제 9 장

보안

- 클래스 로더
- 바이트 코드 검사
- 보안 관리자와 허가
- 자바 보안 패키지
- 인증
- 자바 인증 프레임웍
- 코드 사이닝
- 보안

자바 기술이 처음 무대에 등장했을 때, 그 흥분은 잘 만들어진 프로그래밍 언어였기 때문이 아니라 인터넷을 통해 넘어 온 애플릿을 안전하게 실행할 수 있기 때문이었다.(애플릿에 관한 상세한 정보는 1권의 10장을 참고하기 바란다)
명백하게, 실행가능한 애플릿을 전달하는 것은 수신자가 그 코드가 자신들의 기계를 엉망으로 만들 수 없다고 확신할 때만 쓸모가 있는 것이다. 이런 이유로, 보안은 자바기술 사용자들과 디자이너들 모두에게 중요한 관심사가 되어 왔다. 이것은 보안이 침입의 반작용 또는 재고로서 수행되어지는 다른 언어나 시스템의 경우와는 다른 의미를 가진다. 보안 기법들은 자바 기술의 총체이다.
자바 기술에 있어서 세가지 기법은 안정성을 보증하는 것을 돕는다.

- 언어 디자인 모습(배열 검사, 적법한 형 변환, 포인터 연산이 아닐 경우, 등등으로 묶는다.)
- 그 코드가 무엇을 할 것인지를 제어하는 접근 제어 기법(파일 접근, 네트워크 접근 등등)
- 코드 사이닝에 의해 코드 작성자들은 자바 프로그래밍 코드를 인증하기 위한 표준 암호 알고리즘을 사용할 수 있다. 따라서 코드 사용자들은 정확하게 누가 그 코드를 만들었는지,그 코드가 사인되어진 이후에 변경되었는지 결정할 수 있다.

자바 가상 머신은 잘못된 포인터들과 타당하지 않은 배열 오프셋 등을 검사한다. 다른 단계들은 자바 가상 머신이 수행하는 과정을 제어한다.

클래스 파일들이 가상 머신으로 로드될 때, 이들의 무결성을 검사한다. 우리는 이 과정이 어떻게 진행되는지 상세하게 보여줄 것이다. 좀 더 중요하게, 우리는 여러분 자신의 클래스 로

더를 만듦으로써 가상 머신이 실행하는 것을 어떻게 제어하는지 보여줄 것이다.
최상의 보안에 대해, 클래스를 로딩하기 위한 기본 기법과 맞춤 클래스 로더 모두
코드가 수행할 수 있는 액션들이 무엇인가를 조정하는 보안 관리자와 함께 작동할 필요가 있다. 여러분은 자신의 보안 관리자 클래스를 어떻게 작성하는지 알게 될 것이다.

마지막으로. 여러분은 코드 사이닝과 사용자 인증을 가능하게 하는, java.security 패키지에 의해 제공되어진 암호 알고리즘을 보게 될 것이다.

언제나처럼, 우리는 응용 프로그래머에게 최대의 관심사였던 이러한 주제들에 초점을 맞추고 있다. 좀 더 깊이 있는 내용을 위해서는, Li Gong의 저서 Inside Java 2 Platform Security[Addison-Wesley 1999]를 추천한다.

클래스 로더

자바 프로그래밍 언어 컴파일러는 소스를 가상 머신이라고 불리는 가설적인 기계언어로 변환한다. 가상 머신 코드는 .class 확장자를 가지는 클래스 파일에 저장되어 진다. 클래스 파일은 하나의 클래스 내에 있는 모든 메소드들에 대한 코드를 포함한다. 이들 클래스 파일들은 가상 머신의 명령어 집합을 대상 머신의 기계어로 번역될 수 있는 프로그램에 의해
번역되어질 필요가 있다.

가상 머신 인터프리터는 단지 프로그램의 실행에 필요한 클래스 파일들만을 로드한다는 것을 주목해라. 예를 들어. MyProgram.class를 실행했다고 가정해보자. 여기에 가상 머신이 수행하는 단계들이 있다.

1.가상 머신은 클래스 파일들을 로드하기 위한 기법을 가지고 있다. 예를 들면, 디스크로 부터 파일들을 읽음으로써 또는 웹으로부터 클래스 파일들을 요청함으로써. 이것은 My Program클래스 파일의 내용들을 로드하기 위해 이 기법을 사용한다.
2. 만약 MyProgram클래스가 데이터 영역이나 다른 클래스 형태의 상위 클래스들을 가진다면 이러한 클래스 파일들 또한 로드되어진다.(주어진 클래스에 의존적인 모든 클래스들을 로딩하는 과정은 클래스를 분해한다고 불리워진다.)
3. 가상 머신은 그리고 MyProgram에서 main메소드(정적이어서 클래스의 인스턴스가 생성될 필요가 없는) 를 수행한다
4.만약 main 메소드 또는 main호출이 추가적인 클래스들을 요청한다면, 이것들은 그 다음에 로드되어진다.

특별한 프로그램이 요구하는 클래스들을 로딩하는 것 외에 ,가상 머신은 시스템 클래스들을 로드해야 한다. 가상 머신은 시스템 클래스를 어떻게 로드해야 하는지 알고 있다. (예를 들

어, rt.jar의 JAR 파일로부터), 가상 머신의 일부이며 주로 C에서 구현되어진 bootstrap클래스 로더를 사용한다. 그러나, 여러분은 응용 클래스들을 로드하기 위해 맞춤 클래스 로더를 제공할 수 있다. 클래스 로더의 가장 일반적인 예는 애플릿 클래스 로더이다. 애플릿 클래스 로더는 어떻게 네트웍을 통하여 클래스 파일들을 로드하는지 그리고 사인된 JAR파일들을 인증하는지 알고 있다.

노트: 부라우져는 각각의 웹 페이지를 위해 애플릿 클래스 로더 클래스의 분리된 인스턴스들을 사용한다. 가상 머신은 각각의 클래스를 위해 사용되어진 클래스 로더 정보를 얻어낸다. 이름은 같지만 다른 클래스 로더 객체들을 가지는 클래스들은 다른 클래스들로 고려되어진다. 이것은 가상 머신이 다른 웹페이지들의 클래스들을 구분할 수 있도록 해준다.

애플릿 클래스 로더와 같은 맞춤 클래스 로더는 클래스 파일들을 로딩하고 위치하게 하는 잘 만들어진 구조를 대신한다. 이것은 여러분이 바이트 코드들을 가상 머신으로 넘기기 전에 특별한 보안 검사를 수행하도록 한다. 예를 들어, 여러분은 "paid for"라고 표기되지 않은 클래스를 로드하는 것을 거절할 수 있는 클래스 로더를 작성할 수 있다. 다음 절에서 어떻게 하는지 보여 준다.

여러분 자신의 클래스 로더 작성

클래스 로더는 추상 클래스 Classloader를 구현한 것이다. 이 클래스에 있는 LoadClass메소드는 어떻게 상위 레벨의 클래스를 로드할 지를 결정한다. 클래스가 클래스 로더를 통하여 한번 로드되어지면, 그것을 참조하는 모든 다른 클래스들은 동일한 클래스 로더를 통하여 로드되어 진다.

여러분 자신의 클래스 로더를 작성하기 위해서, 간단히 다음의 메소드를 오버라이드한다.

```
loadClass(String className, Boolean resolve)
```

여러분은 이 메소드를 구현해야 한다.

1. 이 클래스 로더가 이미 해당 클래스를 로드했는가를 검사하라. 이 를 위해서 , 여러분의 클래스 로더는 그것이 이전에 로드했던 클래스들의 기록을 유지할 필요가 있다.
2. 만약 그것이 새로운 클래스라면, 여러분은 그것이 시스템 클래스인 가를 검사할 필요가 있다. 만약 그렇지 않다면, 로컬 파일 시스템이나 어떤 다른 소스로부터 그 클래스를 위한 바이트 코드를 로드하라.

3. 바이트 코드를 가상 머신에 전해주기 위한 ClassLoader 상위 클래스의 defineClass 메소드를 호출하라.

만약 resolve플래그가 설정되면, 여러분은 ClassLoader 상위 클래스의 resolveClass 메소드를 호출해야 한다. 여러분의 클래스 로더는 이 클래스가 참조하는 다른 클래스들을 로드하기 위해 다시 호출될 것이다. (만약 가상 머신이 단순히 클래스가 존재하는지를 알아내기를 시도한다면, 여러분의 loadClass 메소드는 resolve를 false로 설정되어 호출될 것이다. 그러나 모든 클래스는 여러분이 인스턴스를 생성하기 전에 또는 메소드를 호출하기 전에 충분히 결정되어져야 한다.)

일반적으로, 클래스 로더는 이미 로드된 클래스들에 대한 참조들을 저장하기 위해 해쉬 테이블과 같은 맵을 사용한다. 다음의 코드 예제는 전형적인 클래스 로더의 loadClass 메소드의 구조를 보여준다.

```
public class TypicalClassLoader extends ClassLoader
{  protected synchronized class loadClass(String name, boolean resolve)
        throws ClassNotFoundException
   {  //check if class already loaded
      Class c1 = (Class)classes.get(name);

      if (c1 == null)  //new class
        {  try
           {  // check if system class
              return fundSystemClass(name);
           }
           catch (ClassNotFoundException e)  {}
           catch(NoClassDefFoundError e)  {}

           //  load class bytes--details depend on class loader

           byte[]  classBytes = loadClassBytes(name);
           if  (classBytes == null)
               throws new ClassNotFoundException(name);

           c1 = defineClass(name, classBytes,  0,  classBytes.length);
           if (c1 == null) throws new ClassNotFoundException(name);
```

```
            classes.put(name,   c1);  // remember class
        }

        if (resolve)   resolveClass(c1);

        return c1;
      }

    private byte[] loadClassBytes(String name)
    {   .   .   .
    }
    private   Map   classes = new HashMap ();
}
```

예제 9-1의 프로그램에서, 우리는 암호화된 클래스 파일들을 로드하는 클래스 로더를 구현한다. 그 프로그램은 사용자에게 로드하기 위한 첫 클래스(그것은 main을 포함하는 클래스)의 이름과 복호화 키를 요구한다. 이때 그것은 특정한 클래스를 로드하기 위해 특별한 클래스 로더를 사용한다. 그리고 main 메소드를 호출한다. 이 클래스 로더는 특정한 클래스와 그것에 의해 참조되어지는 모든 비 시스템 클래스들을 해독한다.
끝으로, 그 프로그램은 로드된 클래스의 main 메소드를 호출한다.(그림 9-1을 참조)

그림 9-1: 클래스 로더 프로그램

단순성을 위해 우리는 암호화 영역의 2000년간의 발전 과정을 무시하고, 클래스 파일들을 암호화하기 위해 고전의 시저(Caesar) 암호화를 사용한다. – 그래서 우리는 이 책을 안심하고 수출할 수 있다.

노트: David kahn의 뛰어난 저서 The Code Breakers,(Macmillan,NY,1967, p. 84) Caesar 암호문을 위한 역사적인 소스로서 Suetonius를 언급한다. Caesar는 24자의 로만 알파벳을 3개의 문자로 바꾸었다. 이 책이 집필되어질 때, 미국 정부는 강력한 암호화의 수출에 제한을 가하고 있다.

따라서, 우리는 그것이 추측상 수출에 적법일 정도로 아주 약하기 때문에 우리의 예제를 위해 Caesar의 방법을 사용한다.

우리의 Caesar암호문 버전은 1에서 255사이의 수를 키로서 가진다. 해독을 위해, 간단히 매 바이트마다 키 값을 더하고 256 모듈러 연산을 행한다. 예제 9-2의 Caesar.java 프로그램은 암호화를 수행한다.

보통의 클래스 로더와 혼동하지 않기 위해, 우리는 암호화된 클래스 파일들을 위해 .Caesar라는 다른 확장자를 사용한다.

해독을 위해, 클래스 로더는 간단하게 각 바이트마다 키 값을 뺀다. 이 책을 위한 CD-ROM에서, 여러분은 전통적인 방식인 3의 키 값으로 암호화된 4개의 클래스 파일들을 발견할 수 있을 것이다. 여러분은 일반적인 바이트 코드 인터프리터를 통해 이러한 클래스들을 로드할 수 없다. 그러나 여러분은 우리의 ClassLoaderTest 프로그램에서 정의된 맞춤 클래스 로더를 사용함으로써 암호화된 프로그램을 수행할 수 있다.

클래스 파일들을 암호화하는 것은 많은 실질적인 사용법이 있다.(만약, 물론 여러분은 Caesar암호문보다 더 강력한 암호문을 사용한다.) 해독 키 없이 클래스 파일들은 쓸모가 없다. 그들은 표준의 바이트코드 인터프리터에 의해 실행되지도 않고 쉽게 분해되지도 않는다.

이것은 여러분이 클래스 사용자를 인증하기 위해 또는 어떤 프로그램이 그것의 실행이 허락되어지기 전에 값이 지불되어졌는지 보증하기 위해 맞춤 클래스 로더를 사용할 수 있다는 것을 의미한다.

물론, 암호화는 맞춤 클래스 로더의 단지 하나의 응용일 뿐이다. 여러분은 다른 문제들을 풀기 위해 클래스 로더들의 다른 타입을 사용할 수 있다. 예를 들어, 여러분의 코드가 인증된 사용자들에 의해서만 실행되도록 할 것인지 제어하기 위한 것과 같은 문제.

예제 9-1: ClassLoaderTest.java

```
import java.util.*;
import java.io.*;
import java.lang.reflect.*;
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;

public class ClassLoaderTest
{   public static void main(String[] args)
   {  Frame f = new ClassLoaderFrame();
      f.show();
   }
}

class ClassLoaderFrame extends JFrame
{   public ClassLoaderFrame()
   {   setTitle("ClassLoaderTest");
      setSize(300, 200);
      addWindowListener(new WindowAdapter()
         {   public void windowClosing(WindowEvent e)
            {   System.exit(0);
            }
         } );
      getContentPane().setLayout(new GridBagLayout());
      GridBagConstraints gbc = new GridBagConstraints();
      gbc.weightx = 0;
      gbc.weighty = 100;
      gbc.fill = GridBagConstraints.NONE;
      gbc.anchor = GridBagConstraints.EAST;
      add(new JLabel("Class"), gbc, 0, 0, 1, 1);
      add(new JLabel("Key"), gbc, 0, 1, 1, 1);
      gbc.weightx = 100;
      gbc.fill = GridBagConstraints.HORIZONTAL;
      gbc.anchor = GridBagConstraints.WEST;
      add(nameField, gbc, 1, 0, 1, 1);
      add(keyField, gbc, 1, 1, 1, 1);
      gbc.fill = GridBagConstraints.NONE;
      gbc.anchor = GridBagConstraints.CENTER;
      JButton loadButton = new JButton("Load");
      add(loadButton, gbc, 0, 2, 2, 1);
      loadButton.addActionListener(
```

```
            new ActionListener()
            {   public void actionPerformed(ActionEvent event)
                {   runClass(nameField.getText(), keyField.getText());
                }
            });
    }

    public void add(Component c, GridBagConstraints gbc,
        int x, int y, int w, int h)
    {   gbc.gridx = x;
        gbc.gridy = y;
        gbc.gridwidth = w;
        gbc.gridheight = h;
        getContentPane().add(c, gbc);
    }

    public void runClass(String name, String key)
    {   try
        {   ClassLoader loader
                = new CryptoClassLoader(Integer.parseInt(key));
            Class c = loader.loadClass(name);
            String[] args = new String[] {};

            Method m = c.getMethod("main",
                new Class[] { args.getClass() });
            m.invoke(null, new Object[] { args });
        }
        catch (Throwable e)
        {   JOptionPane.showMessageDialog(this, e);
        }
    }

    private JTextField keyField = new JTextField("3", 4);
    private JTextField nameField = new JTextField(30);
}
```

```
class CryptoClassLoader extends ClassLoader
{   public CryptoClassLoader(int k)
    {   key = k;
    }

    protected synchronized Class loadClass(String name,
        boolean resolve) throws ClassNotFoundException
    {   // check if class already loaded
       Class cl = (Class)classes.get(name);

        if (cl == null) // new class
        {   try
            {   // check if system class
                return findSystemClass(name);
            }
            catch (ClassNotFoundException e) {}
            catch (NoClassDefFoundError e) {}

            // load class bytes--details depend on class loader

            byte[] classBytes = loadClassBytes(name);
            if (classBytes == null)
                throw new ClassNotFoundException(name);

          cl = defineClass(name, classBytes,
                0, classBytes.length);
            if (cl == null)
                throw new ClassNotFoundException(name);

            classes.put(name, cl); // remember class
        }

        if (resolve) resolveClass(cl);

        return cl;
    }
```

```
    private byte[] loadClassBytes(String name)
  {  String cname = name.replace('.', '/') + ".caesar";
     FileInputStream in = null;
      try
     {   in = new FileInputStream(cname);
         ByteArrayOutputStream buffer
             = new ByteArrayOutputStream();
         int ch;
         while ((ch = in.read()) != -1)
         {   byte b = (byte)(ch - key);
             buffer.write(b);
         }
         in.close();
         return buffer.toByteArray();
      }
      catch (IOException e)
      {   if (in != null)
          {   try { in.close(); } catch (IOException e2) { }
          }
          return null;
      }
   }

   private Map classes = new HashMap();
   private int key;
}
```

예제 9-2: Caesar.java

```
import java.io.*;

public class Caesar
{   public static void main(String[] args)
    {   if (args.length != 3)
        {   System.out.println("USAGE: java Caesar in out key");
```

```
            return;
        }

        try
        {   FileInputStream in = new FileInputStream(args[0]);
            FileOutputStream out = new FileOutputStream(args[1]);
            int key = Integer.parseInt(args[2]);
            int ch;
            while ((ch = in.read()) != -1)
            {   byte c = (byte)(ch + key);
                out.write(c);
            }
            in.close();
            out.close();
        }
        catch(IOException e)
        {   System.out.println("Error: " + e);
        }
    }
}
```

java.lang.ClassLoader

- Class definClass(String name, byte data[], int offset, int length)

  가상 머신에 새로운 클래스를 추가한다.

  전달인자 : name **클래스의 이름. 패키지 이름의 분리자로서 .을 사용하라. 그리고** class **접미사를 붙이지 말 것.**

  Data **클래스의 바이트 코드들을 확보하기 위한 배열**

  Offset **배열내의 바이트 코드들의 시작**

  Length **배열내의 바이트 코드들의 길이**

- Void loadClass(String name, Boolean resolve)

  ClassLoader**를 확장하는 클래스에 의해 구현되어 진다. 그것은 클래스를 위한 바이트 코드를 얻어,** defineClass**를 호출한다. 만약** resolve **플래그가 참 값이면,** resoveClass **이다. 해당 클래스 로더는 이전에 로드된 클래스들이 다시 로드되어지지 않는다는 것을 확인하기 위해 캐쉬를 구현해야 한다.**

  전달인자 : name **클래스의 이름. 패키지 이름 분리자로서** .class **접미사를 붙이**

지 말고 .을 사용할 것

resolve 클래스가 로드되어진 이후에 resolveClass 메소드가 호출되어질 필요가 있으면 true값

- Void resolveClass(Class c)

  Resolve플래그가 true값이면 loadClass에 의해 호출되어 져야 한다. 그것은 클래스가 직접 또는 간접적으로 언급하는 모든 클래스들이 충분히 가상 머신에 알려질 때까지 종속적인 클래스들을 계속적으로 로딩한다. 클래스가 한번 분해되어지면, 가상 머신은 클래스 객체를 생성하거나 클래스 메소드들을 호출할 수 있다.

  전달인자 : c 분해되어진 클래스

- Class findSystemClass(String name)

  특정한 이름을 가지는 시스템 클래스를 찾고 필요하면 그것을 로드한다. 시스템 클래스는 CLASSPATH 환경 변수의 값을 사용하는, 플랫폼 종속적인 방식인 로컬 파일 시스템으로부터 로드되어진 클래스이다. 시스템 클래스들은 클래스 로드를 가지지 않는다.

  전달인자 : name 클래스의 이름. 패키지이름 분리자로서 .class접미사를 붙이지 말고 .을 사용할 것

바이트 코드 검사

클래스 로더가 가상 머신에 새롭게 로드되어진 자바 플랫폼 클래스의 바이트 코드들을 낼 때 ,이러한 바이트 코드들은 처음에 검증기에 의해 검증되어진다. 검증기는 명령어들이 명백하게 손상을 입히는 액션들을 수행할 수 없도록 검사한다. 시스템 클래스들을 제외한 모든 클래스들은 검증되어 진다. 그러나 여러분은 -noverify 옵션을 사용하여 검증기를 비활성화 할 수 있다.

예를 들어,

```
java – noverify Hello
```

여기에 검증기가 수행하는 일부 검사들이 있다.

- 변수들은 그들이 사용되어지기 전에 초기화되어지는가.
- 메소드가 객체 참조자들의 타입에 맞추어 호출하는가.
- 개별적인 데이터와 메소드들을 접근하는 규칙들이 위반되어지지 않았는가.
- 지역 변수접근이 런타임 스택내에 들어가는가.
- 런타임 스택이 오버플로우되지는 않는가.

만약 이러한 검사들 중 어떤 것이 실패한다면, 그 클래스는 손상된 것으로 간주되어지고 로드되지 않을 것이다.

노트: 만약 여러분이 Godel 이론에 익숙하다면, 여러분은 어떻게 검증기가 클래스들이 형불일치, 비초기화된 변수들,그리고 스택 오버플로우로부터 자유롭다는 것을 입증할 수 있는지 궁금할지도 모른다. Godel 이론은 입력이 프로그램 파일들이고, 출력이 입력프로그램의 특별한 성질(스택 오버플로우로부터 자유로운지와 같은)을 가지는지를 나타내는 불리언 값인 알고리즘을 설계하기가 불가능하다는 것을 말하고 있다. 이것은 선 마이크로시스템즈에서의 공적인 관계부분과 로직 법칙사이에 모순이 아니냐?..그렇지 않다.– 사실, 검증기는 Godel 관점에서의 결정 알고리즘이 아니다. 만약 검증기가 프로그램을 받아들인다면, 그것은 정말 안전하다. 그러나 거기에는 그들이 실질적으로 안전할지라도 검증기에 의해 거절되는 많은 프로그램들이 있을지도 모른다.

이 엄격한 검증은 중요한 보안 사항이다. 비초기화된 변수들과 같은 우연적인 실수들은 그들이 잡혀지지 안는다면 쉽게 파괴를 초래할 수 있다.
더욱 중요하게, 인터넷의 활성화 시대에서, 여러분은 고의로 악의의 효과를 만들어내는 악의가 있는 프로그래머들로부터 보호되어져야 한다. 예를 들어, 런타임 스택의 값들을 변경함으로써 또는 시스템 객체의 사적인 데이터 영역에 쓰기를 함으로써, 프로그램은 부라우져의 보안 시스템을 통과하여 파괴할 수 있다.

그러나 여러분은 거기에 왜 이러한 모든 것을 검사하는 특별한 검증기가 있는지 궁금해할지도 모른다. 결국, 컴파일러는 여러분이 초기화되지 않은 변수를 사용하거나 또는 개별적인 데이터 영역이 다른 클래스로부터 접근되어지는 클래스 파일들을 만드는 것을 허락하지 않을 것이다. 실제로, 자바 프로그래밍 언어를 위한 컴파일러에 의해 생겨난 클래스 파일은 항상 검증기를 통과한다. 그러나 클래스 파일들에서 사용되어지는 바이트 형태는 문서화가 잘 되어 있다. 그리고 어셈블리 프로그래밍이나 자바 가상머신을 위한 타당하지만 불안정한 명령어들을 포함하는 클래스를 손수 만들어내기 위한 hex 편집에 경험이 있는 일부 사람들에게는 이것은 쉬운 문제이다. 다시 한번, 검증기는 항상 컴파일러에 의해 생겨난 클래스 파일들만을 검사하기 위한 것이 아니라, 악의로 바뀌어진 클래스 파일들로부터 지키기 위한 것임을 명심하라.

여기에 어떻게 변경된 클래스 파일을 만들 수 있는지 예가 있다. 우리는 예제 9-3의 VerifierTest.java프로그램으로 시작한다. 이것은 메소드를 호출하고 그 메소드의 결과를 디스플레이하는 간단한 프로그램이다. 이 프로그램은 console프로그램이나 애플릿으로

둘 다 실행될 수 있다. 그것의 fun 메소드는 단순히 1과 2를 더하는 계산을 한다.

```
static int fun()
{   int m;
     int n;
    m = 1;
    n = 2;
     int r = m + n;
     return r;
}
```

실험으로서, 이 프로그램을 다음처럼 변경한 후 컴파일 하도록 하라:

```
static int fun()
{   int m = 1;
    int n;
    m = 1;
     m = 2;
     int r = m + n;
     return r;
 }
```

이 경우에 n은 초기화되지 않는다. 그것은 어떤 임의의 값을 가지게 될 것이다. 물론 컴파일러는 문제를 감지하고 그 프로그램을 컴파일 하는 것을 거절한다. 불량 클래스 파일을 만들기 위해서, 우리는 조금 더 힘든 작업을 해야 한다. 처음에 어떻게 컴파일러가 fun메소드를 번역하는지 알아내기 위해 javap프로그램을 실행하라. 그 명령은

```
javap –c VerifierTest
```

기억하기 쉬운 형태의 클래스 파일로 바이트 코드들을 보여준다.

```
Method int fun()
      0 iconst_1
      1 istore_0
      2 iconst_2
      3 istore_1
      4 iload_0
      5 iload_1
      6 iadd
```

7 istore_2

8 iload_2

9 ireturn

우리는 명령어 3을 istore_1에서 istore_o 로 바꾸기 위해 hex 편집기를 사용할 것이다. 지역변수 0 (m)은 두번 초기화되어진다. 그리고 지역변수1(n)은 전혀 초기화되지 않는다. 우리는 이러한 명령어들을 위한 16진수 값을 알 필요가 있다. 이러한 값들은 Tim Lindholm과 Frank Yellin[Addison-Wesly,1997]의 저서 The Java Virtual Machine으로부터 쉽게 이용할 수 있다.

0 iconst_1 04

1 istore_0 3B

2 iconst_2 05

3 istore_1 3C

4 iload_0 1A

5 iload_1 1B

6 iadd 60

7 istore_2 3D

8 iload_2 1C

9 ireturn AC

우리는 수정을 하기 위해 우리의 즐겨쓰는 hex편집기로서 Hex Workshop(이 책을 위한 CD-ROM에 포함된)을 사용할 것이다. 그림 9-2에서 여러분들은 fun 메소드의 바이트 코드가 밝게 표시되어져, Hex WorkShop에 로드되어진 VerifierTest.class 클래스 파일을 보고 있다. 우리는 간단하게 3C를 3B로 바꾸고 클래스 파일을 저장한다.(만약 여러분이 스스로 hex편집기를 실행하고 싶지 않다면, 여러분은 CD-ROM에 편집되어진 VerifierTest.class를 발견할 수 있을 것이다. 단지 VerifierTest 소스 파일을 다시 컴파일하지 않는 것을 확실히 하도록 하라)

VerifierTest 프로그램을 실행하려고 시도해보라. 그러면 여러분은 에러 메세지를 얻게 될 것이다.

Exception in thread “main” java.lang.VerifyError : (class :VerifierTest, Method : fun signature : () I) Accessing value uninitalized register 1

이것은 좋다- 가상 머신이 우리의 수정사항을 감지했다는 것이다.

지금 –noverify 옵션으로 프로그램을 수행해보자. fun 메소드는 겉보기에는 임의의 값을 반환한다. 이것은 실질적으로 초기화 되어지지 않았던 변수 n에 저장되어 졌던 값에 2를 더한

것이다. 여기에 특별한 출력결과가 있다.

1 + 2 = 15102330

부라우져들이 검증을 어떻게 다루는지 알기 위해 우리는 응용 프로그램이나 애플릿 어느 것으로도 실행하기 위해 이 프로그램을 작성했다. 다음과 같은 파일 URL을 사용하여 부라우져로 애플릿을 로드한다.

file|///C:/CoreJavaBook/v2ch9/VerifierTest/VerifierTest.html

그리고 나서 여러분은 검증에 실패했음을 지적하기 위해 출력되는 에러 메세지를 볼 수 있다.(그림 9-3을 참고)

예제 9-3 : VerifierTest.java

```
import java.awt.*;
import java.applet.*;

public class VerifierTest extends Applet
{   public static void main(String[] args)
    {   System.out.println("1 + 2 == " + fun());
    }

    static int fun()
    {   int m;
        int n;
       m = 1;
       n = 2;
        // used hex editor to change to "m = 2" in class file
        int r = m + n;
        return r;
    }

    public void paint(Graphics g)
    {   g.drawString("1 + 2 == " + fun(), 20, 20);
    }
```

}

보안 관리자와 허가

클래스가 한번 클래스 로더나 기본 클래스 로딩 방법에 의해 로드되어지고 검증에 의해 검사되어 졌으면 자바 플랫폼의 3번째 보안 과정(보안 관리자)이 나오게 된다. 보안 관리자는 특정한 동작이 허가되었는지 제어하는 클래스이다.

보안 관리자에 의해 검사되어지는 동작들은 다음을 포함한다.

- 현재 쓰레드가 새로운 클래스를 생성할 수 있는가.
- 현재 쓰레드가 하위 프로세스를 생성할 수 있는가.
- 현재 쓰레드가 가상 머신을 일시 정지시킬 수 있는가.
- 현재 쓰레드가 동적 링크 라이브러리를 로드할 수 있는가.
- 클래스가 다른 클래스의 멤버를 제어할 수 있는가.
- 현재 쓰레드가 특정한 패키지를 제어할 수 있는가.
- 현재 쓰레드가 특정 패키지 내에 클래스들을 정의할 수 있는가.
- 현재 쓰레드가 시스템 프로퍼티를 제어하거나 수정할 수 있는가.
- 현재 쓰레드가 특정 파일로부터 읽고, 쓸 수 있는가.
- 현재 쓰레드가 특정한 파일을 삭제할 수 있는가.
- 현재 쓰레드가 특정 호스트와 포트 번호로부터 소켓 연결을 받아들일 수 있는가.
- 현재 쓰레드가 특정 호스트와 포트 번호에 소켓 연결을 열수 있는가.
- 현재 쓰레드가 특정 로컬 포트 번호를 요청하는 연결을 기다릴 수 있는가.
- 현재 쓰레드가 IP멀티캐스트를 사용할 수 있는가.
- 현재 쓰레드가 주어진 쓰레드나 쓰레드 그룹의 stop, suspend, resume, destroy, setPripority/ setMaxProperity,setName또는 setDaemon메소드를 부를 수 있는가.
- 현재 쓰레드가 소켓이나 스트림 핸들러 팩토리를 설정할 수 있는가.
- 클래스가 프린트 작업을 시작할 수 있는가.
- 클래스가 시스템 클립보드를 제어할 수 있는가.
- 클래스가 AWT이벤트 큐를 제어할 수 있는가.
- 현재 쓰레드가 상위 레벨 윈도우를 만들어 내는 것을 신뢰받을 수 있는가.

보안 관리자가 설치되지 않은 자바 응용 프로그램을 실행할 때 기본 동작은, 이러한 모든 동작은 허가되어 진다. 한편으로 애플릿 뷰어는 즉시 아주 제한된 보안 관리자(AppletSecurity라고 불리는)를 설치한다.

예를 들어, 애플릿은 가상 머신을 빠져나가는 것이 허락되지 않는다. 만약 그들이 exit 메소드를 호출하려고 한다면, 그때 보안 예외상황이 발생하게 된다. 여기에 상세하게 어떤 일이

일어나는지 나와 있다. Runtime클래스의 exit 메소드는 보안 관리자의 checkExit메소드를 호출한다. 여기 exit메소드의 완전한 코드가 있다.

```
public void exit(int status)
{ SecurityManager security = System.getSecurityManager();
 if ( security != null )
        security.checkExit( status );
  exitInternal(status);
}
```

부라우져나 개별적인 애플릿으로 부터 종료 요청이 오면 보안 관리자는 지금 검사를 한다. 보안 관리자가 종료요청에 동의하면, checkExit메소드가 간단히 반환되고 정상적인 프로세싱이 계속된다. 그러나 보안관리자가 그 요청을 승인하기를 원하지 않는다면, checkExit메소드는 SecurityException을 던진다.

exit 메소드는 아무런 예외상황도 발생하지 않을 때만 계속된다. 그때 그것이 private native exitInternal 메소드를 호출하게 되었을 때 실질적으로 가상 머신을 종료한다. 가상 머신을 종료할 수 있는 다른 방법은 없다. 그리고 exitInternal 메소드가 private이기 때문에 그것은 다른 어떤 클래스로부터 호출되어질 수 없다. 따라서 가상 머신을 종료하기를 시도하는 코드는 exit 메소드와 보안 예외상황을 유발시키지 않고 checkExit 보안 검사를 통과해야 한다.

명백하게 보안정책의 통합은 사려 깊은 코딩에 달려 있다. 표준 라이브러리 내에 있는 시스템 서비스 제공자들은 항상 어떤 민감한 동작을 시도하기 전에 보안 관리자를 염두하는 것에 주의해야 한다.

여러분이 자바 응용 프로그램을 수행할 때, 기본으로 보안 관리자가 동작하지 않는다. 여러분의 프로그램은 System클래스 내에 있는 static setSecurityManager 메소드를 호출함으로써 특정한 보안 관리자를 설치할 수 있다. 여러분의 프로그램이 한번 보안 관리자를 설치하면, 만약 처음 보안 관리자가 대체되어지는 것을 동의하면 두 번째 보안 관리자를 설치하기 위한 시도는 성공한다. 이것은 아주 기본적인 것이다. 반면에, 불량 애플릿은 자신의 보안 관리자를 설치할 수 있다. 따라서 그것이 다수의 클래스 로더를 가질 수 있는 동안, 자바 프로그래밍 언어 내에 있는 프로그램은 단지 하나의 보안 관리자에 의해서 관리되어질 수 있다.
모든 클래스들에 대해 동일한 접근을 승인할 것인지 또는 무엇을 할 것인지 결정하기 전에 클래스들의 출처를 고려할 것인가를 결정하는 것은 구현자에게 달려 있다.

자바 2 플랫폼의 기본 보안 관리자는 프로그래머와 시스템 관리자들에게 개별적인 보안 허가를 통해 양질의 제어를 허락한다. 우리는 다음 장에서 이러한 모습들을 설명할 것이다.
처음에 우리는 여러분에게 자바 2 플랫폼 보안 모델의 개요를 줄 것이다. 그리고 나서 우리는 여러분에게 policy파일을 가지고 어떻게 허가를 제어할 수 있는지 보여줄 것이다. 마지막으로 여러분은 여러분 자신의 허가 형태를 어떻게 정의하는지 그리고 여러분이 어떻게 기본 보안 관리자 클래스를 확장할 수 있는지 보여줄 것이다.

자바 2 플랫폼 보안

JDK1.0은 로컬 클래스들은 충분한 승인을 가지고, 원거리의 클래스들은 샌드 박스(sandbox, 한정된 공간)에 의해 한정되어지는 매우 간단한 보안 모델을 가졌었다: 애플릿 보안 관리자는 로컬 자원에 접근하기 위한 모든 접근을 거절했다. JDK1.1은 신뢰된 것에 의해 사인되어진 원거리 코드는 로컬 클래스들처럼 동일한 허가가 승인되어지도록 약간의 수정을 수행했다. 그러나 JDK 두 버전 모두 "전부 또는 전무(All or Nothing)"의 접근을 제공했다. 프로그래머들은 모두 접근할 수 있거나 또는 샌드 박스(Sandbox) 내에서 활동해야 했다.

자바 2플랫폼은 보다 더 융통성이 있는 기법이다. Security policy는 코드 소스를 permission set에 매핑시킨다.

그림 9-4: 보안 정책

코드 소스는 코드 위치(예를 들어, 코드 베이스 URL 또는 jar파일)와 인증의 두 가지 프로퍼티를 가진다. 여러분은 이 장에서 어떻게 코드가 신뢰된 부분들에 의해 인증되어질 수 있는지 나중에 볼 수 있을 것이다.

허가는 보안 관리자에 의해 검사되어지는 어떤 프로퍼티이다. JDK1.2구현은 특별한 허가를 상세하게 요약한 각각의 많은 허가 클래스들을 지원한다. 예를 들어, 다음의 FilePermission 클래스 인스턴스는 /tmp 디렉토리에 어떤 파일을 읽고 쓸 수 있다는 것을 나타낸다.

```
FilePermission p = new FilePermission("/tmp/*", "read,write");
```

더욱 중요하게, JDK1.2에서의 기본 Policy클래스의 구현은 permission 파일로부터 허가권을 읽는다. 허가 파일내에 동일한 읽기 권한이 다음과 같이 표현되어진다.

```
Permission java.io.FilePermission "/tmp/*", "read, write";
```

우리는 다음 장에서 허가 파일들에 대해 얘기한다. 그림 9-5는 JDK1.2에서 허가 클래스들의 계층구조를 보여준다.

그림 9-5: JDK1.2에서 허가(Permission) 계층도

이전 장에서, 여러분은 SecurityManager 클래스가 checkExit와 같은 보안 검사하는 메소드들을 가지고 있는 것을 알았다. 이러한 메소드들은 단지 프로그래머의 편의와 퇴보된 방향의 적합성을 위해 존재한다. 그들 모두는 다음의 두 메소드들 중 하나를 호출한다.

```
void checkPermission(Permission p)
void checkPermission(Permission p,  Object context)
```

하나의 쓰레드가 보안 검사를 수행한다면 다른 쓰레드를 위해서 두번째 방법이 사용되어진다. Context는 검사 시간에 모든 콜 스택을 캡슐화한다. (컨텍스트 객체들을 어떻게 생성하고 사용하는지 알기 위해서 Gong의 저서를 보도록 하자)

예를 들어, 여기에 checkExit 메소드를 위한 소스 코드가 있다.

```
public void checkExit()
{    checkPermission (new RuntimePermission("exitVM"));
}
```

각각의 보안 관리자는 checkPermission 메소드의 자기 구현을 제공하는 것이 자유롭다. 그러나 JDK는 보안 검사를 어떻게 수행할 지의 '표준 모델'을 제공한다. 이 장의 나머지에서 우리는 이 표준 모델을 설명할 것이다. 표준 모델은 다음의 두개의 클래스에 의존한다.

```
java.security.SecureClassLoader
java.lang.SecurityManager
```

이들은 모든 실제적인 설정들(애플릿과 원격 메소드 호출과 같은) 에서 사용되어 지는 클래스 로더나 보안 관리자의 상위 클래스들이다. 원칙적으로. 여러분은 여러분 자신의 클래스 로더나 보안 관리자를 설치할 수 있다. 그러나, 그것은 프로그래머들이 시도하기를 꺼리는

복잡한 일이다. 표준 클래스들을 상속하는 것이 더욱 일반적이다.

표준 모델은 소스 코드들을 허가에 매핑하기 위해 Policy 객체에 의존한다. 어떤 주어진 시간에서는 단 하나의 Policy객체가 효과적일 수 있다. Policy 클래스의 전역 함수 getPolicy는 현재의 정책을 얻는다.

```
Policy currentPolicy = Policy.getPo;icy();
```

Policy클래스의 중요한 메소드는 실제적인 코드 소스를 위해 승인 객체를 반환하는 getPermissions 메소드이다.

```
PermissionCollection permissions
    = currentPolicy.getPermissions(codeBase);
```

각각의 클래스는 코드소스와 클래스의 승인 집합 모두를 캡슐화하는 객체인 protection domain을 가진다. Class 클래스의 GetProtectionDomain 메소드는 도메인을 반환한다.

```
ProtectionDomain domain
    = anObject.getClass().getProtectionDomain();
```

ProtectionDomain 메소드의 getCodeSource와 getPermission 메소드들은 코드 소스와 승인 집합을 반환한다.

표준 모델에서, 승인 집합은 완전히 코드 소스에 종속적이다. 방어 도메인은 SecureClassLoader가 클래스를 로드할 때 설정된다. SecureClassLoader는 코드 소스에 맞는 승인을 위한 현재의 정책을 묻는다. 그때 그것은 주어진 소스 코드와 승인을 가지고 ProtectionDomain객체를 생성한다. 마침내, 그것은 객체에게 defineClass메소드를 넘겨준다. 그림 9-6은 보안 객체들 사이의 관계를 보여준다.

그림 9-6: 보안 클래스들 사이의 관계

SecurityManager가 승인을 검사할 필요가 있을 때, 그것은 콜 스택에 있는 현재의 모든 메소드들의 클래스를 찾는다. 그때 그것은 모든 클래스들의 방어 도메인들을 얻고 그리고 그것의 승인 집합이 현재 검사되어지는 동작을 허가하면 각각의 방어 도메인을 요청한다. 만약 모든 도메인이 동의하면, 그때 검사가 통과된다. 그렇지 않으면 SecurityException이 던져

진다.

왜 콜 스택에 있는 모든 메소드들이 실제적인 동작을 허락할 필요가 있을까? 예를 통해 실행해 보자. 애플릿의 init 메소드가 파일을 열기를 원한다고 가정해 보자. 그것은

```
Reader in = new FileReader(name);
```

을 호출할 것이다. FileReader 생성자는 마침내 FilePermisssion(name, "read")객체를 가지는 checkPermission을 호출하는 보안 관리자의 checkRead 메소드를 호출하는 FileInputStream 생성자를 호출한다.

표 9-1은 콜 스택을 보여준다.

표 9-1 : 허가 검사동안의 호출 스택(Call stack during permission checking)

FileInputStream 과 SecurityManager클래스는 codeSource가 null이고 그것의 퍼미션이 모든 동작을 허락하는 AllPermission클래스의 인스턴스로 구성된 시스템 클래스 들이다.
명백하게, 그들의 퍼미션들은 혼자서 검사의 결과를 결정할 수 없다. 여러분이 볼 수 있듯이, checkPermission메소드는 애플릿 클래스의 엄격한 퍼미션들을 고려해야 한다. 모든 콜 스택을 검사함으로써 보안 구조는 하나의 클래스가 그것의 이익을 위해 민감한 동작을 수행하는 다른 클래스를 요청할 수 없다는 것을 보증한다.

노트 : 퍼미션 검사의 이 간단한 논의는 여러분에게 간단한 개념을 보여준다. 그러나 거기에는 여기에서 우리가 생략한 많은 기술적인 상세한 것들이 있다. 상세한 것들과 그리고 더욱 많은 정보를 위해 우리는 Li Gong의 저서를 읽도록 여러분에게 권한다. 자바 플랫폼 보안 모델의 더욱 결정적인 모습을 위해서 Gary McGraw와 Ed Felten(Josh Wiley & Sons 1999)의 저서 Securing Java를 보라. 여러분은 http://www.securingjava.com에서 그 저서를 온라인 버전으로 볼 수 있다.

---

java.lang.SecurityManager

- void checkPermission(Permission p)
- void checkPermission(Permission p, Object context)

  현재 보안 정책이 주어진 퍼미션을 허가하는지 검사한다. 두번째 메소드는 콜 스택을 캡슐화하는 객체를 받는다. 메소드는 하나의 쓰레드가 다른 쓰레드에게 그것의 이익을 위해서 퍼미션 검사를 수행하기 위해 다른 쓰레드를 요청할 때 사용되어 진다.

java.security.Policy

- static Policy getPolicy()

  현재의 정책 객체를 얻거나 그렇지 않으면 보안 정책이 효과적이지 않다면 null을 얻는다.
- PermissionCollection getPermissions(CodeSource source)

  주어진 코드 소스와 연관된 퍼미션들을 얻는다.

java.lang.Class

- ProtectionDomain getProtectionDomain()

  해당 클래스를 위한 방어 도메인을 얻거나 이 클래스가 방어 도메인이 없이 로드되어지다면 null을 얻는다.

java.lang.ClassLoader

- ProtectionDomain(CodeSource source,PermissionCollection collection)

  주어진 소스와 퍼미션들을 가지는 방어 도메인을 만든다.
- CodeSource getCodeSource()

  해당 방어 도메인의 소스 코드를 얻는다.
- PermissionCollection getPermissions()

  해당 방어 도메인의 퍼미션들을 얻는다.

java.security.PermissionCollection

- void add(Permission p)

  이 퍼미션 집합에 퍼미션을 추가한다.
- Enumeration elements()

  이 집합의 모든 퍼미션들을 통해 반복하기 위한 열거를 반환한다.

java.security.CodeSource

- Certificate[] getCertificates()
  이 소스 코드와 연관된 클래스 파일 서명을 위한 증서들을 얻는다.
- URL getLocation()
  이 소스 코드와 연관된 클래스 파일들의 위치를 얻는다.

보안 정책 파일들

이전 장에서, 여러분들은 SecureClassLoader가 어떻게 클래스들을 로드할 때 퍼미션을 할당하는지 보았다. 각각의 클래스의 코드 소스를 위한 퍼미션들을 찾기 위한 Policy객체를 요구함으로써. 원칙적으로, 여러분은 코드 소스들로부터 퍼미션들을 매핑하는 것을 수행하기 위해 여러분 자신의 Policy클래스를 설치할 수 있다. 그러나 , 이 장에서 여러분은 JDK1.2 인터프리터가 사용하는 정책 클래스에 관하여 배우게 될 것이다.

노트 : 이 정책 클래스는 JDK홈 디렉토리의 jre/lib하위 클래스에서 설정되어진다. 기본으로, 이 파일은 다음 라인을 포함한다.

```
Policy.provider = sun.security.provider.PolicyFile
```

여러분은 여러분 자신의 정책 클래스를 제공할 수 있고 이 파일을 변화 시킴으로써 그것을 설치할 수 있다.

이 표준 정책은 퍼미션들을 위해 코드 소스를 매핑하기 위한 명령어들을 포함하는 policy files를 읽는다. 여러분은 5장에서 RMISecurityManager를 사용하는 프로그램들에게 네트워크 접근을 승낙해 주기를 요구되어 졌던 이러한 정책 파일들을 보았다.

```
grant codeBase  www.horstmann.com/classes
{   Permission java.io.FilePermission "/tmp/*".  "read,write";
}
```

이 파일은 www.horstmann.com/classes로부터 다운로드되어 졌던 모든 코드에게 tmp 디렉토리에 있는 파일들을 읽고 쓰는 권한을 승인한다.
여러분은 표준 위치에 정책 파일들을 설치할 수 있다. 기본으로 거기에는 두 위치가 있다.

- 자바 플랫폼 홈 디렉토리에 있는 java.policy파일
- 사용자 홈 디렉토리에 있는 .java.policy파일(파일이름의 시작이 마침표임을 주목하라)

노트: 여러분은 java.security 설정 파일내에 있는 이러한 파일들의 위치를 변경할 수 있다. 기본은 다음과 같이 지정되어진다.

policy.url.1 = file : $ { java.home } / lib/security/java.policy

policy.url.2 = file : $ { user.home}/ .java.policy

시스템 관리자는 javasecurity 파일을 수정할 수 있고 다른 서버에 있고 사용자들에 의해 편집되어질 수 없는 정책 URL들을 지정할 수 있다. 그 정책 파일들 내에는 많은 수의 정책 URL들(연속된 수를 가지는)이 있을 수 있다. 모든 파일들의 퍼미션들은 결합되어진다.

테스트를 하는 동안, 우리는 계속적으로 이러한 표준 파일들을 수정하고 싶지 않다. 그러므로 우리는 각각의 응용 프로그램을 위해 요구되어지는 정책 파일들을 솔직하게 지정하기를 좋아한다. 단순히 퍼미션들을 개별적인 파일로 놓고, MyApp.policy 라고 말하고 다음과 같은 인터프리터를 시작하라.

java -Djava.security.policy = MyApp.policy MyApp

애플릿을 위해, 여러분은 그 대신에 다음을 사용한다.

appletviewer –J-Djava.security.policy = MyApplet.policy MyApplet.html

(여러분은 해독기에 명령라인 변수를 넘겨주기 위해 appletviewer의 -J옵션을 사용할 수 있다.)

이러한 예들에서, MyApp.policy파일은 효과적으로 다른 정책들이 추가되어진다. 만약 여러분이 두 번째의 등호 표시를 추가한다면, 다음과 같이

java –Djava.security.policy == MyApp.policy MyApp

그때 여러분의 응용 프로그램은 지정된 정책 파일을 유일하게 사용하고 표준 정책 파일은 무시되어진다.

주의: 테스트를 하는 동안 쉬운 실수는 뜻하지 않게 현재 디렉토리에 있는 아마도 AllPermission같은, 많은 퍼미션을 승인하는 java.policy파일을 없애는 것이다. 만약 여러분이

여러분의 응용 프로그램이 여러분의 정책 파일내에서 제약에 주의를 기울지 않는 것 같다면, 여러분의 현재 디렉토리 내에서 남겨진 .java.policy파일을 검사해보라. 만약 여러분이 유닉스 시스템을 사용한다면 이것은 마침표로 시작하는 파일 이름들은 기본으로 보여지지 않기 때문에 특별히 범하기 쉬운 실수이다.

여러분이 전에 본 것처럼, 기본으로, 자바 응용 프로그램들은 보안 관리자를 설치하지 않는다. 그러므로 여러분은 여러분이 하나의 예를 설치할 때까지 정책 파일들의 효과를 알 수 없을 것이다. 여러분은 물론 다음 줄을 여러분의 main 메소드에 추가할 수 있다.

```
System.setSecurityManager(new SecurityManager());
```

또는 여러분은 해독기를 시작할 때 명령 라인 옵션 –Djava.security.manager를 추가할 수 있다.

```
java –Djava.security.manager
    -Djava.security.policy = MyApp.policy MyApp
```

이 장의 나머지 부분에서, 여러분은 정책 파일에서 어떻게 퍼미션들이 표현되어지는지 상세하게 알게 될 것이다. 우리는 이 장에서 후에 다루게 되는 코드 검증을 제외한, 완전한 정책 파일 형태를 설명할 것이다.

정책 파일은 일련의 grant 통관 절차들을 포함한다. 각각의 허가는 다음의 형식을 가진다:

```
grant codesource
{   permission_1;
    permission_2;
    . . .
}
```

코드 소스는 코드 베이스(허가가 모든 소스들로부터 코드를 적용한다면 생략되어질 수 있는)를 포함한다.

그 코드 베이스는 다음과 같이 지정된다.

```
codeBase  “url”
```

만약 URL이 /로 끄나면, 그때 그것은 디렉토리를 말한다. 그렇지 않다면, 그것은 JAR파일의 이름을 말하는 것이다. 예를 들어

grant codeBase "www.horstmann.com/classes/" { . . . }

grant codeBase www.horstmann.com/classes/MyApp.jar { . . . }

그 코드 베이스는 URL이고 심지어 윈도우내의 URL일지라고 항상 파일 분리자로서 순방향 슬래쉬를 포함한다. 예를 들어,

grant codeBase "file:C/myapps/classes/"

노트: 모든 사람들은 http URL이 두개의 슬래쉬(http://)로 시작한다는 것을 안다. 그러나 정책 파일 판독기가 자세히 말해서 file://localFile과 file:localFile의 파일 URL의 두가지 형태들 받아들이는 file UEL에 관해 혼동이 있는 것은 충분한 것 같다. 더욱이, 윈도우 드라이브 문자 앞의 슬러쉬에 대해서. 다음의 모든 것은 허용할만하다.

file:C:/dir/filename.ext

file:/C:/dir/filename.ext

file://C:/dir/filename.ext

file:/// C:/dir/filename.ext

사실상 우리는 file:/// C:/dir/filename.ext가 잘 받아들여 진다는 것을 테스트했다. 그리고 우리는 솔라리스 운영 환경에서, 여러분이 다음의 형태를 사용해야 한다는 것을 설명할 필요가 없다.

file:/dir/filename.ext

퍼미션들은 다음의 구조를 가진다 :

permission className targetName, actionList;

이 클래스 이름은 (java.io.FilePermission과 같은) 퍼미션 클래스의 충분한 자격을 갖춘 클래스 이름이다. target name은 퍼미션 한정적 값이다. 예를 들어 파일 퍼미션이나 소켓 퍼미션에 대한 호스트나 포트를 위한 파일이나 디렉토리 이름. action list 또한 퍼미션 한정적이다. 그것은 콤마에 의해 분리되어지는 read나 connect와 같은 기능들의 리스트이다. 일부 퍼미션 클래스들은 대상이름과 기능 리스트들이 필요하지 않다. 표 9-2는 표준 퍼미션과 그들의 기능들을 싣고 있다.

표 9-2 : 허가(Permission)와 이와 관련된 타겟과 액션

허가 타겟 액션

내용좀 채워 주세요!!!!

여러분이 표 9-2에서 볼 수 있는 것처럼, 대부분의 퍼미션들은 간단히 개별적인 동작을 허가한다. 여러분은 그 동작을 무언의 액션 "permit"을 가지는 목표 대상으로서 간주할 수 있다. 이러한 퍼미션 클래스들은 모두 BasicPermission클래스(그림 9-5 참조)를 상속한다. 그러나 파일,소켓, 그리고 속성 퍼미션들을 위한 대상물들은 더욱 복잡하다. 그리고 우리는 상세하게 그들을 연구할 필요가 있다. 파일 퍼미션 대상물들은 다음의 형태를 가질 수 있다 :

file	**파일**
directory/	**디렉토리**
directory/*	**디렉토리 내의 모든 파일들**
*	**현재 디렉토리 내의 모든 파일들**
directory/-	**디렉토리 또는 그것의 하위 디렉토리 중 하나에 있는 모든 파일들**
-	**현재 디렉토리 또는 그것의 하위 디렉토리 중 하나에 있는 모든 파일들**
<<ALL FILES>>	**파일 시스템 내의 모든 파일들**

예를 들어. 다음의 퍼미션 등록은 /myapp 디렉토리와 그것의 하위에 있는 모든 파일들에 대한 접근 기회를 준다.

```
permission java.io.FilePermission "/myapp/-",
    "read, write, delete";
```

여러분은 윈도우 파일 이름내에서 역 슬래쉬를 나타내기 위해 \\ 이스케이프 연속을 사용한다.

```
permission java.io.FilePermission :c:\\myapp\\-",
    "read, write, delete";
```

소켓 퍼미션 대상물들은 호스트와 포트 범위를 구성한다. 호스트 명세들은 다음의 형태를 가진다.

Hostname 또는 IPaddress	단일 호스트
Localhost 또는 빈 문자열	로컬 호스트
*.domainSuffix	주어진 접미사로 끝나는 도메인을 가지는 어떤 호스트
*	모든 호스트들

포트 범위들은 선택적이고 형식을 가진다.

: n	단일 포트
: n-	n과 n이상의 모든 포트들
: -n	n과 n이하의 모든 포트들
: n1 – n2	주어진 범위의 모든 포트들

여기에 예제가 있다:

```
permission java.net.SocketPermission
    "*.horstmann.com:8000-8999", "connect";
```

결국, 속성 퍼미션 대상물들은 두 가지 형식 중 하나를 가질 수 있다.

property	지정된 속성
propertyPrefix.*	주어진 접두사를 가지는 모든 속성들

"java.home"과 "java.vm.*" 예제들이 있다.
예를 들어, 다음의 퍼미션 절차는 프로그램이 java.vm으로 시작하는 모든 속성들을 읽는 것을 허락한다.

```
permission java.util.PropertyPermission "java.vm.*",    "read";
```

여러분은 정책 파일들에서 시스템 속성들을 사용할 수 있다. ${property}표시는 속성값에 의해 대체되어진다. 예를 들어, ${user.home}이 사용자의 디렉토리에 의해 대체되어진다. 여기에 퍼미션 등록에 있어 시스템 속성의 전형적인 사용이 있다.

```
permission java.io.FilePermission "${user.home" "read, write";
```

플랫폼 독립적인 정책 파일들을 생성하기 위해, 명백한 / 또는 \\ 분리자들 대신에 file.separator속성을 사용하는 것은 좋은 생각이다. 이것을 더욱 단순하게 하기 위해, 특별한 표시 ${ / }는 S{file.separator}를 위한 줄임말이다. 예를 들어,

```
permission java.io.FilePermission "{user.home} ${ / }-"
    "read, write";
```

은 사용자의 홈 디렉토리와 그것의 하위 디렉토리들내에서 읽고 쓰는 퍼미션을 허가하기 위한 이동 가능한 엔트리이다.

JDK는 여러분이 정책 파일들을 편집하기 위해 사용할 수 있는 policytool이라 불리는 초보적인 툴을 제공한다. 여러분이 이 툴을 시작할 때, 여러분은 정책 파일에서 읽을 수 있다. 그때 이 툴은 그들에게 할당된 허가를 가지는 모든 코드 소스들을 보여준다. (그림 9-7을 보라) 여러분이 "Edit Policy Entry"버튼을 클릭했을 때, 코드 소스를 위한 모든 허가들이 디스플레이 된다. (그림 9-8을 보라). 만약 여러분이 하나의 허가를 선택하고" Edit Permission"버튼을 클릭한다면, 여러분은 여러분이 허가 엔트리의 속성들을 편집하도록 하는 다이얼 로그를 얻는다.(그림 9-9를 보라) 여러분이 볼 수 있는 것처럼, 다이얼로그는 편리할 수 있는 대상물과 동작들을 위한 타당한 선택들을 보여준다.

물론, 이 툴은 대부분의 환경들에 의해 완전히 혼란을 겪는 최종 사용자들을 위해서는 적절하지 않다. 우리는 정책 파일들의 정확한 파일 형식에 대해 걱정하기를 원하지 않는 시스템 관리자에 의해 사용되어지는 관리자 툴을 위한 개념의 증거로 생각하고 있다. 여전히 빠진 것은 비전문가에게 의미 있는 카테고리들(상,중,하 보안과 같은)의 적당한 집합이다 . 일반적인 소견으로서, 우리는 자바 2플랫폼이 확실히 훌륭한 보안 모델을 위한 모든 부분들을 포함하고 있다고 믿는다. 그러나 최종 사용자들과 시스템 관리자들에게 이러한 부분들을 제공해주는 것에서 이익을 얻을 수 있다.

그림 9-7: 코드 소스를 출력하는 정책 도구

그림 9-8 : 코드 소스에 대한 출력을 출력하는 정책 도구

그림 9-9 :정책도구로 허가 편집하기

커스텀 허가

이 절에서 여러분은 어떻게 여러분 자신의 사용자들이 그들의 정책 파일들에서 언급할 수 있는 허가 클래스를 제공할 수 있는지 알게 될 것이다.

여러분의 허가 클래스를 구현하기 위해서, 여러분은 Permission클래스를 상속하고 다음 메소드들을 보충한다:

대상물과 동작 리스트를 위해 두개의 String 전달인자를 가지는 생성자

String getActions()

boolean equals()

int hashCode()

boolean implies(permission other)

마지막 메소드가 가장 중요하다. 허가는 일반적인 허가들일수록 더욱 특별한 것을 내포하는 질서가 있다. 파일 허가를 고려해 보라.

```
p1 = new FilePermission("/tmp/-". "read, write");
```

이 허가는 /tmp 디렉토리와 그것의 하위 디렉토리에 있는 어떤 파일의 읽기와 쓰기를 허락한다.

여기에 이 허가를 내포하는 몇몇 더욱 특별한 허가들이 있다.

```
p2 = new FilePermission("/tmp/-", "read");
p3 = new FilePermission("/tmp/aFile", "read, write");
p4 = new FilePermission("/tmp/aDirectory/-", "write");
```

다른 말로하면, 파일 허가 p1은 다른 파일 허가 p2를 내포하고 있다. 만약

1. p1의 대상 파일 집합이 p2의 대상 파일 집합을 포함하면
2. p1의 기능 집합이 p2의 기능집합을 포함하면

여기에 implies 메소드를 사용하는 예제가 있다. FileInputStream 생성자가 읽기를 위해 파일을 열기를 원한다면, 그것은 그것이 그렇게 하기 위한 허가를 가지고 있는지 검사한다. 검사를 수행하기 위해, 특별한 파일 허가 객체가 checkPermission 메소드에 넘겨진다:

```
checkPermission(new FilePermission(filename, "read"));
```

이제 보안 관리자는 그들이 이 허가를 내포하는지 어떤지 모든 적절한 허가를 요청한다
만약 그들 중 하나가 그것을 내포하면, 그때 검사를 통과한다. 특별히, AllPermission은 모든 다른 허가를 내포한다.

만약 여러분이 여러분 자신의 허가 클래스들을 정의한다면, 그때 여러분은 여러분의 허가 객체를 위한 함축의 적절한 의미를 정의할 필요가 있다. 예를 들어, 여러분이 자바 기술에 의해 에너지를 얻는 셋탑 박스를 위한 TVPermission을 정의한다고 가정하자. 허가는

new TVPermission(“Tommy:2-12:1900-2200”, “watch, record”)

Tommy가 오후 7시와 10시 사이에 채널 2번과 12번 사이에서 텔레비전을 보고 녹화하는 것을 허락한다. 여러분은 이 메소드가 다음과 같은 더욱 특정한 것을 포함하게 하기 위해 implies 메소드를 구현해야 한다.

new TVPermission(“Tommy:4:2000-2100”, “watch”)

허가 클래스 구현

다음 예제 프로그램에서, 우리는 텍스트 영역으로의 텍스트의 삽입을 감지하기 위해 새로운 허가를 구현한다. 프로그램은 여러분이 섹스, 약물, C++과 같은 “불량 단어”를 텍스트 영역으로 추가할 수 없다고 확신한다. 우리는 불량 단어들의 리스트가 정책 파일에 공급될 수 있도록 하기 위해 커스텀 허가를 사용한다.

JtextArea의 다음 하위 클래스는 보안 관리자에게 그것이 새로운 텍스트를 추가하는 것에 동의할지 않을지 보안관리자에게 물어 본다.

```
class WordCheckTextArea extends JtextArea
{  public void append( String text )
   {   WordCheckPermission p
            = new WordCheckPermission(text, “insert”);
       SecurityManager manager = System.getSecurityManager();
       if (manager != null) manager.checkPermission(p);
       super.append(text);
    }
   }
```

만약 보안 관리자가 WordCheckPermission을 주면, 그때 텍스트가 추가된다. 그렇지 않으면, checkPermission 메소드는 예외상황을 던진다.

단어 검사 허가는 두가지의 가능한 한 기능을 가진다 : insert(특정한 텍스트를 추가하기 위한 허가)와 avoid(명백한 불량 단어들을 피하는 텍스트를 추가하기 위한 허가). 여러분은 다음의 정책 파일을 가지고 이 프로그램을 수행해야 한다.

```
grant
{   permission WordCheckPermission “sex, drugs, C++”, “avoid”;
}
```

이 정책 파일은 불량 단어 섹스,약물, C++을 피하는 어떤 텍스트를 추가하기 위한 허가를 허락한다.

WordCheckPermission클래스를 설계할 때, 우리는 implies메소드에 특별한 주의를 기울여야 한다. 여기에 허가 p1이 허가 p2를 내포하는지 어떤지 관리하는 규칙들이 있다.

1. 만약 p1이 기능 avoid를 가지고 p2가 기능 insert를 가진다면, 그때 p2의 대상은 p1에 있는 모든 단어들을 피해야 한다. 예를 들어, 허가

   WordCheckPermission "sex, drugs, C++", "avoid"

   는 다음의 허가를 함축한다.

   WordCheckPermission "Mary had a little lamb", "insert"

2. 만약 p1과 p2 모두 avoid 메소드 기능을 가진다면, 그때 p2의 단어 집합은 p1의 단어 집합에 있는 모든 단어들을 포함해야 한다. 예를 들어, 허가

   WordCheckPermission "sex, drugs, C++", "avoid"

   는 다음의 허가를 내포한다.

   WordCheckpermission "sex, drugs", "avoid"

3. 만약 p1과 p2가 모두 insert기능을 가진다면, 그 때 p1의 텍스트는 p2의 텍스트를 포함해야 한다. 예를 들어, 허가

   WordCheckPermission "Mary had a little lamb", "insert"

   는 다음의 허가를 내포한다.

   WordCheckPermission "a little lamb", "insert"

여러분은 예제 9-5에서 이 클래스의 구현을 찾을 수 있다.
여러분이 Permission클래스의 혼란스럽게 유명한 getName메소드를 가지고 허가 대상물을 검사함을 주목하라.

허가가 정책 파일에서 문자열의 쌍으로 나타내어 지기 때문에, 허가 클래스들은 이러한 문자열들을 분석하기 위해 준비되어져야 할 필요가 있다. 특별히, 우리는 콤마로 분리된 avoid 허가의 불량 단어들의 리스트를 순수한 Set으로 변환하기 위해 다음의 메소드를 사용한다.

```
public Set badWordSet ()
{  StringTokenizer tokenizer
       =  new StringTokenizer (getName (), "," );
   Set  set  =  new HashSet();
   while  (tokenizer.hashMoreTokens())
       set.add(tokenizer.nextToken());
   return set;
}
```

이 코드는 우리에게 집합들을 비교하기 위해 equals와 containsAll 메소드들을 사용하는 것을 허락한다. 2장에서 본 것처럼, 집합 클래스의 equals 메소드는 그들이 순서에 관계없이 동

일한 요소들을 포함한다면 두개의 집합들이 같음을 찾아낸다. 예를 들어, "sex, drugs. C++" 과 "C++,drugs, sex"로 부터의 결과 집합들은 동일하다.

주의: 여러분의 허가 클래스가 공적인 클래스임을 확실히 하라. 정책 파일 로더는 부트 클래스 외부에 패키지 가시성을 가지는 클래스들을 로드할 수 없다. 그리고 그것은 조용히 그것이 찾을 수 없는 클래스들을 무시한다.

예제 9-5에 있는 프로그램은 어떻게 WordCheckPermission클래스가 동작하는지 보여준다. 텍스트 영역에 아무 텍스트나 치고 "Insert"버튼을 눌러라. 보안 검사가 통과하면, 텍스트는 텍스트 영역에 추가되어진다. 만약 그렇지 않으면, 에러 메세지가 나타난다.(그림 9-10을 보라)

그 프로그램이 적합한 정책 파일을 가지고 시작함을 확인하라.

```
java   -Djava.security.policy = PermissionTest.policy   PermissionTest
```

그렇지 않으면, 텍스트를 추가하기 위한 모든 시도들은 실패할 것이다.

그림 9-10: permissionTest 프로그램

주의 : 만약 여러분이 그림 9-10을 주의 깊게 본다면, 여러분은 프레임 윈도우가 잘못 나타내어진 자막 "Java Applet Window"을 가지는 경고 영역을 가지고 있음을 볼 것이다. 윈도우 자막은 java.awt.AWTPermission의 showWindowWithoytWarningBanner 대상물에 의해 결정되어 진다. 만약 여러분이 좋다면, 여러분은 허가를 허가하기 위한 정책 파일을 편집할 수 있다.

예제 9-4: PermissionTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.security.*;
import java.util.*;
import javax.swing.*;

public class PermissionTest
{   public static void main(String[] args)
    {   System.setSecurityManager(new SecurityManager());
       JFrame f = new PermissionTestFrame();
        f.show();
    }
}

class PermissionTestFrame extends JFrame
{   public PermissionTestFrame()
    {   setTitle("PermissionTest");
        setSize(400, 300);
        addWindowListener(
            new WindowAdapter()
            {   public void windowClosing(WindowEvent e)
                {   System.exit(0);
                }
            });

        textField = new JTextField(20);
        JPanel panel = new JPanel();
        panel.add(textField);
        JButton openButton = new JButton("Insert");
        panel.add(openButton);
        openButton.addActionListener(
            new ActionListener()
            {   public void actionPerformed(ActionEvent event)
                {   insertWords(textField.getText());
```

```
            }
        });

        Container contentPane = getContentPane();
        contentPane.add(panel, "North");

        textArea = new WordCheckTextArea();
        contentPane.add(new JScrollPane(textArea), "Center");
    }

    public void insertWords(String words)
    {   try
        {   textArea.append(words + "\n");
        }
        catch (SecurityException e)
        {   JOptionPane.showMessageDialog(this,
                "I am sorry, but I cannot do that.");
        }
    }

    private JTextField textField;
    private WordCheckTextArea textArea;
}

class WordCheckTextArea extends JTextArea
{   public void append(String text)
    {   WordCheckPermission p
            = new WordCheckPermission(text, "insert");
        SecurityManager manager = System.getSecurityManager();
        if (manager != null) manager.checkPermission(p);
        super.append(text);
    }
}
```

예제 9-5 : WordCheckPermission.java

```
import java.security.*;
import java.util.*;

public class WordCheckPermission extends Permission
{   public WordCheckPermission(String target, String anAction)
    {   super(target);
        action = anAction;
    }

    public String getActions() { return action; }

    public boolean equals(Object other)
    {   if (other == null) return false;
        if (!getClass().equals(other.getClass())) return false;
        WordCheckPermission b = (WordCheckPermission)other;
        if (!action.equals(b.action)) return false;
        if (action.equals("insert"))
            return getName().equals(b.getName());
        else if (action.equals("avoid"))
            return badWordSet().equals(b.badWordSet());
        else return false;
    }

    public int hashCode()
    {   return getName().hashCode() + action.hashCode();
    }

    public boolean implies(Permission other)
    {   if (!(other instanceof WordCheckPermission)) return false;
        WordCheckPermission b = (WordCheckPermission)other;
        if (action.equals("insert"))
        {   return b.action.equals("insert") &&
                getName().indexOf(b.getName()) >= 0;
        }
        else if (action.equals("avoid"))
        {   if (b.action.equals("avoid"))
```

```
            {   return b.badWordSet().containsAll(badWordSet());
            }
            else if (b.action.equals("insert"))
           {   Iterator iter = badWordSet().iterator();
               while (iter.hasNext())
              {   String badWord = (String)iter.next();
                  if (b.getName().indexOf(badWord) >= 0)
                     return false;
               }
               return true;
            }
            else return false;
        }
        else return false;
    }

    public Set badWordSet()
    {   StringTokenizer tokenizer
            = new StringTokenizer(getName(), ",");
        Set set = new HashSet();
        while (tokenizer.hasMoreTokens())
            set.add(tokenizer.nextToken());
        return set;
    }

    private String action;
}
```

java.security.Permission

- permission(String name)
  주어진 대상의 이름을 가지는 허가를 만든다.
- String getName()
  이 허가의 대상 이름을 반환한다.
- boolean implies(Permission other)
  이 허가가 다른 허가를 내포하는지 검사한다. 만약 다른 허가가 이 허가에 의해 나타내어지는 상태의 결과를 더욱 특별한 상태 묘사를 한다면 이 경우이다.

커스텀 보안 관리자

이 절에서, 우리는 여러분에게 어떻게 간단하지만 완벽한 보안 관리자를 만드는지 보여줄 것이다. 우리는 그것을 WordCheckSecurityManager라고 부른다. 그것은 모든 파일 접근을 감지하고 여러분이 섹스,약물,C++과 같은 금지된 단어들을 포함하는 텍스트 파일을 열 수 없는 것을 확실하게 한다.
우리는 표준 보안 관리자 클래스의 checkPermission메소드를 오버라이딩 함으로서 파일 접근을 감지한다. 허가가 파일 읽기 허가가 없으면, 그때 우리는 간단히 super.checkPermission을 호출한다. 어떤 파일로부터 읽기가 가능한지를 검사하기 위해 우리는 그 파일을 열고 그것의 내용들을 자세히 살펴야 한다. 우리는 단지 그것이 금지된 단어들 중 어떤 것도 포함하지 않을 때 접근권한을 준다. (우리가 시스템과 속성 파일들에 대한 접근을 방해하기를 원하지 않기 때문에 우리는 .txt확장자를 가진 파일들만을 감지한다.)

```
public class WordCheckSecurityManager extends SecurityManager
{   public void checkPermission ( Permission p )
    {   if   (p   instanceof   FilePermission
             && p.getActions ().equals ("read"))
        {   String filename = p.getName ();
            if (containsBadWords (filename))
                throws new SecurityException ("Bad words in "
                        + filename );
        }
        else   super.checkPermission (p);
       }
       .   .   .
    }
```

노트: 파일 읽기 요구의 알리는 다른 방법은 checkRead 메소드를 오버라이드 하는 것이다. SecurityManager 클래스는 filePermission 객체가 가지는 checjPermission을 호출하기 위해 이 메소드를 구현한다. CheckPermission 메소드를 호출하는 다른 보안 검사를 위해서 30개 정도의 메소드가 있다 – 이 절 끝 부분의 API노트를 보라. 이러한 메소드들은 역사적인 이유들이 존재한다. 그 허가 시스템은 자바 2 플랫폼에서 소개되어졌다. 우리는 여러분이 이러한 메소드들을 오버라이드 하지 말고 대신에 checkPermission 메소드에서 모든 허가 검사들을 수행하기를 추천했다.

우리의 파일 검사 시나리오에서 단지 하나의 함정이 있다. 사건들의 하나의 가능한 흐름을 고려하라.

- 어떤 클래스의 메소드는 파일을 연다.
- 그때 보안 관리자가 동작하고 그것의 checkPermission메소드를 사용한다.
- CheckPermission메소드는 containBadWords메소드를 호출한다.

그러나 containsBadWords메소드는 그것의 내용들을 검사하기 위해 스스로 파일을 읽어야 한다. 이 내용들은 다시 보안 관리자를 호출한다. 보안 관리자가 그것이 호출된 문맥에서 찾아낼 수 있는 방법이 없다면 이것은 아주 뒤떨어진 결과일 것이다. GetClassContext메소드는 그 메소드가 어떻게 호출되어졌는지 알아내기 위한 방법이다. 이 메소드는 그것이 호출이 현재 미결인 모든 클래스들을 주는 클래스 객체의 배열을 반환한다. 예를 들어, 보안 관리자가 처음 호출 되어질 때, 그 배열은 다음과 같다.

```
class   WordCheckSecurityManager
class   SecurityManager
class   java.io.FileInputStream
class   java.io.FileReader
class   SecurityManagerFrame
.   .   .
class   java.awtEventDispatchThread
```

인덱스 0인 클래스는 현재 실행 호출을 한다. 불행히도, 우리는 단지 클래스들뿐만 볼 수 있고 미결의 메소드들의 이름은 볼 수 없다. 보안 관리자가 스스로 파일을 열기를 시도할 때, 그것은 다시 호출되어지고 getClassContext메소드는 다음의 배열을 반환한다.

```
class   WordCheckSecurityManager
class   SecurityManager
class   java.io.FileInputStream
class   java.io.FileReader
class   WordCheckSecurityManager
class   WordCheckSecurityManager
class   SecurityManager
class   java.io.FileInputStream
class   java.io.FileReader
```

```
class  SecurityManagerFrame
. . .
class  java.awt.EventDispatchThread
```

이 경우에, 보안 관리자는 파일 접근을 허가해야 한다. 우리는 어떻게 이것을 할 수 있을까? 우리는 할 수 있는지 없는지 테스트 할 수 있다.

```
getClassContext () [0] == getClassContext () [4]
```

그러나 이 접근은 깨어지기 쉽다. 여기에 그것이 잘못되는 명백한 경우가 있다:
예를 들어, 구현이 변경되어 졌다고 상상해라. 그러면 FileReader생성자는 직접적으로 보안 관리자를 호출한다. 그때 테스트는 그 위치가 배열에서와 같지 않기 때문에 의미가 없어진다. 그것은 미결된 호출들중의 어떤 것이 동일한 보안 관리자로부터 생겨났는지를 테스트하기 위해 보다 더욱 강력하다.

여기에 이 테스트를 수행하는 메소드가 있다. 이 메소드가 checkPermission에 의해 호출 되어 졌기 때문에, 콜 스택에는 적어도 보안 관리자 클래스의 두개의 복사본이 있다. 우리는 처음에 이것을 넘어간다. 그리고 동일한 보안 관리자의 다른 인스턴스를 찾는다.

```
boolean inSameManager()
{  Class[]  cc = getClassContext ();

   // skip past current set of calls to this manager
    int i = 0;
   while  (  i  < cc.length && cc[0] == cc[i])
       i++;

  // check if there is another call to this manager
    while (  i <  cc.length )
      {   if ( cc[0] == cc[I]) return true;
          i++;
      }
      return false;
   }
```

우리는 checkPermission 메소드내에서 이 메소드를 호출한다. 만약 우리가 보안 관리자가 반복적으로 호출됐다면, 그때 우리는 containsBadWords 메소드를 다시 호출할 수 없다.

```
if (p  instances of FilePermission
   && p.getActions (). Equals("read"))
{   if  (inSameManager ())
      return;
   String filename = p.getName ();
  if (containsBadWords (filename) )
    throws new SecurityException ("Bad  words  in " + filename);
}
```

예제 9-6**은 이 보안 관리자를 작동하도록 한다. 이 보안 관리자는** main **메소드 내에 설치되어진다. 프로그램을 실행할 때, 여러분은 파일을 지정할 수 있다. 이 프로그램은 텍스트 영역으로 그것의 내용들을 로드할 것이다. 그러나, 만약 파일이 보안 검사를 실패하면, 이 프로그램은 보안 예외상황을 던진다. 그리고 대신에 어떤 메세지를 보여준다.(그림** 9-11**을 보라) 예를 들어, 여러분이** "Alice in Wonderland "**를 디스플레이 할 수 있지만, 프로그램은** "The Count of Monte Cristo"**를 로드하는 것을 거절한다.**

노트 : 여러분은 왜 우리가 파일 이름을 선택하기 위해 JfileChooser**을 사용하지 않는지 궁금할지도 모른다.** JfileChooser**클래스는 아마 그것들이 파일들이 있는지 그리고 디렉토리가 있는지 알아내기 위해 출력하는 파일들을 읽으려고 시도한다..**

그림 9-11: SecurityManagerTest 프로그램

여러분은 어떻게 여러분이 이 프로그램을 작동시키는지 주의를 기울일 필요가 있다. WordCheckSecurityManager **클래스는 스스로** AllPermission**이 주어질 필요가 있다. 이것에 대한 이유는 미비하다.** WordCheckSecurityManager **클래스는 파일 읽기 허가와는 달리 모든 허가를 위한** SecurityManager **하위 클래스를 호출한다.** Securitymanager **클래스가 허가를 평가할 때, 그것은 콜 스택에 있는 모든 메소드들이 특별한 허가를 받아야 하는지 확인한다.** WordCheckSecurityManager **는 이러한 클래스들 중의 하나이다. 그러나 그것은 시스템 클래스가 아니다. 따라서 여러분은 프로그램의 다른 클래스들에게 모든 권한을 인정하지 않고 그것에게 모든 권한을 명백하게 주어야 한다.**

다른 클래스 파일들로부터 WordCheckSecurityManager클래스 파일들을 분리하기 위해, 클래스 파일만을 포함하는 JAR파일을 만들어라.

```
jar  cvf  WordCheck.jar  WordCheckSecurityManager.class
```

WordCheckSecurityManager.class 파일을 삭제한다.

다음으로,다음의 내용을 가지는 WordCheck.policy 정책 파일을 생성하라.

```
grant  codeBase  "file:WordCheck.jar"
{  permission java.security.AllPermission;
}
```

이 정책은 WordCheck.jar 파일 내에 있는 모든 클래스들에게 모든 허가를 준다. 마지막으로 다음과 같이 응용 프로그램을 시작하라.

```
java   –Djava.security.policy = WordCheck.policy
     -classpath  WordCheck.jar;. SecurityManagerTest.java
```

팁: 만약 여러분이 여러분 자신의 프로그램에서 보안 관리자를 변화시키려고 생각한다면, 여러분은 처음에 여러분이 이전 절에서 설명되어졌던 표준 보안 관리자와 커스텀 허가를 사용할 수 있는지 알아보아야 한다. 보안 관리자를 작성하는 것은 에러가 나기 쉽고 교묘한 보안 결함을 일으킨다. 대신에 허가 시스템이라는 주제와 표준 보안 관리자를 사용하는 것이 훨씬 낫다.

예제 9-6: SecurityManagerTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
```

```
public class SecurityManagerTest
{  public static void main(String[] args)
   {  System.setSecurityManager(new WordCheckSecurityManager());
      JFrame f = new SecurityManagerFrame();
      f.show();
   }
}

class SecurityManagerFrame extends JFrame
{  public SecurityManagerFrame()
   {  setTitle("SecurityManagerTest");
      setSize(400, 300);
      addWindowListener(
         new WindowAdapter()
         {  public void windowClosing(WindowEvent e)
            {  System.exit(0);
            }
         });

      fileNameField = new JTextField(20);
      JPanel panel = new JPanel();
      panel.add(new JLabel("Text file:"));
      panel.add(fileNameField);
      JButton openButton = new JButton("Open");
      panel.add(openButton);
      openButton.addActionListener(
         new ActionListener()
         {  public void actionPerformed(ActionEvent event)
            {  loadFile(fileNameField.getText());
            }
         });

      Container contentPane = getContentPane();
      contentPane.add(panel, "North");
```

```
      fileText = new JTextArea();
      contentPane.add(new JScrollPane(fileText), "Center");
   }

   public void loadFile(String filename)
   {  try
      {  fileText.setText("");
         BufferedReader in
            = new BufferedReader(new FileReader(filename));
         String s;
         while ((s = in.readLine()) != null)
         fileText.append(s + "\n");
         in.close();
      }
      catch (IOException e)
      {  fileText.append(e + "\n");
      }
      catch (SecurityException e)
      {  fileText.append("I am sorry, but I cannot do that.");
      }
   }

   private JTextField fileNameField;
   private JTextArea fileText;
}
```

예제 9-7: WordCheckSecurityManager.java

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

public class WordCheckSecurityManager extends SecurityManager
{  public void checkPermission(Permission p)
   {  if (p instanceof FilePermission
```

```
      && p.getActions().equals("read"))
   {  if (inSameManager())
         return;
      String fileName = p.getName();
      if (containsBadWords(fileName))
         throw new SecurityException("Bad words in "
            + fileName);
   }
   else super.checkPermission(p);
}

boolean inSameManager()
{  Class[] cc = getClassContext();

   // skip past current set of calls to this manager
   int i = 0;
   while (i < cc.length && cc[0] == cc[i])
      i++;

   // check if there is another call to this manager
   while (i < cc.length)
   {  if (cc[0] == cc[i]) return true;
      i++;
   }
   return false;
}

boolean containsBadWords(String fileName)
{  if (!fileName.toLowerCase().endsWith(".txt")) return false;
      // only check text files
   BufferedReader in = null;
   try
   {  in = new BufferedReader(new FileReader(fileName));
      String s;
      while ((s = in.readLine()) != null)
      {  for (int i = 0; i < badWords.length; i++)
```

```
            if (s.toLowerCase().indexOf(badWords[i]) != -1)
               return true;
         }
         in.close();
         return false;
      }
      catch(IOException e)
      {  return true;
      }
      finally
      {  if (in != null)
            try { in.close(); } catch (IOException e) {}
      }
   }

   private String[] badWords = { "sex", "drugs", "c++" };
}
```

java.lang.System

- void setSecurityManager (SecurityManager s)

  이 응용 프로그램의 나머지를 위한 보안 관리자를 설정한다. 만약 s가 null이면, 어떤 동작도 취해지지 않는다. 이 메소드는 현재 보안 관리자가 새로운 보안 관리자의 설치를 허가하지 않으면 보안 예외상황을 던진다.

- securitymanager getSecurityManager ()

  시스템 보안 관리자를 얻는다. 아무것도 설치되지 않으면 null을 반환한다.

java.lang.Securitymanager

- Class[] getClassContext()

  현재 실행 메소드를 위한 클래스 배열을 반환한다. 0의 위치의 인자는 현재 실행 메소드의 클래스이다. 위치 1의 인자는 현재 메소드의 호출 클래스 이다. 메소드 1이 아닌 클래스 이름만이 이용 가능하다.

- void checkCreateClassLoader()

  현재 쓰레드가 클래스 로더를 생성할 수 있는 가를 검사한다.

- void checkAccess(Thread g)

  현재 쓰레드가 stop, suspend, resume, setPrioruty, setName**과** setDaemon **메소드를 쓰레드** g**에서 호출 가능한지를 검사한다.**

- void checkAccess(ThreadGroup g)

  현재 쓰레드가 stop, suspend, resume, setPrioruty, setName과 setDaemon 메소드를 쓰레드 그룹 g에서 호출 가능한지를 검사한다.
- void checkExit(int status)

  현재 쓰레드가 상태코드 status로 가상 머신을 종료 시킬 수 있는지를 검사한다.
- void checkExec(String cmd)

  현재 쓰레드가 시스템 명령어 cmd를 실행 할 수 있는지를 검사한다.
- void checkLink(String lib)

  현재 쓰레드가 동적으로 라이브러리 lib를 로드 할 수 있는지를 검사한다.
- void checkRead(FileDescriptor gd)
- void checkRead(String file)
- void checkWrite(FileDescriptor gd)
- void checkWrite(String file)
- void checkDelete(String file)

  해당 메소드들은 현재 쓰레드가 주어진 파일을 읽고, 쓰고, 삭제할 수 있는지를 검사한다.
- void checkRead(String file, Object context)

  다른 쓰레드가 주어진 파일을 읽을 수 있는지 검사한다. 다른 쓰레드는 getSecurityContext를 호출해야 하고, 호출의 반환값은 context 전달인자의 값으로 넘겨진다.
- void checkConnect(String host, int port)

  현재 쓰레드가 주어진 포트에서 주어진 호스트에 연결할 수 있는지 검사한다.
- void checkConnect(String host, int port, Object context)

  다른 쓰레드가 주어진 포트에서 주어진 호스트에 연결할 수 있는지 검사한다. 다른 쓰레드는 getSecurityContext를 호출해야 하고, 호출의 반환값은 context 전달인자의 값으로 넘겨진다.
- void checkListen(int port)

  현재 쓰레드가 주어진 로컬 포트에서 연결을 기다릴 수 있는지 검사한다.
- void checkAccept(String host, int port)

  현재 쓰레드가 다른 주어진 호스트와 포트로부터 소켓 연결을 받아들일수 있는지 검사한다.
- void checkSetFactory()

  주어진 쓰레드가 소켓이나 스트림 핸들러 팩토리를 설정 가능한지 검사한다.
- void checkPropertiesAccess()
- void checkPropertyAccess(String key)

  이 메소드들은 현재 쓰레드가 시스템 속성들이나 주어진 키를 가지는 시스템 속성

에 접근할 수 있는지 검사한다.

- void checkSecurityAccess(String key)
  쓰레드가 주어진 키 값을 가지는 보안 속성에 접근할 수 있는지 검사한다.
- boolean checkTopLevelWindow(Object window)
  주어진 윈도우를 보안 경고없이 출력할 수 있으면 true를 반환한다.
- void checkPrintJobAccess()
  쓰레드가 프린트 작업에 접근할 수 있는지 검사한다
- void checkSystemClipboardAccess()
  쓰레드가 시스템 클립보드에 접근할 수 있는지 검사한다.
- void checkAwtEventQueueAccess()
  쓰레드가 AWT 이벤트 큐에 접근할 수 있는지 검사한다.
- void checkPackageAccess(String pkg)
  쓰레드가 주어진 패키지로부터 클래스들을 로드할 수 있는지 검사한다. 이 메소드는 일부 클래스 로더의 loadClass 메소드로부터 호출된다.
- void checkPackageDefinition(String pkg)
  쓰레드가 주어진 패키지에 있는 새로운 클래스를 정의할 수 있는지 검사한다. 이 메소드는 종종 커스텀 클래스 로더의 loadClass 메소드로부터 호출된다.
- void checkMemberAccess(Class c1, int member_id)
  쓰레드가 클래스의 멤버에 접근할 수 있는지 검사한다(멤버 ID를 얻는 방법에 대해서는 11장을 참조바람).

java.security 패키지

이전에 우리가 말했던 것처럼 애플릿은 자바 플랫폼 위에서 비정상적으로 시작된 것이었다. 실제로 사람들은 JDK 1.0 보안 모델에 있는 많은 유용한 것들을 이용하지 않고도, 널리 알려진 "nervous text" 애플릿과 같은 움직이는 애플릿을 작성하게 되었다. 예를 들어 JDK 1.0에서 작성된 애플릿들이 아주 제한적인 관리만이 가능해서 여러분 회사의 보안 인트라넷으로부터 애플릿을 실행하는데 상대적으로 위험부담이 적을 지라도 회사 인트라넷으로 사용하기에는 좋지않다. 그것은 빠르게 선에게 명백해졌다. 애플릿이 발생된 곳에 따라서 다른 보안 레벨을 할당할 수 있는 것이 사용자에게 중요했다. 만약 애플릿이 신뢰할 수 있는 공급자로부터 발생하고 변경되지 않았다면 애플릿 사용자는 애플릿에게 보다 많은 권한을 줄지를 결정할 수 있다.

이 추가된 제어는 자바 1.1의 애플릿-사이닝 메커니즘 때문에 현재 가능하다. 애플릿에게 보다 많은 신뢰를 주기위해서, 우리는 두 가지를 알 필요가 있다:

1. 애플릿에 어디에서 발생했는가?
2. 코드가 전송도중 손상되었는가?

지난 50년 동안 수학자들과 컴퓨터 과학자들은 데이터 통합의 보증과 전자 서명을 위한 복

잡한 알고리즘들을 발전시켜왔다. java.security 패키지는 이러한 많은 알고리즘의 구현을 포함하고 있다. 다행스럽게도 여러분은 java.security 패키지에 있는 알고리즘을 사용하는데 수학적으로 이해할 필요는 없다. 다음 절에서 여러분은 어떻게 메세지 다이제스트가 데이터 파일의 변화를 감지할 수 있고 디지털 서명이 서명인의 신원을 증명할 수 있는지 알게 될 것이다.

메세지 다이제스트

메세지 다이제스트는 데이터 블록의 디지털 지문(fingerprint) 이다. 예를 들어, 소위 SHA1(보안 해쉬 알고리즘 #1)은 얼마나 길든지 간에 어떠한 데이터 블록을 160비트(20 바이트)의 시퀀스로 간략화 한다. 실제 지문처럼 두개의 메세지가 동일한 SHA1 지문을 가지지 않기를 바란다. 물론, 반드시 동일한 지문을 가지는 어떠한 메세지들이 있기 때문에 단지 2^{160} 개의 SHA1 지문들이 있다는 것은 사실이 아니다. 그러나 2^{160} 은 중복 발생의 가능성을 무시할 수 있을 정도로 아주 크다. True Odds-How Risks Affect Your Everyday Life[Merritt Publishing 1996] 에서 제임스 왈쉬(James Walsh)에 따르면 여러분이 번개를 맞아서 죽을 확률은 약 3만분의 1이다. 지금 예를 들어, 여러분이 좋아하는 경영자나 교수들 중에서 9명의 각기 다른 사람을 생각해보라. 여러분과 그들 모두가 번개를 맞아서 죽을 기회는 원본과 동일한 SHA1 지문을 가지는 위조된 메세지의 확률보다 크다.(물론 여러분이 알지 못하는 10명 보다 많은 사람들이 번개에 의해 죽을 것이다. 그러나 우리는 여러분의 특별한 선택에 대해서 자그마한 기회도 없어진다는 이야기를 하고 있다.)

메세지 다이제스트는 두 가지의 기본적인 속성을 가진다.

1. 데이터의 한 비트 또는 여러 개의 비트들이 변한다면 메세지 다이제스트 또한 변한다.
2. 주어진 메세지를 소유한 사람은 원본과 동일한 메세지 다이제스트를 가지는 위조된 메세지를 만들 수 없다.

두 번째 속성은 물론 다시 확률의 문제이다. 백만장자 아버지에 의한 다음의 메세지를 고려해보자:

"내가 죽으면, 나의 재산을 내 아이들에게 동일하게 나누어 줄 것이다. 그러나 내 아들 조지는 아무것도 받지 못할 것이다."

이 메세지는 다음의 SHA1 지문을 가진다

2D 8B 35 F3 BF 49 CD B1 94 04 E0 66 21 2B 5E 57 70 49 E1 7E

의심이 많은 아버지는 한 명의 대리인에게는 메세지를, 다른 사람에게는 지문을 보관했다. 지금 조지가 메세지를 가지고 있는 변호사에게 뇌물을 줄 수 있다고 가정하자. 그는 그 메세지를 빌이 아무것도 얻지 못하도록 변경하기를 원한다. 물론 지문도 완전히 다른 비트 패턴으로 변한다.

2A 33 0B 4B B3 FE CC 1C 9D 5C 01 A7 09 51 0B 49 AC 8F 98 92

조지가 지문에 맞는 어떤 다른 단어를 찾을 수 있을까? 만약 그가 지구가 형성된 이래로 100만개의 메세지를 계산하는데 1초가 걸리는 10억대의 컴퓨터를 가진 사람이었을지라도, 그는 대체할 메세지를 발견하지 못했을 것이다.

많은 알고리즘이 메세지 다이제스트를 계산하기 위해 설계되어지고 있다. 두 개의 잘 알려진 국립 표준 기술 연구소에 의해서 발전되어진 보안 해쉬 알고리즘인 SHA1 과 MIT의 로널드 리베스트(Ronald Rivest)에 의해 발명된 알고리즘인 MD5이 있다. 두 알고리즘은 교묘한 방식으로 메세지의 비트들을 섞는다. 이 알고리즘에 관한 상세한 내용은 윌리엄 스탈링스(William Stallings)의 저서 Network and Internetwork Security[Prentice-Hall 1995]를 참조 바란다. 최근에는 MD5에서 희박한 규칙성이 발견되었고, 몇몇 암호 작성자들은 이러한 이유 때문에 MD5를 피하고 SHA1을 사용하기를 권장하고 있다.(두가지 알고리즘은 계산하기 쉽다)
자바 프로그래밍 언어는 SHA1과 MD5를 모두 구현한다. MessageDigest 클래스는 지문 알고리즘을 캡슐화하는 객체를 생성하기 위한 팩토리이다. 그것은 MessageDigest 클래스를 상속하는 클래스의 객체를 반환하는 getInstance라고 불리는 정적 메소드를 가진다. 이것은 MessageDigest 클래스가 이중의 역할을 수행함을 뜻한다:

- 팩토리 클래스로서
- 모든 메세지 다이제스트 알고리즘을 위한 상위 클래스로서

예를 들어 여기에 여러분이 SHA 지문을 계산할 수 있는 객체를 얻는 방법이 있다.

```
MessageDigest alg = MessageDigest.getInstance("SHA-1");
```

(MD5를 계산할 수 있는 객체를 얻기위해서는 getInstance의 인자로써 "MD5" 문자열을 사용하라)

여러분이 MessageDigest 객체를 얻은 후에 여러분은 반복적으로 update 메소드를 호출 함으로써 메세지에 있는 모든 바이트들에게 그것을 공급한다. 예를 들어 다음의 코드는 지문을 채취하기 위해 위에서 생성된 alg 객체에게 파일의 모든 바이트를 넘겨준다.

```
FileInputStream in = new FileInputStream(f);
int ch;

while ((ch = in.read()) != -1)
        alg.update((bute)ch);
```

여러분이 마쳤을 때 digest 메소드를 호출하라. 이 메소드는 지문 채취 알고리즘의 요구에 의해 계산을 하고 바이트의 배열로써 다이제스트를 반환해서 입력을 바꾸어 준다.

```
byte[] hash = alg.digest();
```

예제 9-8 프로그램은 SHA 또는 MD5를 사용하여 메세지 다이제스트를 개선한다. 여러분은 파일로부터 다이제스트된 데이터를 로드할 수 있거나 텍스트 영역에 메세지를 입력할 수 있다. 그림 9-12는 그 응용프로그램을 보여준다.

그림 9-12: Computing a message digest

예제 9-8: MessageDigestTest.java

```
import java.io.*;
import java.security.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MessageDigestTest
{   public static void main(String[] args)
   {  JFrame f = new MessageDigestFrame();
      f.show();
   }
}

class MessageDigestFrame extends JFrame
{   public MessageDigestFrame()
   {   setTitle("MessageDigestTest");
      setSize(400, 200);
      addWindowListener(
         new WindowAdapter()
         {   public void windowClosing(WindowEvent e)
            {   System.exit(0);
            }
         });

      JPanel panel = new JPanel();
      ButtonGroup group = new ButtonGroup();
```

```
ActionListener listener =
    new ActionListener()
    {  public void actionPerformed(ActionEvent event)
       {  JCheckBox b = (JCheckBox)event.getSource();
          setAlgorithm(b.getText());
       }
    };
addCheckBox(panel, "SHA-1", group, true, listener);
addCheckBox(panel, "MD5", group, false, listener);

Container contentPane = getContentPane();

contentPane.add(panel, "North");
contentPane.add(new JScrollPane(message), "Center");
contentPane.add(digest, "South");
digest.setFont(new Font("Monospaced", Font.PLAIN, 12));

setAlgorithm("SHA-1");

JMenuBar menuBar = new JMenuBar();
JMenu menu = new JMenu("File");
JMenuItem fileDigestItem = new JMenuItem("File digest");
fileDigestItem.addActionListener(
    new ActionListener()
    {  public void actionPerformed(ActionEvent event)
       {  loadFile();
       }
    });
menu.add(fileDigestItem);
JMenuItem textDigestItem
   = new JMenuItem("Text area digest");
textDigestItem.addActionListener(
    new ActionListener()
    {  public void actionPerformed(ActionEvent event)
       {  String m = message.getText();
          computeDigest(m.getBytes());
```

```
            }
        });
    menu.add(textDigestItem);
    menuBar.add(menu);
    setJMenuBar(menuBar);
}

public void addCheckBox(Container c, String name,
   ButtonGroup g, boolean selected, ActionListener listener)
{  JCheckBox b = new JCheckBox(name, selected);
    c.add(b);
    g.add(b);
    b.addActionListener(listener);
}

public void setAlgorithm(String alg)
{   try
   {   currentAlgorithm = MessageDigest.getInstance(alg);
       digest.setText("");
    }
    catch(NoSuchAlgorithmException e)
    {   digest.setText("" + e);
    }
}

public void loadFile()
{  JFileChooser chooser = new JFileChooser();
    chooser.setCurrentDirectory(new File("."));

    int r = chooser.showOpenDialog(this);
    if(r == JFileChooser.APPROVE_OPTION)
    {   String name
           = chooser.getSelectedFile().getAbsolutePath();
       computeDigest(loadBytes(name));
    }
}
```

```
public byte[] loadBytes(String name)
{   FileInputStream in = null;

    try
    {   in = new FileInputStream(name);
        ByteArrayOutputStream buffer
            = new ByteArrayOutputStream();
        int ch;
        while ((ch = in.read()) != -1)
            buffer.write(ch);
        return buffer.toByteArray();
    }
    catch (IOException e)
    {   if (in != null)
        {   try { in.close(); } catch (IOException e2) {}
        }
        return null;
    }
}

public void computeDigest(byte[] b)
{   currentAlgorithm.reset();
    currentAlgorithm.update(b);
    byte[] hash = currentAlgorithm.digest();
   String d = "";
    for (int i = 0; i < hash.length; i++)
    {   int v = hash[i] & 0xFF;
        if (v < 16) d += "0";
       d += Integer.toString(v, 16).toUpperCase() + " ";
    }
    digest.setText(d);
}

private JTextArea message = new JTextArea();
private JTextField digest = new JTextField();
```

```
    private MessageDigest currentAlgorithm;
}
```

java.security.MassageDigest

- static MessageDigest getInstance(String algorithm)
  지정된 알고리즘을 구현하는 MessageDigest 객체를 반환한다. 만약 알고리즘이 제공되지 않으면 NoSuchAlgorithmException을 던진다.
  전달인자 : algorithm "SHA-1" 또는 "MD5"와 같은 알고리즘의 이름
- void update (byte input)
- void update (byte[] input)
- void update (byte[] input, int offset, int len)
  지정된 바이트를 사용하여 다이제스트를 업데이트한다.
- byte[] digest ()
  해쉬 계산을 마치고, 계산된 다이제스트를 반환하고 그리고 알고리즘 객체를 재설정한다.
- void reset ()
  다이제스트를 재설정한다.

디지털 서명

지난 절에서, 여러분은 원본 메세지를 위한 지문인 메세지 다이제스트를 어떻게 계산하는지 보았다. 만약 메세지가 변경된다면, 그때 변경된 메세지의 지문은 원본의 지문과 맞지 않을 것이다. 만약 메세지와 그것의 지문이 각각 전달되어진다면, 그때 수신인은 그 메세지가 변경되어졌는지 검사할 수 있다. 그러나, 만약 메세지와 지문 모두가 도중에 도청 되어졌다면, 메세지를 변경하고 지문을 재계산하는 것은 쉬운 문제이다. 결국, 메세지 다이제스트 알고리즘은 공개적으로 알려져 있고, 그들은 어떤 비밀 키도 요구하지 않는다. 그런 경우에, 위조된 메세지와 재계산된 지문의 수신인은 그 메세지가 변경되었다는 것을 결코 알 수 없다. 이 절에서, 여러분은 어떻게 디지털 서명이 메세지 다이제스트를 인증할 수 있는지 알게 될 것이다. 메세지가 인증되면, 우리는 다음의 사실을 안다.

- 메세지가 변경되지 않았다.
- 메세지가 승인된 송신자로부터 왔다.

디지털 서명이 어떻게 동작하는지 이해하기 위해서, 우리는 *공개키 암호화라고* 불리는 분야의 개념을 설명할 필요가 있다. *공개키 암호화 기법*은 공개 키와 비밀 키의 개념을 바탕으로 한다. 그 이념은 여러분이 여러분의 공개 키를 세상에 있는 모든 사람에게 말할 수 있다는 것이다. 그러나, 여러분이 가지고 있는 비밀키는 여러분이 그것을 지키고 다른 사람들에

게 그것을 공개하지 않는 것이 중요하다. 그 키들은 수학적인 관계에 의해 맞추어진다. 그러나 하나로부터 다른 것을 계산하는 것은 불가능하다고 생각되어진다. 모든 사람들이 여러분의 공개 키를 알고 있을지라도 그들은 여러분이 살아 있는 동안 여러분의 비밀 키를 계산할 수 없다. 그들이 이용할 수 있는 계산 자원이 얼마나 많이 있느냐는 문제가 아니다.

아무도 공개 키로부터 비밀 키를 계산할 수 없다는 사실을 믿기란 어려울지도 모른다. 그러나 아무도 오늘날 일반적으로 사용되어지는 암호화 알고리즘을 위해 이러한 일을 하는 알고리즘을 찾아내지 못했다. 그 키들이 충분히 길다면, 단순히 모든 가능한 키를 시도하는 강제적인 방법에는 수 천년동안 생성되어진 태양계 내에 있는 모든 원자들로부터 만들어질 수 있는 그 이상의 컴퓨터가 요구되어진다. 물론 어떤 사람이 이러한 강제적인 방법보다 더욱 영리한 계산 키를 위한 알고리즘을 내놓는 것은 가능하다. 예를 들어, RSA알고리즘(리베스트, 샤미르, 아델만에 의해 발명된 암호화 알고리즘)은 큰 수들을 인수분해 하는 어려움에 의존적이다. 지난 20년 동안, 많은 뛰어난 수학자들이 훌륭한 팩토링 알고리즘을 내놓기 위해 시도해 왔다. 그러나 지금까지 어떠한 성공도 없었다. 이러한 이유 때문에, 대부분의 암호 해독자들은 2000 또는 그 이상의 비트의 로그계수를 가지는 키들이 현재 어떠한 공격으로부터 라도 완벽하게 안전하다고 믿는다.

암호화를 위해서 그리고 인증을 위해서 공개 키와 비밀 키 쌍에는 두 가지 종류가 있다.
어떤 사람이 여러분에게 공개 암호화 키를 가지고 암호화 되어진 메세지를 보낸다면, 그때 여러분은 비밀 해독 키를 가지고 그것을 해독할 수 있다. 그러나 다른 사람은 그 누구도 그렇게 할 수 없다. 반대로, 여러분이 여러분의 비밀 인증 키를 가지고 메세지를 서명한다면, 그때에는 그 누구라도 여러분의 공개 키를 가지고 검사함으로써 그 서명을 입증할 수 있다. 그 증명은 여러분이 서명했던 메세지들만을 통과시키고 만약 다른 누군가가 그 메세지를 서명하기 위해 그의 키를 사용했다면 그것은 실패한다. (칸은 몇 백년동안 암호 해독에 있어 첫 새로운 개념이었던 그의 저서 Code Breakers의 새로운 판에서 말하고 있다.)

RSA와 DSA(디지털 서명 알고리즘)와 같은 많은 암호 해독 알고리즘들은 이 개념을 사용한다. 키들의 정확한 구조와 맞추기 위해 그것이 의미하는 것은 알고리즘에 달려 있다. 예를 들어, 여기에 공개와 비밀 DSA키의 대등하는 쌍이 있다.

공개 키:

p:fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a47e6df63413c5e12ed0899bcd132acd50d99151bdc43ee737592e17

q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5

g:678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b71fd73da179069b32e2935630e1c2062354d0da20a6c416e50be794ca4

y:c0b6e67b4ac098eb1a32c5f8c4c1f0e7e6fb9d832532e27d0bdab9ca2d2a8123ce5a8018b8161a760480fadd040b927281ddb22cb9bc4df596d7de4d1b977d50

비밀 키:

p:fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a47e6df63413c5e12ed0899bcd132acd50d99151bdc43ee737592e17

q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5

g:678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b71fd73da179069b32e2935630e1c2062354d0da20a6c416e50be794ca4

x: 146c09f881656cc6c51f27ea6c3a91b85ed1d70a

이들 키 사이에는 수학적인 관계가 있다. 그러나 관계의 정확한 성질은 실제 프로그래밍에서는 재미가 없다. (만약 여러분이 관심이 있다면, 여러분은 이전에 언급했던 윌리암 스탈링의 저서 Network and Internetwork Security[Prentice-Hall 1955]**또는** The Handbook of Cryptography**에서 그것을 찾을 수 있을 것이다.)**

명백한 질문은 어떻게 키 쌍을 생성하느냐는 것이다. 일반적으로, 이것은 여러분에게 키 값을 반환하는 결정적인 과정 속에서 일부 임의의 과정의 결과를 제공함으로써 행해진다. 운좋게, 공개 키 해독을 위한 임의의 키를 얻는 방법은 아무도 의문을 가지지 않는다. 그러나 암호 해독가들과 수학자들은 고민할 필요가 있다.

여기에 실제로 그것이 어떻게 동작하는 지가 있다. 앨리스가 Bob**에게 메세지를 보내기를 원한다고 가정하자. 그리고** Bob**는 이 메세지가 사기꾼이 아니라 앨리스로 부터 온 것인지 알기를 원한다. 앨리스는 이 메세지를 작성하고 그리고는 그녀의 비밀 키를 가지고 메세지 다이제스트를 서명한다.** Bob**은 그녀의 공개 키의 복사본을 얻는다. 그리고는** Bob**은 서명을 확증하기 위해 공개 키를 이용한다. 만약 검증이 통과하면, 그때** Bob**은 두가지 사실을 보증 받는다.**

1. **원본 메세지가 손상되지 않았다**
2. **그 메세지는** Bob**이 검증을 위해 사용했던 공개 키와 맞는 비밀 키의 소유자 앨리**

스에 의해 서명 되어 졌다.
그림 9-13을 참조

그림9-13: DSA를 사용한 공개 키 서명 교환

여러분은 왜 비밀 키를 위한 보안이 아주 중요한지 알 것이다. 만약 누군가 앨리스의 비밀 키를 훔치거나 정부가 그녀에게 그것을 넘겨주기를 요구할 수 있다면, 그때 그녀는 곤란을 겪을 것이다. 도둑이나 정부 직원은 , 자금 전달 명령 또는 등등의 메세지를 보냄으로써 그녀를 흉내낼 수 있다. 다른 사람들은 그것이 앨리스로부터 왔다고 믿을 것이다.

자바 보안 패키지는 DSA를 가지고 만들었다. 만약 여러분이 RSA를 사용하고 싶다면, 여러분은 RSA(www.rsa.com)으로부터 클래스들을 살 필요가 있을 것이다. 동작을 위해 우리는 DSA알고리즘을 쓰도록 하자. 사실상, 거기에는 3가지의 알고리즘들이 있다:

1. 키 쌍을 생성하기 위한 것
2. 메세지를 서명하기 위한 것
3. 서명을 검증하기 위한 것

물론, 여러분은 단지 한번 키 쌍을 생성하고 그것을 서명과 많은 메세지들을 검증하는데 사용한다. 새로운 임의의 키 쌍을 생성하기 위해, 여러분은 정말 임의의 숫자들을 사용하는 것을 확실히 하라. 예를 들어, 현재 날짜와 시간에 의해 만들어지는 Random클래스에 있는 규칙적인 임의의 숫자 생성기는 충분히 무작위가 아니다. (전문가들은 java.util 에 있는 기초적인 임의의 숫자 생성기는 암호 해독적으로 안전하지 않다고 말한다.) 예를 들어, 컴퓨터 시계가 1초의 1/10까지 정확하다고 가정해 보면 날마다 적어도 864,000개의 시드가 있다. 만약 공격자가 키가 생성된 날을 알고 있다면(흔히 만기일을 추론할 수 있는), 그때 그 날을 위한 모든 가능한 시드를 생성하는 것은 쉬운 일이다.

SecureRandom클래스는 Random클래스에 의해 생성되어진 이러한 것보다 더욱 안전한 임의의 숫자들을 생성한다. 여러분은 여전히 임의의 부분에 수열을 시작하기 위한 시드를 제공할 필요가 있을 것이다. 이것을 하기 위한 최고의 방법은 백색 잡음 생성기와 같은 하드웨어 디바이스로부터 임의의 입력을 얻는 것이다. 임의의 입력을 위한 다른 적당한 소스는 사용자에게 임의로 키보드를 치도록 요구하는 것이다. 그러나 각각의 키 두드림은 임의의 시드를 위해 단지 하나 또는 두 비트를 제공해야 한다. 일단 여러분이 바이트의 배열로 그러한 임의의 비트들을 모으면, 여러분은 그것을 setSeed 메소드에게 넘긴다.

```
SecurityRandom secrand = new SecureRandom ();
byte[] b = new byte[20];
// fill with truly random bits
secrand.setSeed(b);
```

만약 여러분이 무작위 숫자 생성기를 시작하지 않는다면, 그때 그것은 쓰레드를 시작함으로써, 그것들을 쉬게 함으로써, 그것이 깨었을 때 정확한 시간을 측정함으로써 자신의 20바이트 시드를 계산할 것이다.

노트: 이것은 이러한 점에서 혁신적인 알고리즘이지만 안전하다고 알려져 있지는 않다. 그리고 과거에 하드 디스크 접근 시간과 같은 컴퓨터의 다른 구성 요소들의 시간조절에 의존하는 알고리즘들은 후에 완벽하게 무작위가 아니라는 것이 증명되었다.

일단 여러분이 생성기를 시작하면, 여러분은 nextBytes 메소드를 가지고 임의의 바이트를 이끌어 낼 수 있다.

```
byte[] randomBytes = new byte[64];
secrand.nextBytes (randomBytes);
```

사실상, 새로운 DSA키를 계산하기 위해, 여러분은 여러분 스스로 임의의 숫자들을 계산하지 않는다. 여러분은 임의의 숫자 생성기 객체를 DSA키 생성 알고리즘에게 넘겨준다.

새로운 키 쌍을 만들기 위해서, 여러분은 KeypairGenerator 객체가 필요하다. 앞 장에서 소개한 MessageDigest 클래스와 마찬가지로, KeyPairGenerator클래스는 팩토리 클래스이며 실제적인 키 쌍 생성 알고리즘을 위한 상위 클래스이다. DSA키 쌍 생성기를 얻기 위해, 여러분은 문자열“DSA”을 가지는 getInstance 메소드를 호출한다.

```
KeyPairGenerator keygen = KeyPairGenerator.getInstance(“DSA”);
```

반환된 객체는 사실상 KeyPairGenerator 클래스의 상위 클래스인 sun.security.provider.DSA – KeyPairGenerator 클래스의 객체이다.

키를 생성하기 위해, 여러분은 Key strength 와 안전한 난수 생성기를 가지고 키 생성 알고리즘을 초기화 해야 한다. Key strength 는 생성된 키의 길이가 아니라 키의 생성 요소 중 하나의 크기이다. DSA의 경우에, 그것은 공개 키와 비밀 키를 만드는 수량 중의 하나인 로그

계수에서 키들의 비트 수이다. 여러분이 512비트의 로그계수를 가지는 키를 만들어 내기를 원한다고 가정해보자.

```
SecureRandom secrand = new secureRandom();
secrand.setSeed(. . .);
keygen.initialize (512, secrand);
```

이제 여러분은 키 쌍을 생성할 준비가 되었다.

```
KeyPair keys = keygen.generateKeyPair ();
KeyPair morekeys = keygen.generateKeyPair ();
```

각각의 키 쌍은 공개 키와 비밀 키를 가진다.

```
publicKey pubkey = keys.getPublic ();
privateKey privkey = keys.getPrivate ();
```

메세지에 서명하기 위해, 여러분은 사인 알고리즘 객체가 필요하다. 여러분은 Signature 팩토리 클래스를 사용한다.

```
Signature signalg = Signature.getInstance (“DSA”);
```

사인 알고리즘 객체들은 메세지를 사인하고 검증하기 위해서 사용되어질 수 있다. 메세지 사이닝을 위한 객체를 준비하기 위해서, initSign메소드를 사용하고 사인 알고리즘을 위한 비밀 키를 넘겨주라.

```
Signalg.initSign (privkey);
```

이제 여러분은 메세지 다이제스트 알고리즘에서와 동일한 방식으로 알고리즘 객체들을 위해 바이트들을 추가하기 위해 update메소드를 사용한다.

```
while ((ch = in.read ()) != -1)
    signalg.update ((byte)ch);
```

결국, 여러분은 sign메소드를 가지는 사인을 계산할 수 있을 것이다. 사인은 바이트의 배열로서 반환되어진다.

```
Byte[] signature = signalg.sign ();
```

메세지의 수신인은 DSA사인 알고리즘 객체를 얻어야 하고 전달인자로서 공개 키를 가지는 initVerify메소드를 호출함으로서 사인 검증을 위하여 그것을 준비해야 한다.

```
Signature verifyalg = Signature.getInstance(“DSA”);
verifyalg.initVerify (pubkey);
```

그때, 그 메세지는 알고리즘 객체에게 보내져야 한다.

```
While ((ch = in.read()) != -1)
```

```
verifyalg.update ((byte)ch);
```

마침내, 여러분은 사인을 검증할 수 있다.

```
boolean check = verifyalg.verify (signature);
```

verify 메소드가 true를 반환하면, 사인은 맞는 비밀 키를 가지고 사인된 메세지의 타당한 사인이었다. 메세지의 송신자와 내용들은 인증되어진다.

예제 9-9는 키 생성, 사이닝, 그리고 검증 순서를 나타내고 있다.

예제 9-9: SignatureTest.java

```
import java.security.*;

public class SignatureTest
{   public static void main(String[] args)
    {   try
        {   KeyPairGenerator keygen
                = KeyPairGenerator.getInstance("DSA");
            SecureRandom secrand = new SecureRandom();
             keygen.initialize(512, secrand);

            KeyPair keys1 = keygen.generateKeyPair();
            PublicKey pubkey1 = keys1.getPublic();
            PrivateKey privkey1 = keys1.getPrivate();

            KeyPair keys2 = keygen.generateKeyPair();
            PublicKey pubkey2 = keys2.getPublic();
            PrivateKey privkey2 = keys2.getPrivate();

            Signature signalg = Signature.getInstance("DSA");
             signalg.initSign(privkey1);
             String message
                = "Pay authors a bonus of $20,000.";
             signalg.update(message.getBytes());
```

```
            byte[] signature = signalg.sign();
           Signature verifyalg = Signature.getInstance("DSA");
            verifyalg.initVerify(pubkey1);
            verifyalg.update(message.getBytes());
            if (!verifyalg.verify(signature))
                System.out.print("not ");
            System.out.println("signed with private key 1");

            verifyalg.initVerify(pubkey2);
            verifyalg.update(message.getBytes());
            if (!verifyalg.verify(signature))
                System.out.print("not ");
            System.out.println("signed with private key 2");
        }
        catch(Exception e)
        {   System.out.println("Error " + e);
        }
    }
}
```

java.security.keyPairGenerator

- static KeyPairGenerator getInstance (String algorithm)
  지정된 알고리즘을 구현하는 KeyPairGenerator **객체를 반환한다. 만약 그 알고리즘이 제공되지 않으면** NoSuchAlgorithmException**을 발생한다.**

  전달인자: algorithm “DSA”**와 같은 알고리즘의 이름**
- void initalize (int strength, SecureRandom random)
  전달인자: strength **알고리즘 특정 수치, 일반적으로, 알고리즘 전달인자들 중 하나의 비트 수**
  random **생성 키들을 위한 임의의 비트 소스**
- keyPair generateKeyPair ()
  새로운 키 쌍을 생성한다.

java.security.KeyPair

- PrivateKey getPrivate ()
  키 쌍으로부터 비밀 키를 반환한다.
- Publickey getPublic ()

키 쌍으로부터 공개 키를 반환한다.

java.security.Signature

- static Signature getInstance (String algorithm)
  지정된 알고리즘을 구현하는 Signature객체를 반환한다. 만약 그 알고리즘이 제공되지 않으면 NoSuchAlgorithmException을 발생한다.

  전달인자: algorithm "DSA"와 같은 알고리즘의 이름

- void initSign (PrivateKey privateKey)
  사이닝을 위한 객체를 초기화 한다. 그 키 타입이 알고리즘 타입과 맞지 않으면 InvalidKeyException을 발생한다.

  전달인자: privateKey 누구의 사인이 계산되고 있는지 확인을 위한 비밀 키
- void update (byte input)
- void update(byte[] input)
- void update (byte[] input, int offset, int len)
  주어진 바이트를 사용하여 메세지 버퍼를 갱신한다.
- byte[] sign ()
  사인 계산을 끝내고 계산된 사인을 반환한다.
- void initVerify (PublicKey publicKey)
  인증을 위한 이 객체를 초기화한다. 만약 키 타입이 알고리즘 타입과 맞지 않다면 InvalidKeyException을 발생한다.

  전달인자: publicKey 검증되어지기 위한 신원의 공개 키
- boolean verify (byte[] signature)
  사인이 타당한지 검사한다.

인증

여러분이 여러분 친구로부터 그의 비밀 키를 가지고 여러분 친구에 의해 사인된 메세지를 얻는다고 가정해 보자. 그것은 우리가 여러분에게 보여 주었던 메소드를 사용한다. 여러분은 이미 그의 공개 키를 가지고 있거나, 또는 그에게 복사본을 요구함으로써 또는 여러분 친구의 웹 페이지로부터 그것을 얻음으로써 그것을 쉽게 얻을 수 있다. 그러면 여러분은 그 메세지가 실제로 여러분 친구에 의해 작성되어 졌고 변경되지 않았다는 것을 검증할 수 있다. 이제 여러분은 유명한 회사를 대표한다고 주장하는 낯선 사람으로부터 메세지를 얻었다고 가정하자. 그 메세지는 여러분에게 덧붙여진 프로그램을 실행하기를 촉구한다. 그 이방인은 여러분이 그가 그 메세지의 저자임을 검증할 수 있도록 여러분에게 심지어 그의 공

개 키의 복사본을 보낸다. 여러분은 그 사인이 타당한지 검사한다. 이것은 그 메세지가 적절한 비밀 키로 사인 되어 졌고 그것이 손상되지 않았다는 것을 증명한다.

주의하라: 여러분은 여전히 누가 그 메세지를 작성했는지 모른다. 어떤 누군가가 공개와 비밀 키의 쌍을 생성했고 비밀 키를 가지고 그 메세지를 사인했으며 그리고 사인된 메세지와 공개 키를 여러분에게 보냈다. 송신자의 신원을 결정하는 문제는 인증 문제라고 불리워 진다.

인증 문제를 해결하는 일반적인 방법은 간단하다. 그 이방인과 여러분이 여러분들 두 사람 모두가 신뢰하는 공통된 친구를 가지고 있다고 가정해 보자. 그 이방인은 개인적으로 여러분의 친구를 만나서 공개 키를 가지는 디스크를 준다고 가정하자. 나중에 여러분의 친구가 여러분을 만나고 여러분에게 그가 이방인을 만났고 그 이방인이 정말 유명한 소프트웨어 회사에서 일한다고 보증한다. 그리고 그 디스크를 여러분에게 준다.(그림 9-14 참조)
여러분의 친구가 이방인의 신뢰성을 보증하는 방식이다.

그림 9-14: 신뢰할 수 있는 중재자를 통한 인증

사실상, 여러분의 친구는 여러분을 만날 필요가 없다. 대신, 그는 이방인의 공개 키 파일을 위해 그의 비밀 사인을 사용할 수 있다 (그림 9-15 참조).

그림 9-15: 신뢰할 수 있는 중재자의 서명을 통한 인증

여러분이 공개 키 파일을 얻었을 때 여러분은 친구의 서명을 검증한다. 그리고 여러분이 그를 믿기 때문에 여러분은 그가 그의 서명을 사용하기 전에 그 이방인의 신원을 검사 했다고 확신한다.
그러나 여러분은 공통된 친구를 가지고 있지 않다. 어떤 신뢰 모델들은 항상 여러분이 그 연결고리의 모든 멤버를 신뢰할 수 있도록 하는 서로 알고있는 연결 고리인 “신뢰의 고리”가 존재한다고 가정한다. 물론 실제로 그것은 항상 사실은 아니다. 여러분은 여러분 친구인 엘리스를 신뢰하고, 엘리스는 밥을 신뢰하고 있지만, 여러분은 밥을 모르고 여러분이 그를 신뢰한다는 것을 확신하지 않는다. 다른 신뢰 모델들은 우리가 모두 신뢰할 수 있는 인정많은 형(Big Brother)이 있다고 가정한다. 어떤 회사들은 Verisign, Inc.(www.verisign.com) 그리고 United State Postal Service와 같은 형이 되기 위해서 일하고 있다.
여러분은 종종 하나 또는 그 이상의 신원을 확인할 수 있는 엔터티들에 의해 사인된 디지털 서명을 만나게 될 것이고, 여러분이 그 확증자들을 어느 정도로 신뢰할지를 평가할 필요가 있을 것이다. 여러분은 아마도 그들의 장황한 인증 실행 보고서를 읽기 때문이거나 새로운

마스터 키가 만들어 질 때 마다 검은색 서류 가방을 가진 다수의 사람들을 밀실안으로 모이도록 요구하는 것을 들었기 때문에 Verisign을 아주 많이 신뢰하고 있을지도 모른다.
그러나, 여러분은 무엇이 인증되어지고 있는가에 관한 실질적인 기대를 가져야 한다. Verisign의 CEO인 Stratton Sclavos는 Verisign에 의해 인증되어진 공개 키를 가진 개인들을 개인적으로 만나지 않는다. 아마도, 개인들은 단지 웹상에서 양식을 채웠을 것이다(그림 9-16 참조).

그림 9-16: 디지털 ID를 위한 요구

이러한 양식은 요청자가 이름, 조직, 국가, 그리고 전자우편 주소를 기술하기를 요구한다. 일반적으로, 그 키(또는 키를 얻는 방법에 대한 지시)는 이 메일 주소로 보내어 진다. 그러므로, 여러분은 아무런 이름이나 조직을 입력하여도 이 메일 주소만은 진짜로 입력하여야 한다. Verisign으로부터 "클래스 1" 아이디를 받았으면 정보는 검증되지 않는다. 거기에는 더욱 엄격한 아이디의 체계가 있다. 예를 들어, "클래스 3" 아이디를 가지고 있다면 Verisign은 개인 요청자에게 공증인 이전에 나타나기를 요구하고, 그것은 요청자 단체의 재정적인 신용도를 검사한다. 다른 확증자들은 다른 절차를 밟게 될 것이다. 따라서 여러분이 인증된 메세지를 받았을 때 그것은 여러분이 무엇을 인증받고 있는지 이해하는 것이 중요하다.

X.509 인증 포맷

사인된 인증서를 위한 가장 일반적인 형식중의 하나는 X.509 형식이다. X.509 인증은 이 메일 메세지를 사인하거나, 프로그램 코드를 인증하거나, 많은 다른 종류의 데이터를 보증하기 위해서 Verisign, Microsoft, Netscape 그리고 다른 많은 회사들에 의해 폭넓게 사용되어지고 있다. X.509 표준은 국제 전화 표준화 기구(CCITT)에 의해서 디렉토리 서비스를 위한 추천되고있는 X.500 시리즈의 일부이다. 그것의 가장 간단한 형식인 X.509 인증은 다음의 데이터를 포함하고있다:

- 인증 형식의 버전
- 인증 시리얼 번호
- 사인 알고리즘 식별자(알고리즘 아이디 + 인증을 사인하기위해 사용되는 알고리즘의 매개 변수들)
- 인증 서명자의 이름
- 유효기간(시작/끝 날짜)
- 인증된 식별자의 이름
- 인증된 식별자의 공개키(알고리즘 아이디 + 알고리즘의 전달인자들 + 공개키 값)
- 사인(서명인의 비밀 키를 가지고 암호화된 필드들에 대한 해쉬 코드)

그러므로, 서명인은 확실한 신원일 경우에는 특별한 공개 키를 가진다는 것을 보증한다. 기

본적인 X.509 형식의 확장은 인증을 위해서 부가적인 정보를 포함하는 것이 가능하도록 만들어준다. X.509 인증 구조의 더욱 상세한 정보에 대해서는 http://www.ietf.cnri.reston.va.us/ids.by.wg/X.509.html 을 살펴보기 바란다. Peter Gutmann의 웹 사이트(http://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.txt)은 X.509 형식 안에서 다른 벤더들에 의해서 구현되어진 차이점에 대해서 많은 재미있고 유용한 설명을 포함하고 있다.

X.509포맷의 정확한 구조는 "abstract syntax notation#1" 또는 ASN.1이라는 공식적인 표기법으로 설명된다. 그림 9-17은 X.509포맷 버전 3의 ASN1정의를 나타내고 있다. 정확한 문법은 우리에게 중요하지 않지만 여러분이 볼 수 있는 것처럼 ASN1은 인증 파일에 대해 정확한 정의를 하고 있다. Basic encoding rules 또는 BER은 이진 파일에서 이 구조를 어떻게 저장하는지 정확하게 설명하고 있다. BER은 정수, 문자열, 비트 문자열,을 어떻게 암호화하는지 설명하고 SEQUENCE,CHOICE 또는 OPTIONAL와 같은 것을 생성한다.

사실상, BER규칙은 독특하지는 않다; 일부 요소들을 지정하는 몇가지의 방법들이 있다. Distinguished encoding rules(DER)은 이러한 모호함을 없애준다. BER 인코딩 형식에 대한 읽기 쉬운 설명을 위해서 우리는 http://www.rsa.com/rsalabs/pubs/PKCS/ 로부터 얻을 수 있는 Burton.S.Kaliski , Jr.의 저서 A Layman's Guide to a Subset of ASN.1, BER, and DER을 추천한다. 여러분은 http://www.cs.auckland.ac.nz/~pgut001/dumpasn1.c. 에서 BER부호화된 파일들을 덤핑하기 위한 유용한 프로그램을 위해 소스 코드를 찾을 수 있을 것이다.

노트: 여러분은 Oliver Dubuission의 저서 ASN-1-Communication Between Heterogeneous Systems[Academic Press 2000]와 John Larmout의 저서 ASN.1 complete (www.nokalva.com/ asn1/larmouth.html)에서 더 많은 정보를 얻을 수 있다.

그림 9-17: ASN.1 definition of X.509v3

인증 생성

JDK는 인증 셋을 생성하고 관리하기 위한 명령 라인 툴인 keytool 프로그램을 가지고 있다. 우리는 이 툴의 기능이 결국 더욱 사용자 친화적인 프로그램인 다른 것들 속에 들어가기를 기대하고 있다. 그러나 당장, 우리는 앨리스가 어떻게 문서를 사인할 수 있고 그것을 밥에게 보내어 밥이 어떻게 그 문서가 정말 사기꾼이 아닌 앨리스에 의해 사인되어 졌는지를 검증하는 것을 보이기 위해 keytool을 사용할 것이다. 우리는 keytool 모든 특징들을 논하

지는 않을 것이므로 완전한 정보를 위해서는 JDK문서를 참조하기 바란다.

keytool 프로그램은 키 보관소, 인증 데이터 베이스 그리고 비밀 키를 관리한다. 키 보관소에 있는 각각의 엔트리는 별명을 가진다. 여기에 앨리스가 키 보관소 alice.store를 만들어 내고 별명 alice를 가진 키 쌍을 생성하는 방법이 있다.

```
keytool -genkey - keystore alice.store -alias alice
```

키스토어를 생성하고 개방할 때, 여러분은 키 스토어 패스워드를 독촉 받게 될 것이다. 예를 들어, 단지 password를 사용하라. 만약 여러분이 어떤 심오한 목적으로 keytool 생성된 키스토어를 사용했다면, 여러분은 적절한 패스워드를 선택하고 비밀 사인 키를 포함하는 이 파일을 보호할 필요가 있었을 것이다.

키를 생성할 때, 여러분은 다음의 정보를 독촉 받게 될 것이다.

```
Enter keystore password: password
여러분의 이름은?
    [Unknown]: Alice Lee
여러분 부서의 이름은?
    [Unknown]: Engineering Department
여러분 조직의 이름은?
    [Unknown]: ACME Software
여러분의 시 또는 소재지의 이름은?
    [Unknown]: Cupertino
여러분의 주 또는 지역 이름은?
    [Unknown]: CA
두 글자의 국가 코드는?
    [Unknown]: US
<CN=Alice Lee, OU=Engineering Department, O=ACME Software, L=Cupertino, ST=CA, C=US>가 맞느냐?
    [no]: Y
```

keytool은 X.500의 키소유자들을 확인하고 발행인을 검증하기 위하여 Common Name(CN),Organizational Unit (OU),Organization (O),Location (L),State (ST), 그리고 Country(C)의 구분된 이름들을 사용한다.

결국, 여러분은 키 패스워드를 지정하거나 키 패스워드와 같은 키 스토어 패스워드를 사용하기 위해 ENTER를 입력해야 한다.

앨리스가 밥에게 그녀의 공개 키를 주기를 원한다고 가정하자. 그녀는 인증 파일을 전달할 필요가 있다.

keytool -export –keystore alice.store -alias alice -file alice.cert

이제 앨리스는 밥에게 인증서를 전달할 수 있다. 밥이 그 인증서를 받았을 때 그는 그것을 프린트할 수 있다:

keytool -printcert -file alice.cert

출력은 다음과 같다.

소유자: CN=Alice Lee, OU=Engineering Department, O=ACME Software, L=Cupertino,ST=CA, C=US

발행자: CN=Alice Lee, OU=Engineering Department, O=ACME Software, L=Cupertino, ST=CA, C=US

시리얼 넘버: 38107867

유효 시작일: Fri Oct 22 07:44:55 PDT 1999 **유효 마감일**: Thu Jan 2006:44:55 PST 2000

인증 지문:

MD5: 5D:00:0F:95:01:30:B4:FE:18:CE:9A:35:0F:C9:90:DD

SHA1:F8:C2:7C:E2:0B:1F:69:E2:6C:31:9A:F6:35:FA:A3:AF:83:81:6A:6A

이 인증은 자기 서명이다. 그러므로, 밥은 이 인증이 타당한 것인지 검사하기 위해 다른 신뢰된 인증서를 사용할 수 없다. 대신에 그는 그는 앨리스에게 전화를 해서 전화를 인증 지문을 그녀에게 읽어준다.

일부 인증 발행인들은 웹 사이트에서 인증 지문들을 발급한다. 예를 들어, JRE는 jre/lib/security디렉토리 내에 키 스토어 cacerts를 포함한다. 그것은 Thawte와 VeriSign으로 부터의 인증서들을 포함한다. 키 스토어의 내용들을 나열하기 위해서 –list옵션을 사용하라.

keytool -list -keystore jre/lib/security/cacerts

키 스토어를 위한 패스워드는 changeit이다. 이 키스토어 안에 있는 인증서들 중의 하나는 다음과 같다.

Thawtepremiumserverca, Fri Feb 12 12:15:26 PST 1999, trustedCertEntry

Certificate fingerprint (MD5):

06:9F:69:79:16:66:90:02:1B:8C:8C:A2:C3:07:6F:3A

여러분은 http://www.thawte.com/certs/trustmap.html의 Thawte 웹 사이트를 방문함으로써 타당한지를 여러분의 인증이 타당한지를 검사할 수 있다.

노트: 물론, 이 검사는 keytool(그리고 keytool프로그램이 호출하는 rt.jar내에 있는 JRE코드)의 복사본이 변경되지 않았다는 것을 여러분이 신뢰하기를 요구한다. 만약 여러분이 편집증 환자라면, 여러분은 파일을 (신뢰된 파일 복사 프로그램을 사용하는) 다른 머신으로 복사해서 JDK의 새롭게 다운로드되어진 복사본을 가지고 그 머신에서 그것을 검사하기를 원할지도 모른다.

한번 밥이 그 인증을 신뢰하면 그는 그의 키스토어로 그것을 읽어 들일 수 있다.

```
Keytool -import -keystore bob.store -alias alice -file alice.cert
```

주의 : 여러분이 충분히 신뢰하지 않는 키스토어에 결코 인증서를 읽어들이지 마라. 일단 인증서가 키스토어에 추가되어진다면, 키스토어를 사용하는 프로그램은 인증서가 사인을 검증하기 위해 사용되어질 수 있다고 가정한다.

이제 앨리스는 밥에게 사인된 문서들을 보내기를 시작할 수 있다. Jarsigner툴은 JAR파일들을 사인하고 검증한다. 앨리스는 간단히 JAR파일에 사인 되어지기 위한 문서를 추가한다.

```
jar cvf document.jar document.txt
```

그때 그녀는 파일에 사인을 추가하기 위해 jarsigner툴을 사용한다. 그녀는 사용하기 위한 키스토어, JAR파일 그리고 키스토어의 별명들을 지정할 필요가 있다.

```
jarsigner -keystore alice.store document.jar alice
```

밥이 파일을 받았을 때, 그는 jarsigner프로그램의 –verify옵션을 사용한다.

```
Jarsigner -verify -keystore bob.store document.jar
```

밥은 키 별명을 지정할 필요가 없다. Jarsigner프로그램은 디지털 서명에 있는 키 소유자의

X.500이름을 알아내고 키스토어내에 있는 적절한 인증서를 찾는다.

만약 JAR파일이 손상되지 않고 사인이 맞다면, 그때 jarsigner프로그램은 출력한다.

Jar verified

그렇지 않다면, 그 프로그램은 에러 메세지를 보여준다.

사이닝 인증

이전 절에서, 여러분은 공개 키를 다른 동료에게 퍼뜨리기 위해 자기 사인된 인증을 어떻게 사용하는지 보았다. 그러나 인증서의 수신자는 발행인이 가진 지문을 검증함으로써 인증서가 타당하다는 것을 보증할 필요가 있었다. 더욱 일반적으로, 인증서는 신뢰된 중재자에 의해 사인되어 진다.

JDK는 사이닝 인증을 위한 툴들을 포함하지 않는다. 이 절에서 여러분은 키스토어로 부터 비밀 키를 가진 인증서를 사인하는 프로그램을 어떻게 작성하는지 보게 될 것아다. 이 프로그램은 그것 자신의 권한내에서는 유용하고 그것은 여러분에게 인증 내용과 키스토어를 접근하기 위한 프로그램을 어떻게 작성하는지 보여준다.

프로그램 내부를 보기 전에, 사용을 위해 그것을 어떻게 두는지 보도록 하자. 앨리스가 그녀의 동료 신디에게 사인된 메세지를 보내기를 원한다고 가정해 보자. 그러나 신디는 사인 지문을 검증하기 위해 그녀에게 사인 파일을 보낸 모든 사람에게 전화를 하기를 원하지는 않는다. 거기에는 신디가 사인을 검증하는 것을 신뢰할 수 있는 엔터티가 필요하다. 예를 들어 우리는 신디가 이 서비스를 수행하기 위해 ACME 소프트웨어의 정보 자원 부(Information Resources Department)를 신뢰한다고 가정할 수 있다. 이 과정을 실험하기 위해 여러분은 추가된 키 스토어 acmesoft.store 를 생성할 필요가 있다. 키를 생성하고 자기 사인된 인증서를 퍼뜨려라.

```
keytool –genkey –keystore acmesoft.store –alias acmeroot
keytool –export –alias acmeroot –keystore acmesoft.store –file acmeroot.cert
```

그리고 신디의 키 스토어에 그것을 구축하라.

```
keytool –import –alias acmeroot –keystore cindy.store –file acmeroot.cert
```

여전히 신디는 루트 인증의 지문을 검증할 필요가 있으나 지금부터 그녀는 간단히 그것에 의해 사인된 모든 인증서를 받아들일 수 있을 것이다.
엘리스가 신디와 ACME 소프트웨어에 있는 모든 사람들에게 메시지를 보내는데 있어서, 그녀는 정보 자원 부에 인증서를 가져가서 사인받을 필요가 있다. 그러나 JDK에 있는 keytool

프로그램은 이 기능을 가지지 않는다. 예제 9-10에 있는 인증 사인 프로그램이 있다. 그 프로그램은 인증 파일을 읽고 키 스토어에 있는 비밀 키를 가지고 그것을 사인한다. 인증된 ACME 소프트웨어의 구성원은 엘리스의 신원을 확인하고 다음과 같이 사인된 인증서를 발급해준다:

java CertificateSigner –keystore acmesoft.store –alias acmeroot –infile alice.cert –outfile alice_signedby_acmeroot.cert

인증 사인 프로그램은 ACME 소프트웨어 키 스토어에 접근 권한을 가지고 있어야 하고, 구성원은 키 스토어 패스워드를 알아야 한다. 명백하게 이것은 민감한 작업이다.
이제 엘리스는 alice_signedby_acmeroot.cert 파일을 신디와 ACME 소프트웨어에 있는 다른 사람에게 준다. 대신, ACME 소프트웨어는 단순히 회사 디렉토리에 그 파일을 저장할 수 있다. 이 파일이 엘리스의 공개 키와 이 키가 정말로 엘리스 것임을 ACME 소프트웨어에 의한 증명을 포함한다는 것을 기억하라.

노트: keytool 프로그램은 키 사이닝을 위한 다른 메커니즘을 지원한다. 옵션 –certreq 는 Thawte와 Verisign과 같은 인증 기관또는 네트스케이프 인증 서버와 같은 자체 인증 구동 소프트웨어에 의해 진행되어질 수 있는 표준 형태로 인증요구를 만들어 낼 수 있다.

신디가 사인된 인증서를 그녀의 키 스토어에 포함시켰을 때 키 스토어는 그 키가 이미 키 스토어에 존재하는 신뢰된 루트 키에의해 사인되어 졌다는 것을 증명하고, 그녀는 인증 지문을 검증하기를 요구 받지 않는다.

주의:이 시나리오는 단지 설명할 목적을 위한 것이다. keytool은 최종 사용자를 위한 툴로서는 정말로 적절하지 않다. keytool은 그것이 이미 신뢰하고있는 동료에 의해 사인되어진 인증서를 묵시적으로 받아들이고, 그것은 사용자에게 그것이 검증할 수 없는 것들을 확인하기를 요청한다. 그것은 결코 인증서를 거절하지는 않는다. 그것은 혼동된 사용자에게는 타당하지 않은 인증서라고 받아들여지기 쉽다.

일단 신디가 루트 인증서와 그녀에게 규칙적으로 문서를 보내는 사람들의 인증서를 추가하면, 그녀는 결코 다시 키 스토어에 대해 걱정할 필요가 없다.
이제 예제 9-10의 소스 코드를 보도록 하자. 처음에 우리는 키 스토어를 로드 한다. KeyStore 클래스의 getInstance 팩토리 메소드는 적절한 타입의 기 스토어 인스턴스를 만든다. Keytool

에 의해 생성된 키 스토어는 “JKS” 타입을 가진다. 이 키 스토어 타입의 제공자는 “SUN”이다.

```
KeyStore store = KeyStore.getInstance(“JKS”, “SUN”);
```

이제 우리는 키 스토어 데이터를 로드할 필요가 있다. 로드 메소드는 입력 스트림과 패스워드를 요구한다. 패스워드는 문자열이 아니라 char[] 배열로서 지정되는 것을 주목하라. JVM은 그들이 가베지 콜렉트 되기 전에 오랜 시간동안 문자열을 보관한다. 예를 들어 스왑 파일의 내용을 검사함으로써 해커들은 잠재적으로 이러한 문자열을 찾을 수 있다. 그러나 문자 배열은 그들이 사용되어진 후에 즉시 비워질 수 있다.

여기 키 스토어를 로딩하기위한 코드가 있다. 우리가 사용후에 즉시 패스워드를 공백으로 채운다는 것을 주목하라.

```
InputStream in = . . . ;
char[] password = . . .;
store.load(in, password);
Arrays.fill(password, ‘ ‘ );
In.close();
```

다음으로, 우리는 사이닝을 위한 비밀 키를 검색하기 위해 getKey 메소드를 사용한다. getKey 메소드는 키 엘리어스와 키 패스워드를 요구한다. 그것의 반환 타입은 Key이고 우리는 우리가 검색된 키가 비밀키 임을 알기 때문에 PrivateKey로 형변환을 한다.

```
char[] keyPassword = . . . ;
PrivateKey issuerPrivateKey = (PrivateKey)store.getKey(alias, KeyPassword);
Arrays.fill(keyPassword, ‘ ‘ );
```

이제 우리는 사인되어질 필요하있는 인증서를 읽기위한 준비가 되어있다. CertificateFactory 클래스는 입력 스트림으로부터 인증서를 읽을 수 있다. 처음에 여러분은 적절한 타입의 팩토리를 얻을 필요가 있다.

```
CertificateFactory factory
    = CertificateFactory.getInstance("X.509");
```

그리고 입력 스트림을 가지고 generateCertificate 메소드를 호출하라.

```
in = new FileInputStream(inname);

X509Certificate inCert
    = (X509Certificate)factory.generateCertificate(in);
in.close();
```

generateCertificate의 반환 타입은 X509Certificate와 같은 구체화된 클래스들의 상위 클래스인

추상화된 Certificate 클래스 이다. 우리가 입력 파일이 X509Certificate를 포함한다는 것을 알기 때문에 우리는 형변환을 사용한다.

주의:거기에는 Certificate라고 불리는 두가지 타입들이 있다: java.security 패키지에서 권장하지않는 인터페이스와 java.security.cert 패키지에있는 추상 클래스는 우리가 사용하기를 원하는 클래스이다. 여러분은 java.security와 java.securitycert 패키지를 여러분의 프로그램에 임포트 한다면 모호함을 해결하고 java.security.cert.Certificate 클래스를 확실히 참조할 필요가 있다.

이 프로그램의 목적은 인증서에서 바이트를 사인하는 것이다. 여러분은 getTBSCertificate 메소드를 통해서 바이트를 검출한다.

```
byte[] inCertBytes = inCert.getTBSCertificate();
```

다음으로, 우리는 사인된 인증서에 넣을 필요가 있는 발행인의 확실한 이름이 필요하다. 그 이름은 키 스토어에 있는 발행인의 인증서에 저장된다. 여러분은 getCertificate 메소드를 가지고 인증서를 가져온다. 인증서가 공적인 정보이기 때문에 여러분은 패스워드가 아닌 별명을 제공한다.

```
X509Certificate issuerCert
  = (X509Certificate)store.getCertificate(alias);
```

getCertificate 메소드는 Certificate 타입을 반환 하지만 다시한번 우리는 반환된 값이 사실상 X509Certificate 라는것을 알고있다. 우리는 getSubjectDN 메소드를 호출함으로써 인증서로부터 발행인의 신원을 얻는다. 그 메소드는 Principal 인터페이스를 구현하는 어떤 타입을 반환한다. 개념적으로 principal 은 사람, 조직, 회사와 같은 실제 세상의 엔터티이다.

```
Principal issuer = issuerCert.getSubjectDN();
```

우리는 또한 사이닝 알고리즘의 이름을 검출한다.

```
String issuerSigAlg = issuerCert.getSigAlgName();
```

이제 우리는 표준 보안 라이브러리의 영역을 떠나야 한다. 표준 라이브러리는 새로운 인증서를 생성하기 위한 어떠한 메소드도 포함하지 않는다. 인증 생성을 위한 라이브러리들은 RSA Inc와 같은 협력 업체로부터 이용이 가능하다. 그러나 우리는 sun.security.x509 패키지들을 사용할 것이다. 일반적으로 절차는 유사하게 적용이 가능하다. Java technology와 Sun Microsystems의 협력 업체에 의해 제공된 것이 아닌 패키지는 언제든지 작동 상태가 변경될

지도 모른다.

다음 코드 세그먼트는 이러한 절차를 수행하고 있다.

- 사인 되어진 인증서 내에있는 바이트로부터 인증 정보 객체를 생성한다.
- 발행인의 이름을 설정한다.
- 인증서를 생성하고 발행인의 비밀키를 가지고 그것을 사인한다.
- 파일에 사인된 인증서를 저장한다.

```
X509CertInfo info = new X509CertInfo(inCertBytes);
info.set(X509CertInfo.ISSUER,
    new CertificateIssuerName((X500Name)issuer));

X509CertImpl outCert = new X509CertImpl(info);
outCert.sign(issuerPrivateKey, issuerSigAlg);
outCert.derEncode(out);
```

우리는 제품 코드를 위해 선 라이브러리의 사용을 추천하지 않기 때문에 상세하게 이러한 클래스들의 사용을 언급하지 않을것이다. JDK의 차후 버전은 인증 생성에 대한 클래스를 포함할지도 모른다. 그동안 여러분은 협력 업체 라이브러리에 의존하거나 간단하게 존재하고 있는 인증 생성 소프트웨어를 사용하기를 원할지도 모른다.

예제 9-10: CertificateSigner.java

```
import java.io.*;
import java.security.*;
import java.security.cert.*;
import java.util.*;

import sun.security.x509.X509CertInfo;
import sun.security.x509.X509CertImpl;
import sun.security.x509.X500Name;
import sun.security.x509.CertificateIssuerName;

public class CertificateSigner
{   public static void main(String[] args)
    {   String ksname = null; // the keystore name
        String alias = null; // the private key alias
        String inname = null; // the input file name
```

```
String outname = null; // the output file name
 for (int i = 0; i < args.length; i += 2)
 {   if (args[i].equals("-keystore"))
       ksname = args[i + 1];
     else if (args[i].equals("-alias"))
       alias = args[i + 1];
     else if (args[i].equals("-infile"))
       inname = args[i + 1];
     else if (args[i].equals("-outfile"))
       outname = args[i + 1];
     else usage();
 }

 if (ksname == null || alias == null ||
    inname == null || outname == null) usage();

 try
{   PushbackReader console = new PushbackReader(new
         InputStreamReader(System.in));

    KeyStore store = KeyStore.getInstance("JKS", "SUN");
    InputStream in = new FileInputStream(ksname);
     char[] password
       = readPassword(console, "Keystore password");
     store.load(in, password);
    Arrays.fill(password, ' ');
     in.close();

     char[] keyPassword
       = readPassword(console, "Key password for " + alias);
     PrivateKey issuerPrivateKey
       = (PrivateKey)store.getKey(alias, keyPassword);
    Arrays.fill(keyPassword, ' ');

     if (issuerPrivateKey == null)
         error("No such private key");
```

```
        in = new FileInputStream(inname);

        CertificateFactory factory
            = CertificateFactory.getInstance("X.509");


        X509Certificate inCert
            = (X509Certificate)factory.generateCertificate(in);
        in.close();
        byte[] inCertBytes = inCert.getTBSCertificate();

        X509Certificate issuerCert
            = (X509Certificate)store.getCertificate(alias);
       Principal issuer = issuerCert.getSubjectDN();
       String issuerSigAlg = issuerCert.getSigAlgName();

       FileOutputStream out = new FileOutputStream(outname);

       X509CertInfo info = new X509CertInfo(inCertBytes);
        info.set(X509CertInfo.ISSUER,
            new CertificateIssuerName((X500Name)issuer));

       X509CertImpl outCert = new X509CertImpl(info);
        outCert.sign(issuerPrivateKey, issuerSigAlg);
        outCert.derEncode(out);

        out.close();
    }
    catch (Exception exception)
    {   System.out.println(exception);
    }
}

public static char[] readPassword(PushbackReader in,
   String prompt) throws IOException
```

```
{   System.out.print(prompt + ": ");
    System.out.flush();
    final int MAX_PASSWORD_LENGTH = 100;
    int length = 0;
    char[] buffer = new char[MAX_PASSWORD_LENGTH];

    while (true)
    {   int ch = in.read();
        if (ch == '\r' || ch == '\n' || ch == -1
          || length == MAX_PASSWORD_LENGTH)
        {   if (ch == '\r') // handle DOS "\r\n" line ends
            {   ch = in.read();
                if (ch != '\n' && ch != -1) in.unread(ch);
            }
            char[] password = new char[length];
            System.arraycopy(buffer, 0, password, 0, length);
            Arrays.fill(buffer, ' ');
            return password;
        }
        else
        {   buffer[length] = (char)ch;
            length++;
        }
    }
}

public static void error(String message)
{   System.out.println(message);
    System.exit(1);
}

public static void usage()
{   System.out.println("Usage: java CertificateSigner"
       + " -keystore keyStore -alias issuerKeyAlias"
       + " -infile inputFile -outfile outputFile");
    System.exit(1);
```

```
    }
}
```

java.security.Principal

- String getName()

  이 principal의 이름을 반환한다.

java.security.KeyStore

- static getInstance(String type)
- static getInstance(String type, String provider)

  이 메세지들은 주어진 타입의 키 스토어 객체를 만든다. Provider가 지정되지 않으면 기본 provider가 사용된다. keytool에 의해 생성된 키 스토어를 가지고 작업하기 위해서 타입으로서 "JKS"를 provider로서 "SUN"을 지정하라.

- void load(InputStream in, char[] password)

  스트림으로부터 키 스토어를 로드한다. 패스워드는 그것이 JVM의 문자열 풀의 일부가 되지않도록 캐릭터 배열로 유지된다.

- Key getKey(String alias, char[] password)

  이 키 스토어에 저장된 주어진 alias를 가지는 비밀 키를 반환한다.

- Certificate getCertificate(String alias)

  이 키 스토어에 저장된 주어진 alias를 가지는 공개키를 위한 인증서를 반환한다.

java.security.cert.CertificateFactory

- CertificateFactory getInstance(String type)

  주어진 타입을 위한 인증 팩토리를 생성한다. 이 타입은 "X509"와 같은 인증 타입이다.

- Certificate generate Certificate (InputStream in)

  입력 스트림으로부터 인증서를 로드 한다.

java.security.cert.Certificate

- PublicKey getPublicKey()

  이 인증서에 의해 보증 되어지고 있는 공개 키를 반환한다.

- byte[] getEncoded()

  이 인증서의 암호화 형식을 얻는다.

- String getType()

  "X509"와 같은 이 인증서의 타입을 반환한다.

java.security.cert.X509Certificate

- Principal getSubjectDN()
- Principal getIssuerDN()

  인증서로부터 소유자(또는 제목)와 발행인의 확실한 이름을 얻는다.

- Date getNotBefore()
- Date getNotAfter()
  인증서의 유효기간의 시작과 끝 날짜를 얻는다.
- BigInteger getSerialNumber()
  인증서로부터 순차번호 값을 얻는다.
- String getSigAlgName()
- byte[] getSignature()
  인증서로부터 소유자의 사인과 사인 알고리즘 이름을 얻는다.
- byte[] getTBSCertificate()
  인증 발행인에 의해 사인되어질 필요가있는 DER 코드화된 인증 정보를 얻는다.

코드 사이닝

인증 기술의 가장 중요한 사용중의 하나는 실행 가능한 프로그램을 사인하는 것이다. 만약 여러분이 프로그램을 다운로드 할 때 자연스럽게 그 프로그램이 줄지도 모르는 손상에 관해서 관심을 가지게 된다. 예를 들어 그 프로그램이 바이러스에 의해 감염되어 있을 수도 있다. 만약 여러분이 그 코드가 어디에서부터 왔고 그것이 출처로부터 손상을 입지 않았다는 것을 알고 있다면 그때 여러분은 이것을 모를때보다 더 안심할 수 있다. 만일 사실 프로그램이 자바 언어로 작성되어져 있다면 여러분은 그때 프로그램이 가지도록 허락할 수 있는 권한에 관한 타당한 결정을 하기위해 이러한 정보를 사용할 수 있다. 여러분은 단지 일반적인 애플릿처럼 샌드박스에서 실행하기를 원하거나 서로 다른 권한과 제한의 집합을 그것에게 줄려고 할지도 모른다. 예를들어 여러분이 단어처리 프로그램을 다운로드 한다면 여러분은 그것에게 프린트와 어떤 하위 디렉토리 내에 있는 파일들에 대한 접근을 승인하기를 원할지도 모른다. 그러나 프로그램이 여러분도 모르게 여러분의 파일을 협력 업체에게 보내지 못하도록 네트워크 연결을 할 수 있는 권한을 주고싶지 않을수도 있다.
여러분은 지금 이 복잡한 구조를 어떻게 구현하는지 알고있다.

- 먼저, 그 코드가 어디에서 왔는지 검증하기위한 인증을 사용하라
- 그 다음에 그것의 출처에 따라 그 프로그램을 승인하기를 원하는 허가를 강요하는 보안정책을 가지고 그 코드를 실행하라.

JAR 파일 사이닝하기

이 시점에서 코드 사이닝은 여전히 플랫폼 종속적이다. 만일 여러분이 Netscape나 Internet Explorer를 가지고 JDK 1.1 애플릿을 사용한다면 여러분이 애플릿을 사인하기위해 Netscape 또는 Microsoft 도구를 사용할 필요가 있다. 이 주제에 대한 좀더 상세한 정보를 얻기 위해서는 www.securingjava.com/appdx-c/ 또는 www.suitable.com/Doc_CodeSigning.shtml 을 참조하기 바란다.
이 절에서 우리는 여러분에게 자바 플러그 인을 가지고 사용하기 위해서 애플릿을 사인하는

방법을 보여준다. 여기에 두가지의 시나리오가 있다.

1. 사인된 인트라넷 애플릿
2. 사인된 인터넷 애플릿

처음 시나리오에서 시스템 관리자는 인증서와 정책 파일을 로컬 장비에 설치한다. 자바 플러그인이 사인된 애플릿을 로드할 때 마다 그것은 사인과 애플릿 허가를 위한 정책 파일을 위해 키 스토어를 참조한다. 인증서와 정책을 설치하는 것은 수월하며 각 데스크탑마다 한 번만 설치하면 된다. 최종 사용자는 그때 샌드박스 밖에서 사인된 사내 애플릿을 실행 할 수 있다. 새로운 애플릿이 생성되어 지거나 존재하는 애플릿이 갱신될 때 마다 그것은 반드시 사인 되어지고 웹서버에 배치되어야 한다. 그러나 애플릿이 발전하더라도 데스크탑에서는 관여할 필요가 없다. 우리는 이것이 모든 데스크탑에서 사내 응용 프로그램을 배치하는데 있어서 재미있는 선택이 될수 있는 납득할만한 시나리오라고 생각한다.

두번째 시나리오에서 소프트웨어 업체는 Thawte과 Verisign과 같은 인증기관에 의해 사인된 인증서를 가진다. 최종 사용자가 사인된 애플릿을 포함하는 웹 사이트를 방문할 때 소프트웨어 업체를 확인하는 팝업 상자가 나타나서 최종 사용자에게 모든 권한을 가지고 애플릿을 실행하거나 샌드박스 내에 그것을 국한시키는 것의 두가지 선택이 주어진다. 우리는 나중에 상세하게 이 시나리오를 설명한다.

이 절의 나머지 부분에서 우리는 여러분이 알려진 소스로부터 애플릿에게 특별한 권한을 승인하는 정책 파일을 어떻게 만드는지 설명한다. 이러한 정책 파일을 만드는것과 배치하는 것은 일시적인 최종 사용자를 위한 것이 아니다. 그러나 시스템 관리자들은 인트라넷 애플릿을 분배하기 위한 준비하는 작업을 수행할 수 있다.
ACME 소프트웨어는 최종 사용자들이 로컬 파일 접근을 요구하는 어떤 애플릿을 실행하기를 원한다고 가정해보자. 이 애플릿들이 샌드박스 내에서 실행될 수 없기 때문에 ACME 소프트웨어는 사용자 머신에 정책 파일을 설치할 필요가 있다. 이전 장에서 여러분이 보았던 것 처럼 ACME는 그들의 코드 베이스에 의해 애플릿을 확인할 수 있었다. 그러나 그것은 ACME 애플릿 코드가 다른 웹 서버로 옮겨 질 때 마다 정책 파일을 갱신할 필요가 있다는 것을 의미한다. 대신 ACME는 애플릿 코드를 포함하는 JAR 파일을 사인하기로 결정했다.
사인된 JAR 파일을 만들기위해, 처음에 클래스 파일들을 JAR 파일로 묶어라.

```
jar cvf MyApplet.jar *.class
```

그리고 jarsigner 도구를 실행하고 JAR 파일과 비밀키의 알리어스를 지정하라.

```
jarsigner –keystore acmesoft.store MyApplet.jar acmeroot
```

이 예제에서 JAR 파일은 자기 사인된 루트 키를 가지고 사인되어 진다. 실제로 그것은

루트 키에 의해 그들 스스로 사인된 개별적인 키들을 ACME가 그들의 프로그래머들에게 주었지도 모른다.

물론, 루트 키를 포함하고 있는 키 스토어는 안전한 장소에 보관되어 져야 한다. 인증을 위한 두번째 키스토어 certs.store를 만들고 그것에 acmeroot인증을 추가해 보자.

```
keytool -export -keystore acmesoft.store -alias acmeroot
  -file acmeroot.cert
keytool -import -keystore certs.store -alias acmeroot -file acmeroot.cert
```

다음으로, 여러분은 그 키스토어내에 있는 사인을 가지고 사인된 모든 애플릿을 위한 허가를 주는 정책 파일을 생성할 필요가 있다.

여러분은 정책 파일내에 여러분의 키스토어 위치를 포함해야 한다. 정책 파일의 상단에 다음 줄을 추가하라.

```
keystore "keystoreURL", "keystoreType";
```

키스토어가 keytool에 의해 생성되어 졌다면 타임은 JKS이다.

```
keystore "file:certs.store", "JKS";
```

정책 파일내에 있는 하나 또는 그 이상의 grant항목에 signedBy "alias"를 추가하라.
예를 들면,

```
grant signedBy "acmeroot"
{ permission java.io.FilePermission "<<ALL FILES>>", "read";
  . . .
};
```

앨리어스와 연관된 공개키를 가지고 검증되어질 수 있는 어떤 사인된 코드는 현재는 이러한 허가를 받는다.

여러분은 예제 9-11에 있는 애플릿을 가지고 이것을 시도할 수 있을것이다. 이 애플릿은 로컬 파일로부터 읽기를 시도하고 있다.(그림 9-18 참조) 기본 보안 정책은 애플릿이 그것의 코드 베이스와 어떤 하위 디렉토리들로부터 파일들을 읽도록 한다. 애플릿을 실행하기 위해 appletviewer를 사용하고 여러분은 다른 디렉토리로가 아닌 코드 베이스 디렉토리로부터 파일들을 읽을 수 있음을 검증하라.

그림 9-18: The FileReadApplet 프로그램

이제 JAR파일내에 그 애플릿을 두고 acmeroot키를 가지고 그것을 사인하라. 그리고 나서 그 내용을 가지는 정책 파일 applets.policy를 생성하라.

```
keystore "file:certs.store", "JKS";
grant signedBy "acmeroot"
{ permission java.io.FilePermission "<<ALL FILES>>", "read";
};
```

노트: 여러분은 정책파일을 만들기 위해 policytool을 또한 사용할 수 있다.

이 테스트를 하는 동안 , 키 스토어, 정책파일, 그리고 JAR파일은 모두 같은 디렉토리내에 있다는 것을 확실히 하라.

마지막으로, appletviewer에게 정책 파일을 사용할 것을 말하라.

```
Appletviewer -J-Djava.security.ploicy=applets.policy FileReadApplet.html
```

이제 애플릿은 모든 파일들을 읽을 수 있다.
이것은 사이닝 메커니즘이 동작한는 것을 보여준다. 실질적으로 이 메커니즘을 알맞게 사용하기 위해서, 여러분은 보안 파일들을 위한 적당한 장소를 찾을 필요가 있다. 우리는 다음 절에서 그 주제를 거론한다.
마지막 테스트와 같이 여러분은 Java Plug-in내에서 여러분의 애플릿을 테스트하기를 원할지도 모른다. 그러면 여러분은 jre/lib/security디렉토리에 있는 java.security파일을 수정할 필요가 있다.
이 부분에 위치하라.

```
# The default is to have a single system wide policy file,
# and a policy file in the user's home directory.
policy.url.1=file:${java.home}/lib/securit/java.policy
policy.url.2=file:${user.home}/ .java.policy
```

다음과 같이 정책 파일을 위한 파일 URL을 가지는 라인을 추가하라.

```
Policy.url.3=file:///home/test/applet.policy
```

주의: 반드시 정식 java.security파일을 수정하라. 플러그 인은 JDK에 속하는 것이 아닌 JRE 내에 있는 가상 머신을 사용한다. 예를 들어, 윈도우체제에서 JRE파일들은 \Program Files\JavaSoft디렉토리내에 위치한다.

만약 여러분이 Netscape 4 또는 Internet Explorer에서 플러그 인을 사용한다면, 여러분은 EMBED또는 OBJECT태그를 가지고 플러그 인을 로드하기 위해 HTML파일을 변경할 필요가 있다. Opera 와 Netscape 5는 여러분이 APPLET태그를 사용할 때 자동적으로 플러그 인을 로드한다.(플러그 인을 로드하는 방법에 대한 더 많은 정보에 대해서는 1권의 10장을 참조하기 바란다)

주의 : 만약 여러분이 appletviewer대신에 자바 플러그 인을 가지고 여러분의 애플릿을 테스트 한다면, 그때 여러분은 플러그 인이 처음 로드되어질 때 단지 한번 정책 파일을 읽는다는 것을 알 필요가 있다. 만약 여러분이 정책 파일이나 키스토어를 잘못 건더렸다면, 그때 여러분은 부라우져를 닫고, 잘못된 것을 고치고 부라우져를 다시 실행할 필요가 있다. 단순히 애플릿을 포함하는 웹 페이지를 다시 읽어들이는 것은 효과가 없다.

예제 9-11: FileReadApplet.java

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;

public class FileReadApplet extends JApplet
{   public FileReadApplet()
    {   fileNameField = new JTextField(20);
        JPanel panel = new JPanel();
        panel.add(new JLabel("File name:"));
        panel.add(fileNameField);
```

```
    JButton openButton = new JButton("Open");
     panel.add(openButton);
     openButton.addActionListener(
         new ActionListener()
         {   public void actionPerformed(ActionEvent event)
             {   loadFile(fileNameField.getText());
             }
         });

     Container contentPane = getContentPane();
     contentPane.add(panel, "North");

     fileText = new JTextArea();
     contentPane.add(new JScrollPane(fileText), "Center");
}

public void loadFile(String filename)
{   try
    {   fileText.setText("");
        BufferedReader in
            = new BufferedReader(new FileReader(filename));
        String s;
        while ((s = in.readLine()) != null)
        fileText.append(s + "\n");
        in.close();
    }
    catch (IOException e)
    {   fileText.append(e + "\n");
    }
    catch (SecurityException e)
    {   fileText.append("I am sorry, but I cannot do that.");
    }
}

private JTextField fileNameField;
private JTextArea fileText;
```

}

전개 팁들

여러분이 해야 할 첫 결정은 어디에 정책 파일을 둘 것인가이다. 두가지의 기본 경로가 있다.

- ${java.home}/lib/security 디렉토리에 있는 java.policy
- ${user.home}디렉토리에 있는 .java.policy

우리는 어느것도 적당한 선택이라고 생각하지 않는다. 우리는 여러분이 ${java.home}/lib/security디렉토리에 있는 java.policy파일을 편집하고 다음과 같이 외부 정책 파일을 포함하기 위해 다음 라인을 추가하기를 권장한다.

```
policy.url.3=http://intranet.acmesoft.com/admin/applet.policy
```

그리고 나서, 여러분은 인트라넷 서버에 정책 파일을 설정할 수 있다. 여러분이 단지 단일 위치에서 파일을 관리할 필요가 있기 때문에 이것은 이점이 있다.
유사하게 정책 파일내에서 여러분은 URL을 가지고 키스토어 위치를 지정할 수 있다:

```
keystore http://intranet.acmesoft.com/certs.store, "JKS";
```

사내 인트라넷 응용프로그램을 위해 공개키를 위한 인증서를 키스토어에 추가하라.

노트: 여러분은 JRE내에 cacerts키스토어를 위한 인증서를 추가할 수 있었다. 그러나 그 때 여러분은 여러분이 키스토어를 변경할 때 모든 데스크 탑에서 JRE를 갱신할 필요가 있었다. 우리는 여러분이 cacerts파일을 건드리지 않기를 권장한다.

마지막으로, 적절한 비밀키를 가지고 여러분의 인트라넷 응용 프로그램을 위한 코드를 포함하는 JAR파일을 사인하라.

우리는 여러분이 간단히 인트라넷 애플릿에게 AllPermission을 승인하는 것이 아니라 특정한 하위 디렉토리내에 있는 모든 디렉토리를 접근하는 것, 프린팅, ,인트라넷 내에서 네트워크 접근 등과 같은 지정된 권한들에 의해 "샌드박스(sandbox)"를 확장할 것을 제의한다.

전개는 약간 복잡하게 들리지만 사실상 아주 다루기 쉽다. 여러분은 각 데스크 탑에서 자바 플러그 인 구성을 최소한으로 맞추어 줄 필요가 있다. 나머지는 원격적으로 관리되어질 수 있다. 거기에는 두가지 이점이 있다. 여러분은 유일한 보안을 결정하기 위해 사용자들에게 부담을 줄 시간이 없고 여러분은 여러분의 사용자들이 실행하는 원격의 코드 권한을 다루기

위한 잘 만들어진 통제권을 가지고 있다.

노트 :자바 플랫폼 보안이 인증을 제한하고 코드 허가를 규제하기 때문에, 그것은 마이크로소프트의 ActiveX기술보다 더욱 폭 넓은 보안 모델을 제공한다. ActiveX코드는 인증을 받지만 그것이 실행을 위한 권한을 일단 받으면 그것은 전혀 제어되어질 수 없다.

소프트웨어 개발 인증

지금까지 우리는 애플릿이 인터라넷에서 전달되어지고 시스템 관리자는 애플릿의 권한을 제어하는 정책 파일을 만드는 시나리오를 얘기했다. 그러나 그 방법은 알려진 소스로 부터의 애플릿인 경우만 동작한다.

여러분이 인터넷을 서핑하다가 여러분이 그것에게 그렇게 할 허가를 승인하다는 조건으로 잘 알지 못하는 업체로부터 애플릿을 실행하기를 제의 받는다고 가정하자. (그림 9-19를 참조)

그러한 애플릿은 Thawte나 Verisign과 같은 인증 기관에 의해 발행된 소프트 웨어 개발자 인증서를 가지고 사인되어 진다. 팝업 창은 소프트웨서 개발자와 인증 발행인을 확인한다. 여러분은 지금 두가지 선택이 있다:

- 모든 권한을 가지고 애플릿을 실행하는 것 또는
- sandbox에 애플릿을 제한하는 것

그림 9-19: 사인된 애플릿 시작

여러분은 결정에 있어서 여러분의 결정에 영향을 미치는 어떤 사실들을 가지고 있느냐? 여기에 여러분이 알고 있는 것이 있다:

1. Thawte는 소프트웨어 개발자에게 인증서를 팔았다.
2. 애플릿은 정말 그 인증서를 가지고 사인되어졌고 전송중에 수정되지 않았다.
3. 인증서는 정말 Thawte에 의해 사인되어 졌다 – 그것은 로컬의 cacerts파일에 있는 공개키에 의해 검증되어진다.

코드를 수행하는 것이 안전한지 여러분에게 말해야 하느냐? 여러분은 여러분이 그 업체의 이름(*.netbizz.dk)이 무엇인지 그리고 Thawte가 소프트웨어 개발자에게 그것들을 팔았다는

사실을 모두 알고 있다면 그 업체를 신뢰하느냐? 생각건대 Thawte는 *.netbizz.dk가 분명한 크래커가 아님을 스스로 보증하는데 약간의 문제가 있을 것이다. 그러나 어떤 발행자도 정직한 이해력있는 검사와 소프트 웨서 업체의 역량을 포함할 수 없다.

잘 알려지지 않은 업체의 입장에서, 최종 사용자는 로컬 응용 프로그램의 모든 권한을 가지고 이 애플릿을 샌드박스 밖에서 실행해야 할지에 관한 지적인 결정을 하는데는 미비하다.
만약 그 업체가 잘 알려진 회사라면, 그때 사용자는 적어도 보고서에 의해 회사의 과거 추적 기록을 얻을 수 있을 것이다.

우리는 프로그램이 "나에게 모든 권한을 달라 또는 나는 아무것도 실행할 수 없다'를 요구하는 상황을 좋아하지 않는다. 순진한 사용자들은 종종 그들을 위험으로 몰아 넣을 수 있는 접근을 승인하도록 협박을 받는다. 만약 애플릿이 그들에게 무슨 권한이 필요하고 이러한 권한을 위하여 지정된 허가를 찾는다고 설명한다면 도움이 되겠는가?

불행하게도, 여러분이 알고 있는 있는 것처럼, 아주 기술적으로 될 수 있다. 애플릿이 정말 AWT 이벤트 큐를 검사해야 하기 위한 권한을 가져야 하는지를 최종 사용자가 고려해야 한다는 것은 합당하지 않은 것 같다. 아마도, 플러그 인의 미래 버전은 최종 사용자들에게 의미있는 허가 집합을 정의할 수 있을 것이다. 그때까지, 우리는 소프트웨어 개발자 인증서를 가지고 사인된 애플릿들에 대해 관심이 없다.

만약 여러분이 여러분의 애플릿을 세계적으로 널리 퍼뜨리기를 원하고 여러분의 사용자들이 그들이 그들의 기계에서 충분한 권한을 가지고 여러분의 애플릿을 허락할 수 있을 정도로 충분히 신뢰한다고 여러분을 신뢰한다고 생각한다면, 그때 여러분은 Thawte또는 Verisign으로부터 인증서를 구입하고 여러분의 애플릿을 사인하기 위해 그것을 사용하기를 원할지도 모른다.
상세한 정보를 위해서는 http://java.sun.com/products/plugin/1.2/docs/nsobjsigning.html 을 참조하기 바란다.

암호화

이 장에서, 우리는 자바 플랫폼 보안 API, 즉 디지털 서명을 통한 인증을 가지고 구현된 하 중요한 암호 해독 기술 하나를 얘기했다. 두 번째 중요한 보안 측면은 암호화이다. 정보가 인증될 때, 그 정보는 그 자체로 명백하다. 디지털 서명은 단지 그 정보가 변경되지 않았다는 것을 증명한다. 대조적으로, 정보가 암호화되어졌을 때 그것은 맞는 키를 가지고 복호화되어질 수 있다.

인증은 코드 사이닝을 위해 충분하다. ---- 그 코드를 숨길 필요는 없다. 그러나 암호화는 애플릿 또는 응용 프로그램이 크레디트 카드 코드 또는 다른 개인정인 데이터와 같은 비밀 정보를 전송할 필요가 있을 때 필요하다.

그러나, 자바 2 표준 판은 현재는 암호화를 지원하지 않는다. 만약 여러분이 암호화를 해야 한다면, 여러분은 Java Cryptographic Extensions(JSE)을 얻을 필요가 이다. JCE는 DES(Data Encryption Standard)와 같은 단지 몇 가지의 암호화 알고리즘은 지원한다. 이것은 솔직히 56비트의 키 길이를 가지는 사용하기 편리한 대칭키이다. 만약 여러분이 더욱 강력한 알고리즘을 원한다면 여러분은 RSA Security Inc와 같은 암호 해독 서비스 제공자로부터 그들을 허가할 필요가 있을 것이다. 불행히도, Java Cryptographic Extensions는 단지 미국과 캐나다내에서만 이용할 수 있고 해독 알고리즘은 엄청난 허가 요금을 지불을 요구할 수 있다.

자바 표준 라이브러리에서 지원하는 암호화의 부족에 대한 중요한 이유는 미국정부에 의한 수출제약 때문이다. 미국에서는 수출허가 없이 암호 해독 제품을 수출하는 것은 범죄이다. 최근까지 수출 허가는 비교적 쉽게 깨어질 수 있는 낮은 등급의 보안에 대해서만 승인되어 졌다. 은행업무와 상업 응용프로그램을 위한 높은 등급의 키를 위한 수출 허가의 수는 천천히 향상되고 있다. 한번은 법률 집행 기관들을 위한 백 도어를 가지지 않는 암호화를 사용한 어떤 미국인을 불법으로 만들려는 법률을 통과시키려는 백악관의 노력이 있었다. 우리 견해로는 이러한 노력들은 결국 쓸모없고 역효과를 내는 것이다. 법률 집행을 위한 트랩도어가 없고 매우 높은 보안 등급을 가지는 암호화 소프트웨어는 국경이 없는 인터넷을 통해서 널리 이용가능 하다.

더욱이, RSA알고리즘(상당히 안전한 공개/비밀키를 사용하는 가장 일반적으로 사용되어지는 암호화 알고리즘)은 특허에 의해 보호되어져 왔다. 만약 여러분이 RSA알고리즘을 사용하고 싶다면, 당싱은 RSAsecurity inc로부터 면허가 필요하다. 그리고 여러분은 아마도 그들에게 상당한 요금을 지불해야 할것이다. 이러한 이유 때문에 Java Cryptography Extensions은 RSA알고리즘을 포함하지 않는다. 이러한 상황은 곧 변할것이다. RSA특허는 그 알고리즘이 공개 도메인에 넘어가고 자유롭게 구현되어질 수 있는 2000년 10월에 기간이 끝난다.

우리는 곧 강력한 암호화가 자바 표준 라이브러리의 일부분이 되기를 바란다. 그러므로 개발자들은 수출과 특허 제한을 겪지 않고 코드를 작성할 수 있을 것이다. 한편, 만약 여러분이 미국이나 또는 캐나다에 거주한다면 여러분은 http://java.sun.com/products/jce로부터 JCE를 다운로드할 수 있을 것이다. 선택적으로, 여러분은 웹 페이지 http://java.sun.com/propucts/jce/jce12_providers.html 에서 무균실 도구를 위한 소스들을 찾을 수 있을 것이다.
