Rozdział 10 Bezpieczeństwo w Javie

10.1 Bezpieczeństwo programów i danych

Termin "bezpieczeństwo" ("security") można przykładowo opisać poprzez funkcjonalność programów, które powinny być:
- wolne od wrogich podprogramów - program nie powinien utrudniać pracy w danym środowisku oraz powinien być wolny od niezamierzonych podprogramów np. wirusów;
- "nieinwazyjne" - programy nie powinny mieć dostępu do informacji prywatnych przechowywanych na komputerze lokalnym lub w sieci (kontrolowany dostęp do zasobów prywatnych),
- autoryzowane - programy powinny umożliwiać ustalenie i potwierdzenie tożsamości autora i danych,
- kodowane - programy powinny umożliwiać kodowanie danych,
- kontrolowane - programy powinny umożliwiać tworzenie rejestrów operacji związanych z zagrożeniem bezpieczeństwa (np. pliki typu *.log),
- sterowane - programy powinny być zabezpieczone przed wykorzystywaniem zbyt dużej ilości zasobów,
- certyfikowane - programy powinny spełniać wymagania i uzyskiwać certyfikaty bezpieczeństwa np. C2 czy B1 według systemu opracowanego dla rządu USA i innych.

10.2 Bezpieczeństwo w Javie

Środowisko Java 1.0 udostępniało pierwsze dwa elementy. Kolejne wersje Javy wprowadzały dodatkową funkcjonalność programów i tak JAVA 2 umożliwia spełnienie funkcjonalności zgodnie z pierwszymi pięcioma punktami, z tym, że kodowanie danych dostępne jest tylko dla obywateli USA i Kanady (ograniczenia eksportowe) poprzez rozszerzenie JCE (Java Cryptography Engine). Model bezpieczeństwa tworzony dla aplikacji Javy ewaluował poprzez kolejne wersję platformy. Począwszy od wersji 1.0 opracowano model, który później nazwano "piaskownicą". Odwołanie się do piaskownicy ukazuje analogię programu Javy do dziecka bawiącego się w piaskownicy. Dziecko w piaskownicy ma ograniczony dostęp do otoczenia (wyznaczany przez rozmiar piaskownicy) oraz ma do dyspozycji określone zabawki (zasoby). Tak samo program Javy umieszczony w danym systemie bezpieczeństwa ma takie możliwości jakie daje mu ten system. W wersji 1.0 programy Javy, lub bardziej ogólnie kody dzielone są na zaufane i niezaufane. Kod zaufany to ten, który znajduje się w lokalnym systemie; kod niezaufany to ten, który

pochodzi ze zdalnych urządzeń (np. z sieci). Kod zaufany ma pełny dostęp (pełny w sensie danego systemu operacyjnego - jeśli np. plik nie jest do odczytu przez danego użytkownika w danym systemie operacyjnym, to plik ten będzie również niedostępny przez kod Javy) do zasobów, kod niezaufany ma dostęp taki, na jaki zezwala mu "piaskownica". Standardowo wszystkie applety uruchamiane są w obrębie jakiejś aplikacji np. Netscape, HotJava, appletviewer; które tworzą "piaskownicę" dla danego appletu. W wersji Java1.0 wszystkie applety nie mają dostępu do zasobów lokalnych, co jest uwarunkowane przez model bezpieczeństwa tej platformy Javy. Jako ciekawostkę można podać, że przyczyną dla której nie tworzono dostępu dla appletów do lokalnych zasobów plików nie był aspekt bezpieczeństwa, lecz fakt, że docelowo w propagowanej przez Suna technologii komputerów sieciowych takich zasobów po prostu być nie powinno. Dla aplikacji uruchamianych lokalnie można również stworzyć programy, które będą ustalać dla nich "piaskownicę". Wówczas docelowa aplikacja będzie wywoływana jako atrybut programu ustalającego bezpieczeństwo. Można więc zarówno zdalnie jak i lokalnie ustalić prawa dostępu do zasobów plików, co jest niezwykle przydatne, szczególnie dla programów pracujących z danymi chronionymi (np. w medycynie). W wersji JDK 1.1 wprowadzono zmianę w modelu bezpieczeństwa polegającą na tym, że tzw. podpisany elektronicznie kod (aplet) jest traktowany jako kod zaufany, a co za tym idzie ma dostęp do lokalnych zasobów komputera. Podpisane kody wraz z kluczami są dostarczane do systemu w pliku spakowanym JAR. Platforma Java 2 wprowadza liczne zmiany związane z bezpieczeństwem. Nowy model bezpieczeństwa przyjmowany przez tą platformę zakłada, że każdy kod (lokalny i zdalny, podpisany czy nie) podlega kontroli zasad bezpieczeństwa (policy). W Javie 2 nie istnieje już wbudowany mechanizm dzielący kod na zaufany i niezaufany. Każdy kod podlega kontroli względem ustawień zasad bezpieczeństwa, które to zasady są spisywane w plikach konfiguracyjnych JRE lub definiowanych przez użytkownika plikach zasad bezpieczeństwa. Po kontroli zasad bezpieczeństwa dany kod uzyskuje określone uprawnienia wynikające z ustawień zasad bezpieczeństwa. Przykładowo aplet A podpisany prze X może mieć dostęp do plików w lokalnym systemie plików, dający mu prawo zapisu pliku o nazwie: pliktest. Domyślnie przyjęto, że zasady bezpieczeństwa dają uprawnienia programom tak jak to było w modelu 1.0, tzn. zasady bezpieczeństwa dla apletów uniemożliwiają im dostęp do lokalnego zasobu plików. Określone jest to poprzez fakt, że dla aplikacji nie instalowany jest automatycznie system zarządzania bezpieczeństwem, tak jak to ma miejsce dla apletów. W celu uruchomienia aplikacji z działającym systemem zarządzania bezpieczeństwem należy wywołać interpreter (java) z opcją -Djava.security.manager. Wówczas aplikacje podlegają takiej samej kontroli bezpieczeństwa jak aplety poprzez pliki konfiguracyjne ustawień bezpieczeństwa.
Java 2 wprowadza jako rozszerzenia pojęcia "piaskownica" pojęciem "dziedzina". Otóż w czasie wykonywania kodu, kody grupowane są w grupy kodów (pochodzących z tego samego źródła - CodeBase) o tych samych prawach dostępu. Grupy te zwane dalej dziedzinami oznaczają określony zakres dostępu do zasobów (według ustalonych zasad bezpieczeństwa). Ustalając więc zakres bezpieczeństwa można uzyskać informację (obiekt klasy PermissionCollection) wywołując metodę getPermissions() klasy Policy.

Podsumowując można wskazać następujące elementy związane z ustalaniem zakresu bezpieczeństwa dla aplikacji Javy:

- weryfikacja kodu: weryfikator B-kodu sprawdza czy klasy Javy spełniają prawa języka Java (klasy zestawu core API nie są sprawdzane);
- ładowanie klas: jeden lub więcej systemów ładowania klas wprowadza wszystkie klasy, które nie są częścią core API (do wersji Java 2, ładowanie klas wprowadzało klasy których nie było w ustawieniu CLASSPATH);
- kontrola dostępu: od Javy 2 kontroler dostępu umożliwia lub zabrania dostępu do systemu operacyjnego;
- zarządzanie bezpieczeństwem: system zarządzania bezpieczeństwem był szczególnie istotny w wersjach wcześniejszych niż 2, ponieważ zarządzał on dostępem do zasobów komputera. Od wersji Java 2 system zarządzania bezpieczeństwem odwołuje się bezpośrednio do kontrolera dostępu;
- pakiet java.security: umożliwia dostęp do powyższych systemów (poprzez szereg klas i interfejsów np. AccessController, SecurClassLoader) oraz dostarcza szeregu narzędzi do tworzenia podpisów cyfrowych, kluczy, certyfikatów;
- baza danych kluczy: zbiór kluczy używanych przez system zarządzania bezpieczeństwem i kontrolera dostępu do sprawdzania podpisów elektronicznych związanych z podpisanymi plikami klas Javy.

W Javie 2 należy w sposób szczególny omówić kontroler dostępu ponieważ większość zagadnień bezpieczeństwa jest z nim związana. Kontroler dostępu (AccessController) jest zbudowany w oparciu o cztery zasadnicze elementy:
- CodeSource; reprezentacja źródła pochodzenia kodu;
- Permissions; reprezentacja uprawnień dostępu;
- Policies; reprezentacja wszystkich uprawnień, które powinny być nadane dla danego kodu - określenie zasad bezpieczeństwa;
- Protection Domains; reprezentacja konkretnego źródła i wszystkich uprawnień mu nadanych.

10.3 Obsługa zasad bezpieczeństwa w Javie

Zasady bezpieczeństwa dla platformy Javy są spisane w plikach konfiguracyjnych. Podstawowym plikiem konfiguracyjnym jest plik java.security umieszczony standardowo w katalogu <JRE_home>/lib/security. Plik ten zawiera informacje o lokalizacji pliku zasad bezpieczeństwa, o możliwości tworzenia własnych (w katalogu domowym) plików zasad bezpieczeństwa, o możliwości uruchamiania JVM z dodatkowym plikiem bezpieczeństwa (-Djava.security.policy=własnyplikbezpieczeństwa), o typie magazynu kluczy oraz o dostawcy i lokalizacji pakietu mechanizmów bezpieczeństwa (np. algorytmy kodowania wiadomości, algorytmy generacji podpisów elektronicznych, itp.). Lokalizacja plików zasad bezpieczeństwa w pliku java.security jest ustalana poprzez podanie linii typu policy.url.n=URL; gdzie n jest kolejnym plikem zasad, a URL adresem URL do tego pliku. Pierwszym plikiem zasad bezpieczeństwa jest plik java.policy znajdujący się w tym samym katalogu co java.security. Ponieważ jest to pierwszy, domyślny plik zasad to w pliku java.security ustawiona jest następująca linia konfiguracji: policy.url.1=file:${java.home}/lib/security/java.policy. Jako drugi plik zasad bezpieczeństwa podaje się plik ustawień dla aktualnego użytkownika (z jego katalogu domowego) policy.url.2=file:${user.home}/.java.policy. Dalej można tworzyć dodatkowe pliki zasad bezpieczeństwa, które mogą być dodawane według omówionej zasady do pliku java.security. Podstawowy plik zasad bezpieczeństwa

(oraz inne) zawiera wskazanie na magazyn kluczy (koniecznych przy potwierdzaniu podpisanych kodów) oraz szereg pozycji zawierających CodeSource (źródło kodu), identyfikator podpisu (jeśli taki jest użyty), oraz zestaw uprawnień (Permissions). Poniżej zaprezentowano zawartość pliku standardowego java.policy:

```
// Standard extensions get all permissions by default

grant codeBase "file:${java.home}/lib/ext/-" {
 permission java.security.AllPermission;
};

// default permissions granted to all domains

grant {
 // Allows any thread to stop itself using the java.lang.Thread.stop()
 // method that takes no argument.
 // Note that this permission is granted by default only to remain
 // backwards compatible.
 // It is strongly recommended that you either remove this permission
 // from this policy file or further restrict it to code sources
 // that you specify, because Thread.stop() is potentially unsafe.
 // See "http://java.sun.com/notes" for more information.
 permission java.lang.RuntimePermission "stopThread";

 // allows anyone to listen on un-privileged ports
 permission java.net.SocketPermission "localhost:1024-", "listen";

 // "standard" properies that can be read by anyone

 permission java.util.PropertyPermission "java.version", "read";
 permission java.util.PropertyPermission "java.vendor", "read";
 permission java.util.PropertyPermission "java.vendor.url", "read";
 permission java.util.PropertyPermission "java.class.version", "read";
 permission java.util.PropertyPermission "os.name", "read";
 permission java.util.PropertyPermission "os.version", "read";
 permission java.util.PropertyPermission "os.arch", "read";
 permission java.util.PropertyPermission "file.separator", "read";
 permission java.util.PropertyPermission "path.separator", "read";
 permission java.util.PropertyPermission "line.separator", "read";

 permission java.util.PropertyPermission "java.specification.version", "read";
 permission java.util.PropertyPermission "java.specification.vendor", "read";
 permission java.util.PropertyPermission "java.specification.name", "read";

 permission java.util.PropertyPermission "java.vm.specification.version", "read";
 permission java.util.PropertyPermission "java.vm.specification.vendor", "read";
 permission java.util.PropertyPermission "java.vm.specification.name", "read";
 permission java.util.PropertyPermission "java.vm.version", "read";
 permission java.util.PropertyPermission "java.vm.vendor", "read";
 permission java.util.PropertyPermission "java.vm.name", "read";
};
```

Ustawienia plików zasad bezpieczeństwa odbywają się poprzez edycję tekstowych plików za pomocą edytora lub poprzez standardowe narzędzie dostarczane z Java 2

tzn. policytool. Policytool jest aplikacją JAVY umożliwiającą tworzenie i edycję plików zasad bezpieczeństwa. Zanim zaprezentowane zostaną przykłady należy odnieść się do wymienionych wcześniej elementów kontrolera dostępu. W świetle znanych już metod ustalania zasad bezpieczeństwa w plikach konfiguracyjnych łatwo wytłumaczyć znaczenie czterech wymienionych elementów kontrolera dostępu. Po pierwsze obiekt klasy CodeBase reprezentuje kod (źródło), który jest lub będzie związany z określonymi uprawnieniami. Po drugie dla abstrakcyjnej klasy Permission istnieją końcowe klasy oznaczające konkretne uprawnienia. Te końcowe klasy uprawnień charakteryzują się trzema cechami: typem - oznaczającym w nazwie klasy typ uprawnień z nią związanych np. FilePermission, SocketPermission; nazwą - określającą cel z którym związane są dane uprawnienia (element String oznaczający URL, plik, itp.); akcją – określającą rodzaj uprawnień dla danego typu (np. dla FilePermission: read, write, execute,delet; dla SocketPermission: accept, listen, connect, resolve). Obiekt klasy końcowej np. FilePermission nie nadaje uprawnień w systemie operacyjnym (np. plikowi nie nadawane są uprawnienia w systemie operacyjnym lecz w środowisku Javy). Klasy końcowych uprawnień (oprócz BasicPermission-przydatna przy implementacji własnych typów uprawnień- i SecurityPermission) znajdują się poza pakietem java.security i są umieszczane w pakietach związanych z typem uprawnień np. FilePermission jest w pakiecie java.io; SocketPermission jest w pakiecie java.security, itd. Ponieważ elementarne uprawnienia mogą być mnogie dla danego typu uprawnień (np. ustawiany jest dostęp do odczytu plików z katalogu /tmp oraz drugie uprawnienie dostęp do odczytu plików z katalogu /java/test) stworzono klasę PermissionCollection, której obiekty reprezentują zbiór uprawnień.
Obiekty klasy Permissions reprezentują natomiast kolekcję obiektów klasy PermissionCollection. W ten sposób można pogrupować różne uprawnienia, które później zostaną przydzielone konkretnemu kodowi w celu stworzenia dziedziny uprawnień. Po trzecie klasa Policy ustala odwzorowanie pomiędzy kodem źródłowym a uprawnieniami. Obiekt klasy Policy jest inicjowany głównie poprzez interpretacje plików zasad bezpieczeństwa (java.policy). Zasady bezpieczeństwa są więc ustalane zewnętrznie względem kodu. Po czwarte przydzielenie konkretnemu kodowi CodeSource grupy uprawnień Permissions powoduje stworzenie dziedziny uprawnień (obiekt klasy ProtectionDomain). Obiekt klasy ProtectionDomain reprezentuje więc scaloną część w pliku konfiguracyjnym zasad bezpieczeństwa wydzieloną poprzez fragment: grant CodeBase SignedBy {Permissions}(przykładowo w powyższym pliku java.policy).
Omawiając elementy związane z uprawnieniami w Javie warto podkreślić elastyczność określania celu działania danego uprawnienia. Cel działania danego uprawnienia np. FilePermission może być ustalony poprzez podanie konkretnego elementu lub poprzez wykorzystanie różnych znaków zastępczych. Przykładowo dla uprawnień FilePermission możliwe są następujące kody sterujące:

file (oznacza konkretną nazwę pliku w bieżącym katalogu);
directory (oznacza konkretną nazwę katalogu);
directory/file (oznacza konkretną nazwę pliku w danym katalogu);
directory/* (oznacza wszystkie pliki w danym katalogu);
* (oznacza wszystkie pliki w bieżącym katalogu);
directory/- (oznacza wszystkie pliki w danym katalogu i jego podkatalogach);
- (oznacza wszystkie pliki w bieżącym katalogu i jego podkatalogach);
"<<ALL FILES>>" (oznacza wszystkie pliki w systemie plików).

Podobnie ustalane są różne znaki sterujące dla innych uprawnień. Widać stąd, że sterowanie uprawnieniami jest bardzo elastyczne. Implikację ustawiania uprawnień (np. dla pliku /tmp/jacek/test.java gdy ustalone są uprawnienia dla /tmp/-) powodują implementacje metody abstrakcyjnej klasy Permission w odpowiednich końcowych klasach uprawnień np. FilePermission.
Ustalając zasady nadawania uprawnień należy rozpatrzyć jeszcze jeden mechanizm. Otóż często zdarza się, że użytkownik chce aby jego klasa miała tymczasowo dane uprawnienia w imieniu klasy, która takich uprawnień nie ma. Przykładowo chcemy aby możliwość tworzenia gniazd (Socket) dana była tylko klasom pochodzącym z węzła sieci komputerowej, a nie bezpośrednio klasom pochodzącym z poszczególnych działów firmy. Uprzywilejowanie kodu polega na stworzeniu obiektu interfejsu PrivilegedAction lub PrivilegedExceptionAction i poinformowaniu kontrolera dostępu o tym fakcie poprzez wywołanie metody statycznej AccessController.doPvivileged() z argumentem będącym stworzonym właśnie obiektem. Kod uprzywilejowany umieszcza się w ciele jedynej metody interfejsów PrivilegedAction i PrivilegedExceptionAction, a mianowicie metody run(). Przykładowa konstrukcja wygląda następująco:

```
jakasmetoda() {
        ...
        AccessController.doPrivileged(new PrivilegedAction() {
           public Object run() {
              // uprzywilejowany kod np.:
              System.loadLibrary("awt");
              return null; // nic nie zwraca, ale zawsze musi występować
           }
        });
        ...
}
```

Oznaczając dany kod jako uprzywilejowany (privileged) nadaje się mu tymczasowo dostęp do zasobów, do których ma ustawione uprawnienia (permission), a do których nie ma uprawnień kod, który go wywołuje. Pytanie, które się natychmiast pojawia jest po co uprzywilejować kod, który ma uprawnienia? Otóż jest to istotne w kontekście kodu nieuprzywilejowanego: ładowanie kodu odbywa się do pamięci; ostatnio przywołany kod znajduje się najwyżej; kontroler dostępu wywołany jawnie lub nie, sprawdza uprawnienia (checkPermissions) poszczególnych kodów; jeżeli choć jeden fragment kodu nie ma określonych uprawnień wedle zasad bezpieczeństwa zwracany jest wyjątek AccessControlException, chyba że dany fragment jest zaznaczony jako uprzywilejowany; jeżeli tak jest to kontroler dostępu sprawdza czy zaznaczony kod ma uprawnienia do wykonania danej operacji i jeśli jest to prawdą to zakończony jest proces sprawdzania dostępu - nie sprawdza się dalszego kodu. Podsumowując: Jeżeli kod A o uprawnieniach ustawionych w plikach zasad bezpieczeństwa wywołuje kod B, którego uprawnienia są większe niż kodu A (tzn. kod A nie ma ustawionych takich uprawnień w plikach jak kod B), to wówczas przy ładowaniu kodu (najpierw ładowany jest A potem B, najpierw sprawdzany jest B potem A) nastąpi wyjątek, chyba że oznaczymy B jako kod uprzywilejowany, czyli kod B nie będzie rzutował na A. Jeżeli z każdym kodem jest związana dziedzina

uprawnień i jest tych dziedzin (kodów) m to proces sprawdzania kodu przebiega następująco:

```
i = m;
while (i > 0) {
   if (dziedzina kodu i nie posiada danych uprawnień)
      throw AccessControlException
   else if (kod zaznaczony jako uprzywilejowany)
      return; // koniec sprawdzania
   i = i - 1;
};
```

Zanim rozpatrzone zostaną zagadnienia kryptografii w Javie warto przedstawić kilka przykładów do powyższych rozważań. Powiedziane zostało, że w modelu bezpieczeństwa Java 2, każdy kod (dla aplikacji - o ile działa system zarządzania bezpieczeństwem) podlega kontroli wedle plików zasad bezpieczeństwa. Standardowe ustawienie bezpieczeństwa w pliku java.policy, które przedstawiono powyżej, umożliwia uruchomienie gniazda na portach wyższych niż 1024 (na portach użytkownika):

```
// allows anyone to listen on un-privileged ports
permission java.net.SocketPermission "localhost:1024-", "listen";
```

Ustawienie to uniemożliwia stworzenie serwera na portach mniejszych niż 1024. Oznacza to, że prezentowana wcześniej aplikacja SerwerEcho (w rozdziale poświęconym pracy w sieci) nie może działać przy wywołaniu jej z uruchomionym systemem zarządzania bezpieczeństwa (włączenie sprawdzania uprawnień). W celu sprawdzenia działania uprawnień należy wywołać:

```
java -Djava.security.manager SerwerEcho
```

Zwrócony zostanie wyjątek AccessControlException z podaniem typu uprawnień, których brak.
Warto pamiętać, że nie o wszystkich wyjątkach użytkownik programu może wiedzieć, ponieważ wyjątki mogą być obsługiwane w programie bez zwracania komunikatu.

W celu zapoznania się z mechanizmem ustawiania uprawnień za pomocą narzędzia policytool przedstawiony zostanie następujący przykład appletu:

Przykład 10.1:

```
//Zapisz.java:

import java.awt.*;
import java.io.*;
import java.lang.*;
import java.applet.*;

public class Zapisz extends Applet {
   String plik = "test";
```

```
   File f = new File(plik);
   DataOutputStream strumien_wy;

  public void init() {

   String osname = System.getProperty("os.name");
  }

  public void paint(Graphics g) {
       try {
         strumien_wy = new DataOutputStream(new BufferedOutputStream(new FileOutputStream(plik),128));
         strumien_wy.writeChars("Niespodzianka: Applet zapisal ten plik !!!\n");
         strumien_wy.flush();
         g.drawString("Zapisano plik " + plik + "; sprawdz!!! ", 10, 10);
       }
       catch (SecurityException e) {
         g.drawString("Applet Zapisz spowodowal blad bezpieczenstwa: " + e, 10, 10);
       }
       catch (IOException ioe) {
             g.drawString("Applet Zapisz spowodowal blad we/wy", 10, 10);
       }
   }
}// koniec public
```

Zapisz.html:

```
<HTML>
<HEAD>
   <TITLE>test Zapisz</TITLE>
</HEAD>
<BODY>
To jest przykładowy Applet:
<CENTER><APPLET CODE="Zapisz.class" WIDTH=600 HEIGHT=200
ALIGN=CENTER></APPLET></CENTER>
</BODY>
</HTML>
```

Applet ten tworzy plik tekstowy o nazwie "test" z zapisanym przykładowym tekstem:"Niespodzianka: Applet zapisal ten plik !!!". W przypadku domyślnych ustawień zasad bezpieczeństwa applet ten nie zapisze jednak pliku lecz zwróci wyjątek SecurityException. Można to sprawdzić wywołując skompilowany kod appletu przez:

```
appletviewer Zapisz.html
```

W celu ustawienia konkretnych uprawnień dla danego kodu należy wywołać narzędzie policytool poprzez wpisanie:

```
policytool
```

Pojawi się okno aplikacji Javy z dwoma polami Policy File oraz Keystore (które są puste jeśli nie ma ustawień użytkownika) oraz trzema przyciskami umożliwiającymi edycję plików uprawnień. W celu stworzenia nowego pliku uprawnień i nadania

uprawnień dla zapisu pliku o nazwie "test" należy wcisnąć klawisz Add Policy Entry, co spowoduje otwarcie nowego okna dialogowego, w którym można wpisać uprawnienia dla danego kodu. W nowym oknie dialogowym najpierw należy wybrać źródło kodu, któremu nadajemy uprawnienia (Code Base). Zostawiając to pole puste ustawimy uprawnienia dla dowolnego kodu.
Następne pole SignedBy umożliwia ustawienie identyfikacji jednostki autoryzującej dany kod. Ponownie zostawiając to pole puste ustawimy każdy kod jako autoryzowany. Teraz można ustalić typy dostępu. W tym celu wybrać należy przycisk AddPermission co spowoduje otwarcie trzeciego okna dialogowego z możliwością wyboru typu uprawnień z listy: Permission; oznaczenia celu działania danego uprawnienia (np. nazwa pliku) w polu Target Name, oraz wskazania typu akcji właściwej dla danego uprawnienia w polu Actions. Dla potrzeb tego przykładu należy wybrać z listy Permission:FilePermission, wpisać obok TargetName nazwę pliku czyli "test" oraz typ akcji "write". Po potwierdzeniu utworzenia nowych uprawnień (OK w oknie Permissions; Done w oknie Policy Entry) należy zapisać uprawnienia jako nowy plik np. moje_policy, poprzez wybór polecenia SaveAs z Menu okna Policy Tool. Po nagraniu pliku zasad bezpieczeństwa pojawi się w polu Policy File okna PolicyTool nazwa nowo stworzonego pliku bezpieczeństwa wraz ze ścieżką dostępu (URL). Po wykonaniu tych czynności należy zakończyć pracę z aplikacją PolicyTool i ponownie wywołać applet Zapisz.java poprzez:

```
appletviewer -J-Djava.security.policy=moje_policy Zapisz.html
```

Opcja -J-Djava.security.policy= umożliwia wywołanie danego appletu z tymczasowo przyjętym plikiem zasad bezpieczeństwa. Oczywiście uprawnienia w pliku moje_policy można przenieść do domyślnego pliku w swoim katalogu domowym lub można utworzyć w pliku java.security następne odwołanie do pliku zasad bezpieczeństwa poprzez wpisanie policy.url.3=file:/C:/sciezkadostepu/moje_policy. Wówczas nie trzeba podawać opcji dodatkowego pliku bezpieczeństwa, gdyż będzie on sprawdzany automatycznie. Uwaga: W ustawieniach lokalizacji plików istotna jest różnica zapisu ścieżki dostępu do danych zasobów w systemie plików właściwych dla danego systemu operacyjnego.

Uruchomienie appletu Zapisz.java wraz z dodatkowym plikiem zasad bezpieczeństwa spowoduje nagranie pliku "test"do lokalnego systemu plików.

10.4 Kryptografia

W celu omówienia możliwości zastosowania kryptografii w środowisku języka JAVA należy najpierw omówić podstawowe zagadnienia związane z kryptografią.
Do podstawowych zadań kryptografii należy zaliczyć:
- zapewnienie autentyczności autora obiektu (np. danych) - autoryzacja;
- zapewnienie autentyczności obiektu (np. danych);
- kodowanie danych.

W celu zapewnienia autentyczności autora obiektu np.danych czy kodu, należy stworzyć takie mechanizmy, aby można było stworzyć unikalny identyfikator wskazujący autora obiektu. Najczęstszym rozwiązanie tego problemu jest stworzenie dla autora unikalnego klucza, którym będzie on "zamykał" wszystkie generowane

przez siebie obiekty. Popularne klucze służące kodowaniu wykorzystywano już od dawna. Przykładowo książka oraz film "Klucz do Rebeki" o działaniach wywiadowczych w czasach II Wojny Światowej przedstawia wykorzystanie książki jako klucza do szyfrowania wiadomości. Obie zainteresowane strony: wywiadowca i centrala, posiadają tą samą książkę, która stanowi tłumaczenie odnośników przesyłanych pomiędzy stronami. Oznacza to, że zarówno nadawca jak i odbiorca posiadają dobrze im znany klucz. Jeżeli taki klucz (np. taka książka) jest unikalny i występuje tylko u odbiorcy i nadawcy, to wówczas odbiorca wie kto nadał daną wiadomość. Ta forma klucza nazywa się kluczem sekretnym, co oznacza, że klucz musi być ukryty zarówno przez nadawcę jak i odbiorcę. O wiele bardziej popularne obecnie jest wykorzystanie dwóch różnych kluczy dla potrzeb kryptografii: klucza prywatnego (sekretnego) oraz publicznego.
Idea jest prosta. Autor generuje parę ściśle ze sobą związanych kluczy, a następnie rozsyła wszystkim tym, którzy z nim współpracują klucze publiczne. Klucz prywatny autor przechowuje tak starannie jak klucz sekretny. Autor nadając wiadomość tworzy jej skrót (message digest - skrót wiadomości) korzystając z konkretnego algorytmu (tworzony jest zestaw cyfr, stąd często funkcje generujące skrót nazywa się funkcjami haszującymi - # - to oznaczenie cyfry). Następnie kod jest kodowany za pomocą klucza prywatnego autora. W ten sposób powstaje cyfrowy podpis wiadomości. Sprawdzenie cyfrowego podpisu wiadomości przez odbiorcę odbywa się poprzez ponowne stworzenie skrótu wiadomości a następnie zakodowaniu tego skrótu za pomocą klucza publicznego nadawcy. Jeżeli podpis wygenerowany przez nadawcę i podpis wygenerowany przez odbiorcę są zgodne to oznacza, że istotnie wiadomość nadał znany odbiorcy nadawca. Oprócz identyfikacji nadawcy proces ten zapewnia jeszcze jedną korzyść- zapewnienie autentyczności wiadomości. Otóż wygenerowany skrót wiadomości jest unikalny (powinien być), co oznacza, że dowolna zmiana wiadomości spowoduje zmianę skrótu, a co za tym idzie wystąpi niezgodność podpisów, czyli brak potwierdzenia autentyczności autora i danych. W celu zwiększenia efektywności zarządzania kluczami (gdzie trzymać klucze? czyje to są klucze?...) wprowadza się dwa podstawowe elementy: bazy kluczy oraz certyfikaty. Najlepsze rozwiązanie to bazy certyfikatów czyli zbiorów przechowujących: klucz publiczny nadawcy, oznaczenie nadawcy (imię, nazwa, ...), podpis cyfrowy wydawcy certyfikatu (np. odbiorca generuje podpis cyfrowy danych (klucz publiczny nadawcy) na podstawie swojego klucza prywatnego) oraz oznaczenie wydawcy certyfikatu (imię, nazwa). Zbiory certyfikatów są przechowywane i wymieniane w zależności od potrzeb. Sprawdzenie wiarygodności certyfikatu odbywa się poprzez sprawdzenie podpisu cyfrowego w nim zawartego lub często tworzy się "odcisk" certyfikatu czyli skrót certyfikatu i porównuje się skróty posiadanego certyfikatu i certyfikatu od wystawcy.
Klucze mają oczywiście jeszcze jedno podstawowe wykorzystanie - kodowanie danych. Otóż do tej pory omówione zostały mechanizmy zapewnienia autentyczności autora i obiektu - dane były nie kodowane. Dane można bezpośrednio kodować kluczem sekretnym (prywatnym), natomiast odbiorca może dekodować dane za pomocą klucza publicznego. Rozwiązanie to nie jest jednak zbyt dobre ponieważ wówczas każdy kto posiada klucz publiczny (a ponieważ jest on publiczny nie ma ograniczeń w dostępie) może odkodować wiadomość, czyli właściwie kodowanie danych nie przynosi spodziewanych rezultatów. Do kodowania danych wykorzystuje się jednak mechanizm odwrotny. Mając klucz publiczny odbiorcy kodujemy dane. Tak zakodowane dane odkodować może tylko właściciel klucza prywatnego odpowiadającego kluczowi publicznemu użytemu do kodowania wiadomości.

Do tej pory wskazane zostały pewne mechanizmy kryptografii, bez wymieniania metod matematycznych służących do kodowania, generacji skrótów czy kluczy. Praktycznie w kryptografii operuje się algorytmami, czyli konkretnie zrealizowanymi metodami. Algorytmy dostarczane są w odpowiednich pakietach (jar, zip) dostarczanych przez właściwego dostawcę (Provider). Korzystając z odpowiedniego algorytmu należy wskazać z jakiego pakietu (jeśli nie jest to pakiet domyślny) powinien być zastosowany dany algorytm.

10.4.1 Kryptografia w Javie

Począwszy od JDK 1.1 pojawiły się w javie mechanizmy kryptografii. W Javie 2 rozszerzono te mechanizmy. Wszystkie mechanizmy podzielono funkcjonalnie na dwie architektury: JCA - Java Cryptography Architecture oraz JCE - Java Cryptography Extension. JCA stanowi integralną część Java 2 API i umożliwia podstawowe operacje celem zapewnienia autentyczności autora i danych. Obsługuje więc następujące mechanizmy: skróty wiadomości (MessageDigest), podpisy cyfrowe (Signatures), certyfikaty (Certificates), generacja kluczy (KeyPairGenerator), przechowywanie kluczy (KeyStore). JCE natomiast jest rozszerzeniem Java 2 API i głównie służy do kodowania danych (a także m.in. do generacji i obsługi kluczy sekretnych). Dostęp do JCE niestety jest ograniczony ze względu na prawo eksportowe USA. Korzystać z tej biblioteki mogą jedynie obywatele USA i Kanady. Praktycznie rzecz ujmując warto omówić tylko pierwszy zestaw mechanizmów czyli JCA.
Jak wspomniano wcześniej wszystkie mechanizmy kryptografii bazują na konkretnych algorytmach wykorzystywanych do obliczeń (kodowania). Zestaw algorytmów i narzędzi jest pojmowany w Javie jako pakiet dostawcy (Cryptographic Service Provider package lub w skrócie provider package). Pakiet dostawcy wykorzystywany w danym środowisku Javy musi być zarejestrowany w odpowiednim pliku konfiguracyjnym tj. w pliku java.security, według składni:

security.provider.n=masterClassName

Standardowym pakietem wykorzystywanym dla potrzeb JCA jest pakiet "SUN" dostarczany wraz z dystrybucją JDK. Domyślnie więc w pliku java.security wystąpi zapis:

security.provider.1=sun.security.provider.Sun

W pakiecie "SUN" znajdują się implementacje różnych metod kryptografii: DSA (według NIST FIPS 186), MD5 i SHA-1 dla skrótów wiadomości, generator kluczy DSA, generator pseudo-losowy SHA1PRNG (według IEEE P1363), certyfikaty według X.509, JKS (nazwa implementacji zarządzania kluczami - keystore). Oczywiście można stosować dodatkowe pakiety dostawcy algorytmów kryptografii, przy czym przy inicjowaniu obiektu danej klasy mechanizmu kryptografii poprzez statyczną metodę getInstance() należy dodatkowo podać nazwę pakietu dostawcy. Przykładowo tworząc obiekt klasy MessageDigest można posłużyć się metodą public static MessageDigest getInstance(String algorithm, String provider) throws NoSuchAlgorithmException, NoSuchProviderException. Nazwę algorytmu oraz

nazwę dostawcy podaje się jako łańcuch znaków. Istotne jest właściwe rozróżnienie wielkości znaków gdyż metoda jest czuła na wielkość liter i w przypadku błędu zostanie zwrócony wyjątek. Praca z mechanizmami kryptografii w Javie zaczyna się więc prawie zawsze od inicjowania obiektu danego mechanizmu wywołując statyczną metodę getInstance z podaniem nazwy algorytmu i (jeśli to konieczne) nazwy pakietu. Sposób taki na realizację algorytmów kryptografii w programowaniu jest niezwykle uniwersalny. Pozwala bowiem na uniezależnienie się w znaczny sposób w kodzie programu Javy od tworzonych i rozwijanych algorytmów kryptografii. Dalsze przykłady w tym przewodniku będą opierały się na pakiecie dostawcy kryptografii "SUN".

10.4.2 Skróty wiadomości

Pierwszym mechanizmem kryptografii w Javie, jaki tu zostanie przedstawiony, będzie tworzenie skrótów wiadomości. Do tego celu stworzono klasę MessageDigest wraz z odpowiednimi jej metodami. W celu zainicjowania obiektu tej klasy (podobnie jak w innych przypadkach) wykorzystuje się statyczną metodę getInstance(), np.:

```
MessageDigest md = MessageDigest.getInstance("SHA")
MessageDigest md = MessageDigest.getInstance("MD5").
```

Następnym krokiem po inicjalizacji jest dostarczenie obiektowi danych do stworzenia skrótu. Odbywa się to poprzez podanie tablicy bajtów jako argumentu do jednej z metod update(), np. jeżeli dane tworzą tablicę bajtów t1, wówczas:

```
md.update(t1);
```

jeśli dane tworzą dwie tablice t1 i t2 to:

```
md.update(t1);
md.update(t2);
```

i tak dalej.
Obliczenie skrótu jest ostatnim krokiem i odbywa się na skutek wywołania metody digest(), np.

```
byte  skrot[];
skrot = md.digest();
```

Jeśli jest jedna tablica danych wówczas można skomasować dwa ostatnie kroki i wywołać inną metodę digest() z argumentem będącym tablicą danych np.:

```
skrot = md.digest(t1);
```

Jak pokazano na przykładach metoda digest() zwraca tablicę bajtów będącą właściwym skrótem wiadomości.

Poniżej przedstawiono przykładowy program umożliwiający stworzenie skrótu, wyświetlenie go oraz zapis wraz z wiadomoscią do pliku.

Przykład 10.2:

```
//ZapiszMD.java:

import java.io.*;
import java.security.*;

public class ZapiszMD {

 public static void main(String args[]){
  try{
   FileOutputStream strumien_wy=new FileOutputStream("testMD");
   MessageDigest md = MessageDigest.getInstance("SHA");
   ObjectOutputStream obiekt_wy = new ObjectOutputStream(strumien_wy);
   String wiadomosc = "Czy to sen? Czy to sun? Czy to Java?";
   byte buf[] = wiadomosc.getBytes();
   md.update(buf);
   byte skrot[] = md.digest();
   System.out.println("Oto skrot:"+skrot);
   obiekt_wy.writeObject(wiadomosc);
   obiekt_wy.writeObject(skrot);
  } catch(Exception e){
   System.out.println(e);
  }

 }
}// koniec public class ZapiszMD
```

Program ten zapisze w formie obiektów dwa elementy: dane oraz skrót. Powstały plik zostanie wykorzystany w poniższym przykładzie do weryfikacji danych na podstawie skrótu:

Przykład 10.3:

```
//CzytajMd.java:

import java.io.*;
import java.security.*;

public class CzytajMD {

 public static void main(String args[]){
  try{
      FileInputStream strumien_we=new FileInputStream("testMD");
      ObjectInputStream obiekt_we = new ObjectInputStream(strumien_we);
      Object ob = obiekt_we.readObject();
      if(!(ob instanceof String)) {
          System.out.println("Niewlasciwe dane w pliku");
          System.exit(-1);
      }
      String wiadomosc_we = (String) ob;
      String wiadomosc = "Czy to sen? Czy to sun? Czy to Java?";
      System.out.println("Oto wiadomosc otrzymana: "+wiadomosc_we);
      System.out.println("Oto wiadomosc oryginalna: "+wiadomosc);
```

```
      ob = obiekt_we.readObject();
      if(!(ob instanceof byte[])) {
          System.out.println("Niewlasciwe dane w pliku");
          System.exit(-1);
      }
      byte skrot_we[] = (byte []) ob;
      MessageDigest md = MessageDigest.getInstance("SHA");
      byte buf[] = wiadomosc_we.getBytes();
      md.update(buf);
      byte skrot[] = md.digest();
      System.out.println("Oto skrot otrzymany:"+skrot_we);
      System.out.println("Oto skrot wg danych otrzymanych:"+skrot);
      if (MessageDigest.isEqual(skrot, skrot_we))
          System.out.println("Wiadomosc niezmieniona");
      else
          System.out.println("Wiadomosc zmieniona");

    } catch(Exception e){
      System.out.println(e);
   }
  }
}// koniec public class CzytajMD
```

Program czytający z pliku dwa obiekty a następnie formatujący je do obiektów typu String (dla wiadomości) oraz byte[] (dla skrótu) umożliwia sprawdzenie czy przesłany skrót odpowiada przesłanej wiadomości. Dla celów edukacyjnych przedstawiony wyżej program zna oryginalną wiadomość stąd prezentacja na ekranie wiadomości oryginalnej i przesłanej upewnia w poprawnym użyciu skrótu. Warto przećwiczyć powyższe programy zmieniając nieznacznie (np. 1 literę) wiadomość, nie zmieniając skrótu (np. poprzez nagranie innego obiektu typu String niż wiadomość np. "Czy to san? Czy to sun? Czy to Java?" do pliku wraz z oryginalnym skrótem dla właściwej wiadomości, lub poprzez edycję pliku binarnego np. program "Dos Navigator" ->Edit i zmianę odpowiedniej wartości np. 65 na 61 czyli e w a). Mając plik ze zmienioną wiadomością i oryginalnym skrótem można wywołać ponownie program CzytajMD. W rezultacie otrzyma się informację, że wiadomość została zmieniona.

10.4.3 Kod autentyczności wiadomości - MAC

Powyższa, podstawowa implementacji klasy MessageDigest nie umożliwia jednak dobrego zabezpieczania autentyczności wiadomości. Związane jest to z tym, że ewentualny hacker może zmienić zarówno wiadomość jak i skrót. Oznacza to, że może on wprowadzić swoją wiadomość i skrót dla niej stworzony. Wówczas odbiorca nie znając wiadomości przekonany będzie, że zgodnie z poprawnością skrótu wiadomość jest oryginalna. W celu poprawy bezpieczeństwa autoryzacji danych wprowadza się tzw. kod autentyczności wiadomości (MAC - Message Authentication Code), będący przetworzonym skrótem. W Javie nie ma ściśle sprecyzowanych mechanizmów generacji kodu MAC jest więc wiele możliwych rozwiązań. Jedno z rozwiązań to kodowanie za pomocą kluczy skrótu, inne to używanie przez dwie zainteresowane strony (nadawca i odbiorca) tajnej frazy szyfrującej skrót. Frazę stanowi łańcuch znaków, który miesza się z oryginalną wiadomością. Przykładowo w celu zmodyfikowania programu zapisującego ZapiszMD.java w celu generacji i

zapisu kodu MAC zamiast skrótu należy przykładowo wykonać następujące polecenia:

```
...
String fraza = "To jest moja fraza";
byte f[] = fraza.getBytes();
...
md.update(f);
md.update(buf);  //buf to tablica bajtów według wiadomości
byte skrot[] =md.digest();
md.update(f);
md.update(skrot);
byte MAC[] = md.digest();

...
obiekt_wy.writeObject(MAC);
```

Powyższe rozwinięcie programu ZapiszMD.java generuje skrót będący interpretacją wiadomości i frazy, następnie tak powstały skrót jest traktowany jako zbiór danych, na podstawie którego (ponownie wraz z frazą) generuje się nowy skrót stanowiący kod autentyczności MAC. Jeżeli odbiorca zna frazę może (odpowiednio przetwarzając program CzytajMD.java) sprawdzić autentyczność wiadomości. Ponieważ hacker nie zna frazy (nie powinien) nie może zmodyfikować wiadomości tak, aby odbiorca o tym nie wiedział.

10.4.4 Klucze i podpis cyfrowy

Jak wspomniano wcześniej rozwinięciem skrótu jest podpis cyfrowy. Podpis cyfrowy (Signature) jest niczym innym jak skrótem, kodowanym kluczem prywatnym. Istotna jest więc tutaj znajomość pracy z kluczami. W JCA stosowane są głównie pary kluczy prywatny-publiczny. Ponieważ jednak JCE umożliwia stosowanie kluczy sekretnych, w Javie core API wprowadzono elementy dla pojedynczych kluczy. I tak nadrzędnym interfejsem jest interfejs Key definiujący metody służące do oznaczenia danego klucza (np. typ algorytmu, format. kodowanie). Ponieważ Sun w swej dystrybucji Javy dostarcza pakiet kryptografii "SUN" z implementacją DSA do tworzenia kluczy wprowadzono interfejsy specyficzne dla kluczy DSA tj. DSAKey, oraz z niego wywodzą się dwa potomne interfejsy DSAPrivateKey i DSAPublicKey. Interfejsy te udostępniają metodę umożliwiającą określenie parametrów p, q i g wykorzystywanych do generacji kluczy w metodzie DSA. Klasą (final class - czyli właściwie jest to zwykły kontener danych) obsługującą klucze jest KeyPair. Klasa ta zawiera jedynie dwie metody umożliwiające uzyskanie klucza prywatnego i publicznego: getPrivate(), getPublic(). W celu wygenerowania kluczy należy oczywiście użyć odpowiedniego algorytmu kodowania oraz algorytmu generatora liczb losowych (a właściwie pseudo-losowych). Dlatego stosuje się obiekty klasy KeyPairGenerator służące do generacji kluczy według przyjętego algorytmu np. DSA. Przykładowy fragment kodu umożliwiający otrzymanie pary kluczy o długości 1024 może wyglądać następująco:

```
KeyPairGenerator gen = KeyPairGenerator.getInstance("DSA");
```

```
gen.initialize(1024);
KeyPair kp = gen.generateKeyPair();
```

Ponieważ obiekt kp klasy kontenera KeyPair umożliwia pobranie klucza prywatnego i publicznego łatwo jest więc uzyskać te klucze celem ich dalszej obróbki. Poniższy program oblicza klucze prywatny i publiczny według algorytmu DSA pakietu SUN, a następnie klucze te są wyświetlane na ekranie:

Przykład 10.4:

```
//Klucze.java:

import java.security.*;

public class Klucze {

 public static void main(String args[]){

  try{
   KeyPairGenerator gen = KeyPairGenerator.getInstance("DSA");
   gen.initialize(512);
   KeyPair kp = gen.generateKeyPair();
   System.out.println("Oto klucz prywatny:"+kp.getPrivate());
   System.out.println("Oto klucz publiczny:"+kp.getPublic());

  } catch(Exception e){
   System.out.println(e);
  }

 }
}// koniec public class Klucze
```

Łatwo zauważyć, że metody println() wyświetlą na ekranie kilka parametrów tj. p, q, g, x oraz y. Wartość x to właśnie klucz prywatny, wartość y to klucz publiczny, natomiast wartości p,q, g to parametry DSA obliczone wcześniej dla pakietu SUN dla ustawień rozmiaru klucza 512, 768 i 1024 (dla DSA możliwe są ustawienia rozmiaru klucza będącego wielokrotnością 64, dla RSA możliwe są ustawienia rozmiaru klucza będącego wielokrotnością 8; w obu przypadkach wartość musi być większa niż 512). Obliczone wcześniej parametry znacznie przyspieszają proces generacji kluczy. Inicjowanie generatora kluczy może dodatkowo opierać się o zestaw bajtów podanych przez użytkownika. Najczęściej użytkownik jest zmuszony do zapisania tekstu, a dopiero później generowany jest zestaw kluczy. Przykład tego typu generacji kluczy ukazuje program Klucze2.java.

Przykład 10.5:

```
//Klucze2.java:

import java.security.*;
import java.io.*;

public class Klucze2 {
```

```
 public static void main(String args[]){
  try{
   KeyPairGenerator gen = KeyPairGenerator.getInstance("DSA");
   SecureRandom los = SecureRandom.getInstance("SHA1PRNG");
   System.out.println("Podaj przykladowy tekst:");
   String seed = (new BufferedReader(new InputStreamReader(System.in))).readLine();
   los.setSeed(seed.getBytes());
   gen.initialize(512, los);
   KeyPair kp = gen.generateKeyPair();
   System.out.println("Oto klucz prywatny:"+kp.getPrivate());
   System.out.println("Oto klucz publiczny:"+kp.getPublic());

  } catch(Exception e){
   System.out.println(e);
  }

 }
}// koniec public class Klucze2
```

W powyższym przykładzie program czeka na wprowadzenie tekstu wejściowego, który jest później wykorzystany w wywołaniu generatora liczb losowych, a docelowo w generatorze kluczy. Klucze są następnie wyświetlone na ekranie. Najczęściej parę kluczy generuje się raz, i są one umieszczane w odpowiedniej bazie danych (keystore) dla potrzeb ich wielokrotnego użycia przy autoryzacji każdej wiadomości.

Jak powiedziano wcześniej proste użycie skrótu nie gwarantuje zabezpieczenia autentyczności wiadomości. Jedną z omówionych wcześniej metod było użycie dodatkowych, tajnych fraz. Innym rozwiązaniem w Javie jest zastosowanie kluczy do kodowania skrótów. Otrzymujemy w ten sposób podpis cyfrowy w Javie. Inicjowanie podpisu cyfrowego w Javie jest więc niczym innym jak wywołaniem skrótu z kodowaniem kluczem prywatnym. Wywołanie obiektu klasy Signature (podpis cyfrowy) odbywa się podobnie jak to omawiano wcześniej, poprzez zastosowanie statycznej metody getInstance() z podaniem typu algorytmu i ewentualnie dostawcy. Przykładowe wywołanie może mieć następujący charakter:

```
Signature podpis = Signature.getInstance("SHA1withDSA");
```

Sama nazwa algorytmu wskazuje na typ operacji: po pierwsze generacja skrótu z wykorzystaniem algorytmu SHA1 oraz wykorzystanie klucza DSA. Obiekt klasy Signature posiada zawsze określony stan. Początkowo jest to stan bez inicjowania - UNINITIALIZED, a następnie możliwy jest jeden z dwóch stanów:
podpisywanie - SIGN oraz weryfikacja VERIFY. W celu stworzenia podpisu należy wprowadzić obiekt podpisu w stan SIGN za pomocą metody initSign(), np.:

```
podpis.initSign(kp.getPrivate());
```

Metoda initSign() wymaga więc jako atrybutu klucza prywatnego koniecznego do generacji podpisu. Kolejne dwa kroki w celu stworzenia podpisu to dostarczenie wiadomości do obiektu podpisu a następnie generacja tego podpisu:

```
podpis.update(wiadomosc);
byte kod_podpisu[] =podpis.sign();
```

W ten sposób wygenerowany został ciąg bajtów określający kod podpisu cyfrowego. W celu zobrazowania całego procesu można posłużyć się następującym przykładem:

Przykład 10.6:

```
//Podpis.java:

import java.security.*;
import java.io.*;

public class Podpis {

 public static void main(String args[]){
  try{
   String tekst = "Czy to sen? Czy to sun? Czy to Java?";
   byte wiadomosc[] = tekst.getBytes();
   MessageDigest md = MessageDigest.getInstance("SHA");
   md.update(wiadomosc);
   byte skrot[] = md.digest();
   KeyPairGenerator gen = KeyPairGenerator.getInstance("DSA");
   SecureRandom los = SecureRandom.getInstance("SHA1PRNG");
   System.out.println("Podaj przykladowy tekst:");
   String seed = (new BufferedReader(new InputStreamReader(System.in))).readLine();
   los.setSeed(seed.getBytes());
   gen.initialize(1024, los);
   KeyPair kp = gen.generateKeyPair();
   System.out.println("Oto klucz prywatny:"+kp.getPrivate());
   System.out.println("Oto klucz publiczny:"+kp.getPublic());
   Signature podpis = Signature.getInstance("SHA1withDSA");
   podpis.initSign(kp.getPrivate());
   podpis.update(wiadomosc);
   byte kod_podpisu[] =podpis.sign();
   System.out.println("Oto podpis cyfrowy: "+kod_podpisu);
   System.out.println("A tak wyglada sam skrot: "+skrot);

  } catch(Exception e){
   System.out.println(e);
  }

 }
}// koniec public class Podpis
```

Powyższy przykład definiuje na początku tekst oraz wiadomość będącą ciągiem bajtów związanych z tekstem. W celu ukazania różnicy pomiędzy skrótem a podpisem, po definicji wiadomości generowany jest skrót. Następnie po generacji kluczy tworzony jest podpis cyfrowy dla danej wiadomości i jest on wyświetlany w porównaniu ze skrótem.
Odbiorca otrzymując wiadomość oraz podpis cyfrowy musi zweryfikować podpis, w celu sprawdzenia autentyczności wiadomości oraz określeniu autora tekstu. W tym celu obiekt podpisu należy wprowadzić w stan weryfikacji - VERIFY - wywołując metodę initVerify() z podaniem klucza publicznego autora, np.:

```
podpis.initVerify(kp.getPublic());
```

Następnie dostarcza się dane do obiektu i wywołuje się metodę weryfikacji verify():

```
podpis.update(wiadomosc);
boolean test = podpis.verify(kod_podpisu);
```

W przypadku poprawności podpisu metoda verify() zwraca true w przeciwnym przypadku false.

W większości przypadków, jak wspomniano wcześniej, klucze nie są generowane za każdym razem przy tworzeniu podpisu, lecz są przechowywane w specjalnej bazie danych. Dla Javy taką bazą danych może być keystore, przechowująca klucze i certyfikaty. W celu dostępu do bazy kluczy stosuje się mechanizmy dostarczane przez klasę KeyStore lub poprzez użycie programów narzędziowych dostarczanych z dystrybucją Javy tj., "keytool", "jarsigner" (podpisywanie archiwuw JAR) oraz "policytool". Baza danych "keystore" to nic innego jak plik o właściwej strukturze. W celu identyfikacji, "keystore" dla pakietu SUN posiada identyfikator JKS. Inicjowanie obiektu klasy KeyStore odbywa się podobnie jak dla pozostałych mechanizmów kryptografii w Javie poprzez przywołanie metody statycznej getInstance(). Następnie należy załadować dane aktualnie przechowywane w bazie do pamięci poprzez wykorzystanie metody load(). Metoda load() specyfikuje strumień wejściowy (skąd czytać) oraz ewentualnie hasło dostępu. Każda pozycja w bazie keystore jest identyfikowana przez unikalną nazwę (ang. alias). W celu uzyskania nazwy można posłużyć się metodą aliases(). W celu określenia czy dana pozycja w bazie jest kluczem czy certyfikatem należy wykorzystać metody typu boolean tj.: isKeyEntry() oraz isCertificateEntry() z podaniem jako argumentu odpowiedniej nazwy pozycji, o którą pytamy. Aby stworzyć nową pozycję w bazie keystore musimy najpierw związać nazwę z daną pozycją poprzez metody: setCertificateEntry() oraz setKeyEntry() podając jako argument odpowiednio certyfikat lub klucz, alias, oraz ewentualnie hasło. W razie potrzeby możemy usunąć pozycję z bazy wykorzystując metodę deleteEntry() z podaniem aliasu pozycji do usunięcia. Wszystkie te operacje są właściwe tylko wówczas, gdy baza keystore jest załadowana do pamięci. W celu nagrania nowo stworzonej lub zaktualizowanej bazy na dysku należy przywołać metodę store() podając odpowiedni strumień wyjścia i hasło. Oprócz przechowywania kluczy w bazie konieczna jest również możliwość pobierania z bazy kluczy celem ich wykorzystania np. dla generacji podpisów. W tym celu mając w pamięci załadowaną bazę należy przywołać metodą getKey() lub getCertificate() odpowiedni klucz lub certyfikat poprzez podanie właściwej nazwy jako argumentu metody.

Na zakończenie omawiania mechanizmów kryptografii w JCA warto zwrócić uwagę na istnienie specjalnych wyjątków poszczególnych mechanizmów. Przykładowo: DigestException, KeyException, SignatureException, itp. Wyjątki te najczęściej występują przy niezgodności inicjowania elementu wedle przyjętego algorytmu danego mechanizmu np. błąd w czasie wywoływania metody sign() (SignatureExcpetion) lub w czasie wykonywania metody digest() (DigestException), itd.

10.4.5 Kodowanie danych

Podsumowując zagadnienia kryptografii w Javie należy powiedzieć, że mechanizmy służące zapewnianiu autentyczności są dostarczane w sposób prosty, tak więc zastosowanie ich w programach nie wymaga specjalistycznej wiedzy programisty. Jakość kodowania jest zależna od zewnętrznych algorytmów, tak więc czysty kod
Javy jest niezwykle uniwersalny. Niestety brak możliwości kodowania danych (JCE - tylko w obrębie USA i Kanady) stanowi poważny problem. Kodowanie danych musi odbywać się więc zewnętrznie (inne aplikacje nie w Javie), a zakodowane dane mogą być następnie dostarczane do aplikacji Javy. Można również wykorzystać opracowane przez inne firmy pakiety do kodowania danych, np.:

Australian Business Access
ABA JCE
http://www.aba.net.au/solutions/crypto/jce.html

Baltimore Technologies
J/CRYPTO
http://www.baltimore.ie/products/jcrypto/index.html

Cryptix
Cryptix
http://www.cryptix.org/products/cryptix31/index.html

DSTC
JCSI
http://security.dstc.edu.au/projects/java/release3.html

Entrust(R) Technologies
Entrust/Toolkit Java Edition
http://www.entrust.com/toolkit/java/index.htm

Forge Research
Forge Security Provider
http://www.forge.com.au/products/crypto/index.html

IAIK
IAIK-JCE
http://jcewww.iaik.tu-graz.ac.at/jce/jce.htm

RSA Data Security, Inc.
Crypto-J
http:// www.rsasecurity.com/products/bsafe/cryptoj.html

Należy jednak zawsze zwracać uwagę na warunki licencjonowania, prawa eksportu kodu, oraz dostarczaną funkcjonalność kodu.