folosit în pasul de stabilire a identității celor două entități participante (server şi client). Codul corespunzător serverului responsabil de setarea corespunzătoare a canalelor de comunicație este următorul:

```
public SSLServerSocket createServerSocket(int port, String keystore, String
         password) throws IOException {

   SSLServerSocketFactory ssf = null;
   SSLServerSocket ss = null;
   try {
      SSLContext ctx = SSLContext.getInstance("TLS");
      KeyManagerFactory kmf =
        KeyManagerFactory.getInstance( KeyManagerFactory.
        getDefaultAlgorithm());
      KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
      FileInputStream is = new FileInputStream(keystore);
      ks.load(is, password.toCharArray());
      kmf.init(ks, password.toCharArray());
      ctx.init(kmf.getKeyManagers(), null, new
         java.security.SecureRandom());
      ssf = ctx.getServerSocketFactory();
      ss = (SSLServerSocket) ssf.createServerSocket();
      ss.bind(new InetSocketAddress(port));
      ss.setNeedClientAuth(false);
   } catch (Throwable t) {
```

```
      t.printStackTrace();
      throw new IOException(t.getMessage());
   }
   return ss;
}
```

Exemplul demonstrează o modalitate de încărcare dinamică a certificatelor ce sunt folosite în etapa de stabilire a identităților participanților la comunicație. Acest lucru se realizează folosind clasele *KeyManagerFactory* şi *KeyStore*. Crearea contextului de securitate se realizează în cadrul exemplului folosind clasa *SSLContext* (ce poate fi instanțiată, precum în cadrul exemplului, cu specificarea folosirii protocolului TLS). După cum s-a prezentat şi în exemplul anterior, un *SSLServerSocket* se poate obține plecând de la o instanță *SSLServerSocketFactory*, care poate fi cea implicită (*SSLServerSocketFactory.getDefault*) sau poate fi obținută dintr-un context (*ssf=ctx.getServerSocketFactory()*). Metoda *setNeedClientAuth* este importantă, ea specificând dacă în etapa de handshake este necesar ca şi clientul să prezinte sau nu propriul certificat.

Codul corespunzător clientului este următorul:

```
public void createSSLConnection (String address, int port) throws Exception{

   String store=System.getProperty("KeyStore");
   String passwd =System.getProperty("KeyStorePass");
   char[] storepswd = passwd.toCharArray();
   SSLContext ctx = SSLContext.getInstance("TLS");
   KeyManagerFactory kmf =
        KeyManagerFactory.getInstance( KeyManagerFactory.
        getDefaultAlgorithm());
   KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
   ks.load(new FileInputStream(store), storepswd);
   kmf.init(ks,storepswd);
   ctx.init(kmf.getKeyManagers(), new TrustManager[] {new
      CustomTrustManager()}, null);
   SSLSocketFactory ssf = ctx.getSocketFactory();
   s = (SSLSocket)ssf.createSocket();

   s.connect(new InetSocketAddress(address, port));

   pw = new PrintWriter(new OutputStreamWriter(s.getOutputStream()));
   br = new BufferedReader(new InputStreamReader(s.getInputStream()));
}
```

Setarea contextului este similară celei prezentate în cazul serverului. O construcție diferită este prezentată de inițializarea contextului (metoda *init* a clasei *SSLContext*). În cadrul exemplului este folosită o clasă *CustomTrustManager* ce implementează interfața *X509TrustManager*.

3. Criptare în Java

Pachetul Java Cryptography Extension (JCE) este inclus în JDK începând de la versiunea 1.4. Supportul JCE pentru criptare include:

- Symmetric bulk encryption (DES, RC2, and IDEA)
- Symmetric stream encryption (RC4)
- Asymmetric encryption (RSA)
- Password-based encryption (PBE)
- Key Agreement
- Message Authentication Codes (MAC)

Bouncy Castle (BC) este un toolkit care oferă o implementare lightweight pentru JCE 1.2.1 şi este compatibil cu Java Micro Edition (J2ME), astfel încât reprezintă soluția cea mai bună pentru un API cryptografic pentru dispozitive mobile care rulează Java.

EXEMPLUL 2. Prezentăm un exemplu de aplicație pentru criptarea/decriptarea unui şir de date folosind algoritmul DES (chei simetrice).

Pentru exemplificarea modului de folosire a mecanismelor de criptografie puse la dispoziție de Java prezentăm următorul exemplu (a se vedea şi listingul *example2*):

```
Key key;
try {
   ObjectInputStream in = new ObjectInputStream(
   new FileInputStream("SecretKey.ser"));
   key = (Key)in.readObject();
   in.close();
} catch (FileNotFoundException fnfe) {
   KeyGenerator generator = KeyGenerator.getInstance("DES");
   generator.init(new SecureRandom());
   key = generator.generateKey();
   ObjectOutputStream out = new ObjectOutputStream(
   new FileOutputStream("SecretKey.ser"));
   out.writeObject(key);
   out.close();
}

// Get a cipher object.
Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");

// Encrypt or decrypt the input string.
if (args[0].indexOf("e") != -1) {
   cipher.init(Cipher.ENCRYPT_MODE, key);
   String amalgam = args[1];
   for (int i = 2; i < args.length; i++)
   amalgam += " " + args[i];
   byte[] stringBytes = amalgam.getBytes("UTF8");
   byte[] raw = cipher.doFinal(stringBytes);
```

```
   BASE64Encoder encoder = new BASE64Encoder();
   String base64 = encoder.encode(raw);
   System.out.println(base64);
}
else if (args[0].indexOf("d") != -1) {
   cipher.init(Cipher.DECRYPT_MODE, key);
   BASE64Decoder decoder = new BASE64Decoder();
   byte[] raw = decoder.decodeBuffer(args[1]);
   byte[] stringBytes = cipher.doFinal(raw);
   String result = new String(stringBytes, "UTF8");
   System.out.println(result);
}
```

Exemplul poate fi rulat folosind *ant run* sau folosind:

```
C:\java SecretWriting -e Hello, world!
Lc4WKHP/uCls8mFcyTw1pQ==
C:\java SecretWriting -d Lc4WKHP/uCls8mFcyTw1pQ==
Hello, world!
```

şi are ca rezultat criptarea (dacă se furnizează opțiunea –e) sau decriptarea (dacă se specifică opțiunea –d) unui text de intrare.

În Java orice operație de criptare/decriptare se foloseşte de un obiect *Cipher*. După cum se poate observa în exemplul anterior, un astfel de obiect *Cipher* se foloseşte în conjuncție cu o cheie. În cadrul exemplului se încearcă deserializarea cheii dintr-un fişier sau, dacă operația eşuează, se încearcă generarea unei noi chei. Un obiect *KeyGenerator* este folosit pentru generarea cheii. Un KeyGenerator se obține prin folosirea unei metode adecvate, în funcție de tipul de algoritm de criptare pe care dorim să îl folosim: *KeyGenerator generator = KeyGenerator.getInstance("DES");* (aici se încearcă obținerea unei chei corespunzătoare algoritmului DES – Data Encryption Standard). Un generator de chei trebuie să fie inițializat în prealabil cu un număr aleator pentru a produce o nouă cheie: *generator.init(new SecureRandom())*. După obținerea cheii, se continuă cu obținerea unui cifru folosind un algoritm corespunzător: *Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding")*. Apelul acesta specifică că dorim să folosim algoritmul DES şi câțiva parametrii suplimentari necesari acestuia. În finalul exemplului se poate observa modul în care se realizează criptarea sau decriptarea datelor.

În Java toate operațiile criptografice sunt structurate conform diagramei următoare:

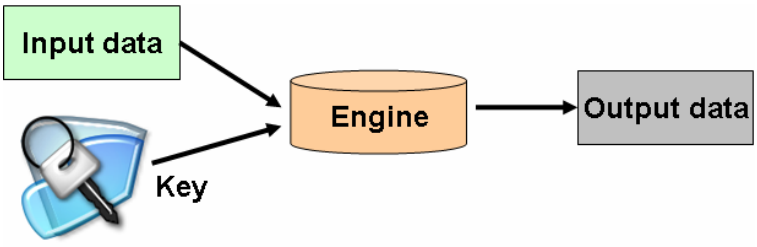

Figura 4. Mecanismul criptografic folosit pentru criptare/decriptare.

În centrul arhitecturii stă algoritmul criptografic folosit (engine). Algoritmul primeşte un set de date de intrare şi opțional o cheie şi produce un set de date de ieşire. Algoritmii criptografici existenți depind de providerii de securitate instalați în sistem. JCE de exemplu vine cu un provider de securitate suplimentar față de ce există instalat în JDK ce include implementările algoritmilor criptografici suportați.

În Java se pot instala mai mulți provideri de securitate în afară de cel instalat implicit de catre JCE. Exemplul următor instalează provider-ul oferit de Bouncy Castle care se numeste BC.

```
import java.security.Provider;
import java.security.Security;
import java.util.Set;
import java.util.Iterator;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

public class ProviderInformation {

   public static void main(String[] args) {
      Security.addProvider(new BouncyCastleProvider());
      Provider[] providers = Security.getProviders();
      for (int i = 0; i < providers.length; i++) {
         Provider provider = providers[i];
         System.out.println("Provider name: " + provider.getName());
         System.out.println("Provider information: " + provider.getInfo());
         System.out.println("Provider version: " + provider.getVersion());
         Set entries = provider.entrySet();
         Iterator iterator = entries.iterator();
         /*while (iterator.hasNext()) {
            System.out.println("Property entry: " + iterator.next());
         }*/
      }
   }
}
```

Implicit pentru toți algoritmii criptografici se foloseşte primul provider care apare în outputul programului de mai sus. Dacă se doreşte folosirea unui anumit provider se specifică explicit numele acestuia, de exemplu ”BC” pentru Bouncy Castle, precum în exemplul următor:

```
Signature sign = Siganture.getInstance("SHA1WithDSA", "BC");
```

În Java conceptul de cheie este modelat de interfața *java.security.Key*. O implementare de cheie depinde de tipul de algoritm de criptare folosit. Există de altfel două interfețe adiționale ce sunt folosite pentru specificarea cheilor în cazul algoritmilor asimetrici: *PublicKey* şi *PrivateKey*. Poate părea suprinzător, însă în JDK nu există mai departe nici o clasă care să implementeze oricare dintre aceste interfețe (din rațiuni de securitate). JCE furnizează un concept suplimentar față de JDK, noțiunea de *cheie secrete.* O cheie secretă reprezintă o cheie ce este partajată între două entități în timpul unei operații criptografice. Cheile secrete pot fi fie simetrice, fie asimetrice. Cheile asimetrice se folosesc de cele două interfețe anterior prezentate, în timp ce cheile simetrice implementează interfața *javax.crypto.SecretKey*.

O clasă ce este folosită pentru gestiunea cheilor asimetrice este *java.security.KeyPair*. Clasa reprezintă o structură de date simplă ce conține două informații importante: o cheie publică şi o cheie privată. Clasa poate fi folosită atât pentru încapsularea unor chei, cât deopotrivă ea este folosită de către generatorul de chei Java. Generarea cheilor asimetrice se face folosind clasa *java.security.KeyPairGenerator.* Clasa, abstractă de altfel, este apelabilă prin metodele statice *getInstance* puse la dispoziție, metode ce primesc ca argumente cel puțin un algoritm de folosit. După inițializare şi înainte de generearea perechii de chei este recomandată folosirea metodei *inițialize* pentru inițializarea corespunzătoare a generatorului de numere aleatoare. În final generarea cheilor se realizează prin metoda *generateKeyPair*, precum în exemplul următor:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
kpg.initialize(512);
KeyPair kp = kpg.generateKeyPair();
```

Pentru cheile simetrice JCE introduce mecanisme suplimentare de generare a acestora: clasele *KeyGenerator* şi *SecretKeyFactory.* Clasa *KeyGenerator*, folosită pentru generarea cheilor secrete simetrice, are o funcționalitate similară celei a clasei *KeyPairGenerator*. Diferența majoră constă în apelarea metodei *generateKey* ce rezultă în returnarea unui obiect de tip *SecretKey*. A doua clasă, *SecretKeyFactory*, converteşte o specificație de cheie algoritmică sau codată în obiecte cheie concrete. Clasa aceasta are un omolog pentru chei asimetrice în clasa *KeyFactory*. Un exemplu de folosire a acestei clase pentru exportarea pe disc a unei chei si, respectiv, importarea de pe disc a unei chei, este prezentat în listingul *example3*.

În plus utilizatorul are posibilitatea de a defini proprii mecanisme de criptografie. De exemplu, utilizatorul poate implementa o cheie secretă proprie bazată pe operația XOR precum în exemplul următor:

```
public class XORKey implements SecretKey {

   byte value;
```

```
   public XORKey(byte b) {
      value = b;
   }

   public String getAlgorithm() {
      return "XOR";
   }

   public String getFormat() {
      return "XOR Special Format";
   }

   public byte[] getEncoded() {
      byte b[] = new byte[1];
      b[0] = value;
      return b;
   }
}
```

O astfel de clasă poate fi folosită în conjuncție cu un generator de chei propriu utilizatorului ce poate fi definit conform exemplului următor (pentru exemplul complet se poate consulta listingul *example5*):

```
public class XORKeyGenerator extends KeyGeneratorSpi {

   SecureRandom sr;

   public void engineInit(SecureRandom sr) {
      this.sr = sr;
   }

   public void engineInit(AlgorithmParameterSpec ap, SecureRandom sr)
         throws InvalidAlgorithmParameterException {
      throw new InvalidAlgorithmParameterException(
            "No parameters supported in this class");
   }

   public SecretKey engineGenerateKey() {
      if (sr == null)
         sr = new SecureRandom();
      byte b[] = new byte[1];
      sr.nextBytes(b);
      return new XORKey(b[0]);
   }
}
```

Algoritmul de criptografie este implementat prin intermediul clasei *javax.crypto.Cipher*. Clasa furnizează o interfață pentru criptarea sau decriptarea atât a unor şiruri de date locale, cât şi pentru obținerea unor streamuri de date de criptare/decriptare. Numele algoritmilor funizați la inițializarea acestei clase diferă de cei folosiți la generarea cheilor de criptografie în sensul că se specifică, pe lângă numele algoritmului folosit, şi o serie de parametrii specifici funcționării respectivului algoritm: padding – la nivelul acestei clase se lucrează numai cu valori multipli de 8 octeți şi modul de operare (pentru mai multe detalii legate de

diferitele moduri de operare a se vedea documentația oficială JCE). Un exemplu de folosire a acestei clase cu şiruri de date locale am văzut anterior. De asemenea, în *cap5/example4* este prezentat un exemplu de folosire a PBE. Password-Based Encryption (PBE) crează o cheie de criptare dintr-o parolă. Pentru a minimaliza şansele ca un atacator să ghicească parola prin forță brută, implementările de PBE folosesc în plus un numar aleator (salt) pentru a crea cheia. Ca exemplu, pe orice sistem Unix care are pachetul *openssl* instalat se pot realiza scripturi care să realizeze PBE:

```
Pentru criptare folosind idea-cbc:
>openssl enc -idea-cbc -e -in plaintext_filename -out ciphertext_filename

Pentru decriptare:
>openssl enc -idea-cbc -d -in ciphertext_filename
```

În exemplul prezentat la începutul subcapitolului se poate observa modalitatea de folosire a clasei Cipher pentru criptarea unor variabile locale. Acelaşi aspect este reliefat şi în exemplul de criptare folosind parolă (algoritmul PBE). O altă modalitate de folosire a clasei Cipher este legată de contruirea unor stream-uri de date ce automatizează criptarea/decriptarea datelor pe măsură ce acestea sunt scrise în fişier, respectiv citite din fişier. Un exemplu de criptare a datelor folosind clasa *CipherOutputStream*, clasă ajutătoare pentru criptarea datelor într-un stream de comunicație, este prezentat în continuare:

```
public class Send {

   public static void main(String args[]) {
      try {
         KeyGenerator kg = KeyGenerator.getInstance("DES");
         kg.init(new SecureRandom());
         SecretKey key = kg.generateKey();

         Cipher c = Cipher.getInstance("DES/CFB8/NoPadding");
         c.init(Cipher.ENCRYPT_MODE, key);
         CipherOutputStream cos = new CipherOutputStream( new
               FileOutputStream("ciphertext"), c);
         PrintWriter pw = new PrintWriter( new OutputStreamWriter(cos));
         pw.println("Stand and unfold yourself");
         pw.close();

         SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
         Class spec = Class.forName("javax.crypto.spec.DESKeySpec");
         DESKeySpec ks = (DESKeySpec) skf.getKeySpec(key, spec);
         ObjectOutputStream oos = new ObjectOutputStream( new
                 FileOutputStream("keyfile"));
         oos.writeObject(ks.getKey());
         oos.writeObject(c.getIV());
         oos.close();
      } catch (Exception e) {
         System.out.println(e);
      }
   }
}
```

În cadrul exemplului se începe prin crearea unei chei secrete, folosind algoritmul DES şi clasa *KeyGenerator*. Ulterior este instanțiată clasa *Cipher* cu specificarea comportării algoritmului DES şi este inițializată cu modul de lucru (criptare în acest caz, specificată prin parametrul *Cipher.ENCRYPT_MODE*) şi cheia de folosit. Clasa *CipherOutputStream* este apoi inițializată peste un alt stream de ieşire şi pe baza obiectului *Cipher* anterior creat. Astfel, intern, orice dată ce va fi trimisă a fi scrisă pe streamul de ieşire va fi trecută prin *Cipher*, deci prin algoritmul de criptare. Ultimul pas, scrierea în fişierul keyfile, demonstrează o modalitate prin care putem transmite cheia de criptografie destinaturului, persoanei ce va realiza decriptarea, prin intermediul unui fişier. Acest pas este necesar în cazul unor algoritmi, cum este şi cazul algoritmului DES, unde nu putem genera o aceeaşi cheie pentru decriptare în mod automat, decriptarea necesitând existența informației legată de cheia de criptare.

În listingul *example4* se poate consulta un exemplu în care criptăm un text pornind de la o parolă text, caz în care nu aveam nevoie de cheie, aceasta putând fi întotdeauna unic generată pornind de la textul parolă cunoscut.

Funcționalitatea inversă, decriptarea datelor pe care le-am scris în fişier folosind tot un stream de date, de data aceasta de intrare, se realizează precum în exemplul următor:

```
public class Receive {

   public static void main(String args[]) {
      try {
         ObjectInputStream ois = new ObjectInputStream( new
                 FileInputStream("keyfile"));
         DESKeySpec ks = new DESKeySpec((byte[]) ois.readObject());
         SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
         SecretKey key = skf.generateSecret(ks);
         Cipher c = Cipher.getInstance("DES/CFB8/NoPadding");
         c.init(Cipher.DECRYPT_MODE, key,
                newIvParameterSpec((byte[]) ois.readObject());
         CipherInputStream cis = new CipherInputStream( new
                 FileInputStream("ciphertext"), c);
         cis.read(new byte[8]);
         BufferedReader br = new BufferedReader( new
                  InputStreamReader(cis));
         System.out.println("Got message: "+br.readLine());
         ois.close();
         br.close();
      } catch (Exception e) {
         System.out.println(e);
      }
   }
}
```

Ca o regulă de securizare a aplicațiilor, în general este indicat ca parolele să fie ținute în memorie nu ca String, ci ca array de caractere, şi trebuie suprascrise cu zero imediat după folosire pentru a preveni memory sau disk snooping. De

exemplu, informația ar putea să fie citită uşor atunci când maşina este obligată să facă swapping. De asemenea, atunci când se serializează obiecte este indicată folosirea cuvântului cheie *transient* pentru ca informația de pe aceste canale să nu fie trimisă în streamul de date.

4. Aplicație practică

Vă propunem scrierea unei aplicații client-server în care clientul decriptează dintr-un fişier un text, realizează o conexiune SSL cu serverul şi îi trimite acestuia şirul anterior citit din fişier. Pentru realizarea acestei aplicații parcurgeți pașii:

Task1.

Pornind de la listingul *example4* şi explicațiile prezentate în subcapitolul 3, realizați un program care citeşte parola de la tastatură şi realizează decriptarea unui text dintr-un fişier. Pentru crearea fişierului conținând textul criptat folosiți exemplul prezentat în cadrul clasei PBEnc.

Task2.

Creați un keystore conținând un certificat propriu semnat de către un CA. Realizați crearea unui trustore pe baza fişierului keystore obținut anterior. Pentru mai multe informații consultați subcapitolul 2.

Task3.

Realizați o aplicație client-server ce comunică folosind un canal de comunicație securizat cu protocolul SSL (lăsați *setNeedClientAuth*(*false*)). Folosiți în acest scop fişierele keystore şi trustore create în task-ul anterior. Consultați detaliile certificatelor primite în procesul de handshake. Trimiteți din partea clientului mesajul decriptat din fişier şi afişați-l la recepție pe partea serverului. Pentru detalii de implementare consultați subcapitolul 2.