Security in Java Platform

L' Architettura di sicurezza di Java
Parte I

Università degli Studi di Salerno, DIA

Parte I: Contenuti

- Architettura di sicurezza di base in Java
- I modelli di sicurezza in JDK 1.0, 1.1
  - Il modello sandbox
- L' architettura di sicurezza in JDK 2
  - Meccanismi per il controllo dell'accesso

Perchè Sicurezza in Java ?

- JDK è una piattaforma sulla quale sviluppare ed eseguire applicazioni in maniera "sicura"
  - programmare mobile code
- Strumenti e servizi già sviluppati in Java, con librerie di classi e API per applicazioni crittografiche

A chi interessa la sicurezza in Java?

- **Utenti Web**: browsers sono di solito abilitati ad eseguire applet
- **Programmatori**: scrivere codice robusto e "sicuro"
- **Amministratori di sistema**: proteggersi da codice mobile dannoso

Panoramica su Java

- L' ambiente di sviluppo di Java comprende:
  - Un linguaggio di programmazione che viene compilato in un formato indipendente dall'architettura (*byte code*)
  - La Java Virtual Machine che esegue il byte code
  - Un ambiente di esecuzione che lancia la JVM e fornisce le classi di sistema

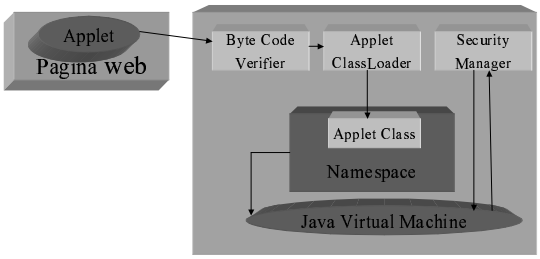
Panoramica su Java

- Applet vs Application:

```
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello world!");}}

import java.awt.Graphics;
public class Hello extends Applet {
  public void init() { resize(150, 25); }
  public void paint(Graphics g) {
    g.drawstring("Hello world!");  }}
```

Il ciclo di vita di un applet

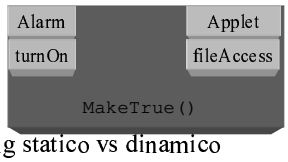


L'architettura base di sicurezza

- La sicurezza di base e' garantita attraverso :
  - una chiara progettazione object-oriented;
  - type safety;
  - risoluzione automatica dei compiti "difficili":
    - gestione automatica della memoria;
    - controllo del range per stringhe ed array;
    - garbage collection
    - gestione delle eccezioni

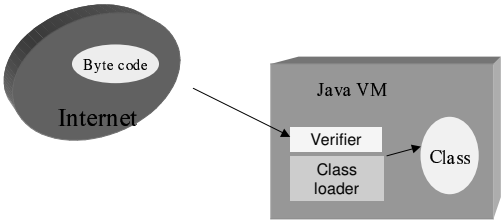
Type safety ↔ Sicurezza

- I programmi non possono effettuare delle operazioni dannose sugli oggetti.
- Esempio:



- Type checking statico vs dinamico

Loading external bytecode

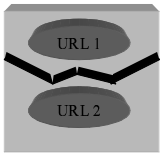


Verifica del Bytecode

- Verifica del formato e della struttura del byte code
- Un theorem prover assicura che:
  - Non si creino falsi puntatori
  - Non si violino restrizioni di accesso
  - Si acceda ad oggetti del tipo corretto
  - Non ci siano stack overflows
  - Sia corretto il numero e il tipo dei parametri nelle chiamate di metodi

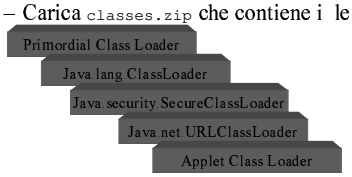
Dynamic Class Loading

- I Class loader determinano il modo in cui nuove classi vengono aggiunte al runtime :
  - trovare e caricare il byte code
  - definiscono *namespaces separati*
- Class loader possono essere ridefiniti dall'utente



Dynamic Class Loading

- Il primordial class loader ha il compito di fare il bootstrap del sistema
  - usa i meccanismi di accesso ai file forniti dal s.o.
  - Carica `classes.zip` che contiene i le Java API



Loading Classes

- Algoritmo per il classloading:
  - Determina se la classe è gia caricata
  - Consulta il Primordial CL per vedre se la classe può essere caricta dal classpath
  - Controlla che il CL abbia i permessi (SM)
  - Costruisci un oggetto Classe dall'array di byte

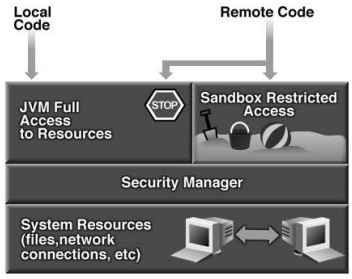
```
Class c = defineClass(name, buf, offset, len,
domain, signers);
```

  - Risolvi le classi refernziate e verifica il bytecode

Il Security Manager

- Il Security Manager controlla l'accesso ad operazioni potenzialmente dannose;
- Prima di consentire l'accesso a tali operazioni le Java API consultano il Security Manager:
  - se il codice non e' trusted viene lanciata una *security exception*
  - altrimenti l'operazione viene eseguita

Il modello di Sicurezza in JDK 1.0



La sandbox

- Gli applet non possono:
  - accedere a file locali;
  - aprire connessioni se non all'host di origine;
  - accedere o cambiare le proprietà di sistema
  - lanciare programmi in locale
  - creare od accedere a thread di altri gruppi

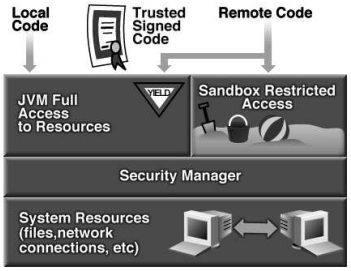
Regole implementate nei Browser

- Browser ridefiniscono propri Classloader e Security Manager
- Applets:
  - non possono accedere ai file locali;
  - aprire connessioni se non all'host di origine;
  - possono leggere solo 9 proprietà di sistema (VM version..)
  - gli applet caricati con `file:` fuori dal `CLASSPATH` usano l' Applet Class Loader

Caratteristiche del modello di sicurezza di JDK 1.0

- La Sandbox protegge l'accesso a tutte le risorse del sistema;
- I programmatori di applicazioni (non di applet) possono ridefinire un nuovo SecurityManager per "uscire" dalla sandbox

Il modello di sicurezza di JDK 1.1



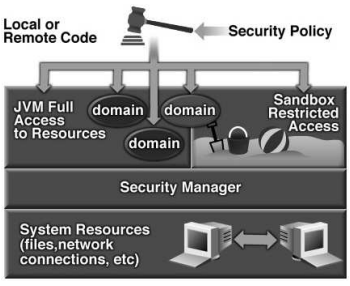
Caratteristiche del modello di sicurezza di JDK 1.1

- La firma del codice puo' essere usata per consentire maggiori privilegi agli applet
  - Diversi livelli di sicurezza possono essere realizzati quando viene eseguito codice remoto
- Autenticazione

Limiti del modello di sicurezza di JDK 1.1

- La politica di sicurezza per gli applet firmati e' binaria (o tutto o niente)
- Applicazioni eseguite localmente sono lanciate fuori dalla sandbox, senza possibilità di controllo
- Il codice che si trova sul `CLASSPATH` è trusted

Il modello di sicurezza di JDK 1.2



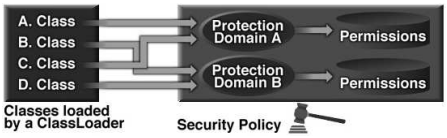
Caratteristiche del modello di sicurezza di JDK 1.2

- API per la sicurezza ed ambiente comune per sviluppatori di applicazioni e applet
- Controllo di accesso fine-grained
- Meccanismi di controllo ben definiti
- Nuovo Security Manager

Il modello di sicurezza di JDK 1.2

Ogni classe appartiene ad un singolo ProtectionDomain che consiste in:

- CodeSource (chi/dove)
- Permissions garantiti (cosa)



Identità del codice

- Ogni pezzo di codice ha una origine ed una firma che ne definisce l'identità:
- Chi ha firmato il codice
  - "The JavaSoft Division Security Group"
  - "Rossi, Paolo"
- Da dove proviene il codice
  - file:/home/paolo/classes/
  - http://java.sun.com/security/util.jar

Politiche di sicurezza

- Il comportamento di JRE e' specificato dalla politica di sicurezza adottata:
  - Matrice di controllo di accesso che assegna permessi al codice in esecuzione
  - Internamente la politica implementata e' rappresentata da un oggetto che il SM puo' consultare, instanziato dalla class `java.security.Policy`
  - Esternamente la politica e' rappresentata da un file ASCII (`.java.policy`)

Il file java.policy

```
keystore "keystoreFile";
        grant [codebase "<URL>"]
              [signedBy "<alias>"]
        {
          permission [permission class]
                     ["target"],
                     ["actions"];
          permission …
        };
grant CodeBase "http://www.dia.unisa.it/",
SignedBy "*"
{ permission java.io.FilePermission
                "read/write", "/tmp";
permission java.net.SocketPermission "connect",
"*.unisa.it";}
```

Permessi e Policy

- Per conoscere quali permessi sono garantiti al codice, viene consultato l'oggetto Policy:

```
Permissions permissions=
 Policy.getPolicy().getPermissions(codesource);
```

Permessi

- Un permesso viene usato per garantire accesso a risorse di sistema (files, sockets, etc.)
- Di solito i permessi vengono accordati su:
  - un target ("/home/schemers/readme")
  - una azione ("read,write")

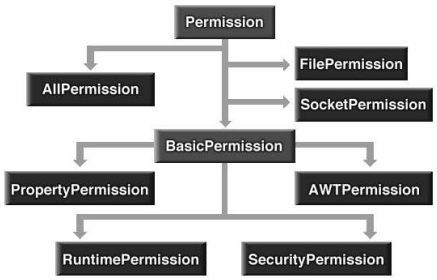
```
p= new SocketPermission
       ("www.unisa.it:-1023",connect"):
```

Permessi

- Tutti i permessi devono implementare il metodo `implies` method.
  "a `implies` b" significa che se "a" ha un permesso, allora anche "b" puo' esercitarlo

```
Permission p1 =
        new FilePermission("/tmp/*","read");
Permission p2 =
   new FilePermission("/tmp/readme","read");
p1.implies(p2) == true
p2.implies(p1) == false
```

Gerarchia degli oggetti Permission



Costruzione di un ProtectionDomain

- I domini vengono costruiti a partire dal codice e dall'insieme di permessi che si vogliono accordare

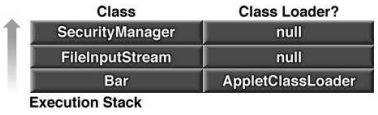
```
ProtectionDomain domain =
  new ProtectionDomain(codesource,permissions);
```

Assegnare un ProtectionDomain

- Costruire l'oggetto CodeSource
- Ottenere le Permissions dall'oggetto Policy
- Creare un ProtectionDomain
- Usare `SecureClassLoader.defineClass` per definire la classe

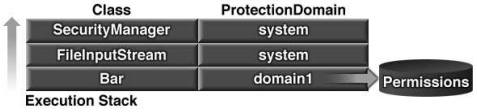
JDK 1.1 Controllo di Accesso

- La classe `Bar` chiama il metodo `FilesInputStream` che a sua volta richiama il Security Manager
- Il SecurityManager controlla se un classloader è nella catena delle chiamate.



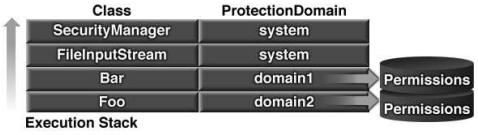
JDK1.2 Controllo di Accesso

- L' AccessController controlla che tutti i domini nella catena di chiamate abbiano i giusti permessi



Controllo di accesso con più domini

- Quando più domini sono nella catena di chiamate, tutti devono avere tutti i permessi



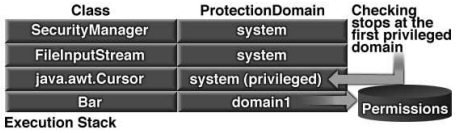
Blocchi Privileged

- Il metodo `doPrivileged`() della classe AccessController permette di ignorare i precedenti chiamanti:
-

```
  void changePasswd() {
          //  ...normal code here
AccessController.doPrivileged
    (new PrivilegedAction() {
          public Object run() {
          // Open file for read/write
                ….return null; }}); }
```

Controllo di Accesso con un blocco Priviliged

- La classe Bar può accedere alla lettura del file, sebbene non abbia i permessi poiché ha un blocco privileged



Algoritmo di Controllo Accesso

-

```
void checkPermission(Permission p) {
  foreach (caller) {
     if (the caller doesn't have permission)
      throw new AccessControlException(p);
if (caller is marked as privileged)
    return;
          }
// Access Granted
              return;
           }
```

SecurityManager e Access Controller

- `java.lang.SecurityManager` non è astratta
- I metodi di JDK invocano `AccessController`
- Per esempio, il metodo `checkRead` invocherà `checkPermission` sul `AccessController` per controllare se è stato garantito un `File Permission`

Uso dell' Access Controller

- I controlli richiamano il Security Manager:

```
SecurityManager sm=System.getSecurityManager();
if (sm!=null) {
    sm.checkread("/tmp")}
```

- In Java 2 si usa Access Controller:

```
FilePermission p =
    new FilePermission("/tmp","read");
AccessController.checkPermission(p);
```

Bibliografia

- G. Mc Graw, E. W. Felten
  "Securing Java"
  Wiley & Sons
- L. Gong
  "Inside Java 2 Platform Security"
  Addison Wesley

Security in Java Platform

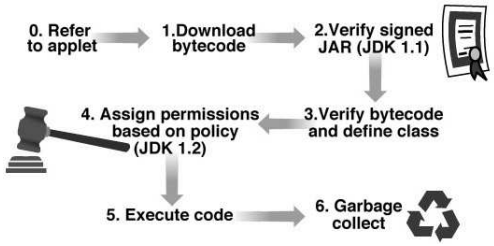
L' Architettura di sicurezza di Java
Parte II

Università degli Studi di Salerno, DIA

Parte II: Contenuti

- Attacchi alla sicurezza e applet in Java:
  - Categorie di attacchi
  - Applet dannosi
- Le API per le applicazioni crittografiche
  - tool e servizi implementati in Java
- Gestione della sicurezza nei browser

Il ciclo di vita di un applet



Il ciclo di vita di un Applet

- Dopo che l'applet e' caricato nella JVM, il browser chiama i seguenti metodi:
  - init() inizializza l'applet e legge i parametri contenuti nel tag
  - start() avvia l'applet quando il browser si posiziona sulla pagina
  - stop() ferma l'esecuzione quando il browser lascia la pagina
  - destroy() rilascia completamente le risorse

Categorie di attacchi

Categorie	***Difese di JDK***
• Modifiche al sistema	Strong
• Invasione della Privacy	Strong
• Denial of Service	Weak
• Antagonismo	Weak

Malicious Applets

- Applet sul Web:
  - Falsificazione di e-mail
  - Furto di cicli di CPU per eseguire altri lavori
  - Crash del sistema per impiego di tutte le risorse
  - Infastidire l'utente con suoni ed immagini

Annoying applets: `NoisyApplet`

```
•  public void init() {
 bark = getAudioClip(getCodebase(), "bark.au"); }
public void start() {//when you enter the page
if noisethread == null) {
    noisethread = new Thread(this);
    noisethread.start(); }
public void stop() {//when you exit
  if (noisethread != null) {
     if (bark != null) bark.stop();
     noisethread.stop();
     noisethread = null;
}
public void run() {
  if (bark != null) bark.loop();
}
```

Denial of Service Applets

- Si crea un applet con massima priorità
- Si ridefinisce il metodo `stop()` come null
- Si fa qualcosa di inoffensivo
- Cicli di sleep
- Si calcola qualcosa in un ciclo infinito
  - Assassin Applet

Contraffazione di mail

- Gli applet si connettono alla porta 25 dove il demone SMTP e' in ascolto
- SMTP marca la mail con l'IP della macchina che effettua la connessione
- Per SMTP la mail proviene dall'host dell'utente che visita la pagina

Altri Malicious Applets online

- Manda in crash il browser consumando le risorse della CPU
- Mostra centinaia di finestre nere
- Mostra false finestre di dialogo
- Utilizza tutto lo spazio su disco allocato al browser
- URL:
  http:\\www.rstcorp.com/hostile-applets

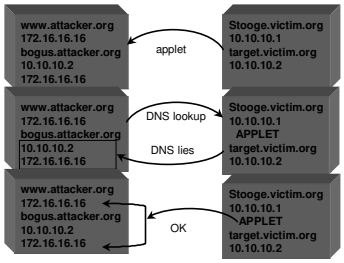
Attack Applets

- Aprono il sistema ad accesso non autorizzato:
  - modifica/rivelazione dati locali, virus, trapdoor
- Quando Java fu rilasciato si pensava che fosse completamente sicuro:
  - 16 bug importanti sono stati scoperti nelle diverse implementazioni di Java

Attack Applets: DNS spoofing

- Gli applet possono connettersi solo al sito da cui sono stati scaricati:
  - DNS traduce i nomi in una lista di IP
  - DNS traduce le richieste di connessione in indirizzi IP
  - la connessione e' autorizzata se ogni macchina e' in entrambe le liste

Attack Applets: DNS spoofing

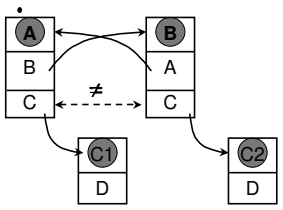


Attack Applets: Dots and Slashes

- Classloading dinamico :
  - dal Web server
  - dal disco locale
- Prima di JDK 1.1 il codice locale era trusted
- Punti nei nomi delle classi sono tradotti in slash:
  - `mycomp.mydirectory.myclass`
    `mycomp\mydirectory\myclass`
- In JDK 1.01 & Navigator 2.01, i nomi delle classi possono iniziare con "\"

Attack Applets: Type Spoofing

- I riferimenti ad altre classi sono risolti
- Vengono creati diversi namespace
- Malicious class loaders possono provocare confusino sui tipi



Attack Applets: Magic Coat

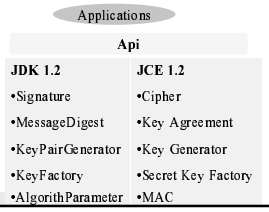
- Un bug nella classe Class permetteva ad applet ostili di cambiare la vista del sistema di chi aveva firmato il codice
- Il metodo `getSigners` restituisce un array di soggetti che possono essere modificati
- L'applet puo' modificare la sua firma in modo da essere firmato da un soggetto trusted

Attacchi Recenti

- Aprile 1999:
  - Un bug in JVM of JDK 2 and Netscape 4.x, evita i controlli della VM
- Agosto 1999:
  - Errore di programmazione in una parte critica per la sicurezza delle librerie Microsoft, violazione di sicurezza (patched)
- Ottobre 1999:
  - Bug nel verifier della Microsoft porta a type confusion

Programming Cryptoghraphy

- Java Cryptoghraphy Architecture (JCA)
  - crypto API in JDK 2
  - Java Cryptoghraphy Extension (JCE)
    - encryption, key exchange, Mac, etc.



Progettazione della JCA

- Indipendenza dall'algoritmo usato ed estensibilità
  - *service classes* forniscono le funzionalità
- Indipendenza dall'implementazione ed interoperabilità
  - architettura *provider-based*
  - Criptography Service Provider (CSP) implementa uno o più servizi crittografici della JCA

Cryptography Service Provider

- Oltre a Dsa, MD, keygen, CSP contiene anche:
  - gestione di key factories and keystore
  - gestione di algorithm parameter
  - gestione di certificate factories
- Oltre al default CSP altri provider possono essere installati staticamente o dynamicamente

Installazione di un Provider

- Posizionare il file JAR che contiene le classi sul CLASSPATH
- Aggiungerlo alla lista dei provider installati:
  - staticamente:
    - aggiorna il security properties file `security.provider.n=Unisa.provider.Master`
  - dinamicamente:
    - richiama il metodo `addProvider` nella classe `Security`

Service Classes

- A *service class* definisce un servizio crittografico in modo astratto;
- L' interfaccia è implementata in forma di SPI (Service Provider Interface)
- Esempio: `Signature (MessageDigest)`
  - fornisce accesso a DSA
  - l'implementazione in SPI è relativa ad un particolare tipo di algoritmo (SHA1withDSA, etc)

Java.security.Security

- La classe `Security` gestisce i provider installati e le proprietà di sicurezza
  - Provider
  - MessageDigest
  - Signature
  - AlgorithParameter
  - Key
  - KeyFactory
  - CertificateFactory

Example 1: Message Digest

- Suppose a message is composed by three byte arrays: i1,i2,i3
  - create a message digest instance
    - public static MessageDigest sha= MessageDigest.getInstance("SHA");
  - supply the data to the message digest object:
    - sha.update(i1); sha.update(i2); sha.update(i3);
  - compute the digest:
    - byte[] hash = sha.digest();

Example 2: Key Pair Generation

- Calculate keys with 1024 bit:
  - get a keygen object
    - `KeyPairGenerator keyGen =KeyPairGenerator.getInstance("DSA")`
  - get a random seed:
    - `SecureRandom rand= Securerandom.getInstance("SHA1PRNG");`
    - `keyGen.initialize(1024,random);`
  - generate the key pair
    - `keyPair pair = keyGen.generateKeyPair();`

Example 3: Signature

- Create a Signature object
  - `Signature dsa = Signature.getInstance("SHA1withDSA")`
  - `PrivateKey priv = pair.getPrivate();`
  - `dsa.initSign(priv);`
- Sign the data
  - `dsa.update(data);`
  - `byte[] signa = dsa.sign();`
- Verify
  - `dsa.initVerify(pub); dsa.update(data);`
  - `boolean verify = dsa.verify(sig);`

Example 3: Signature

- Specificando solo i parametri della chiave:
  - `DSAPrivateKeySpec dsap = new DSAPrivateKeySpec (x,p,q,g);`
  - `PrivateKey priv= keyFactory.generatePrivate(dsap);`
  - `Signature s=Signature.getInstance("SHAwithDSA")`
  - `s.initSign(priv);`
  - `s.update(somedata);`
  - `byte[] signature=sig.sign():`

Tools: keystore

- Il keystore e' un database protetto per memorizzare chiavi e certificati
  - E' implementato tramite un file (`.keystore`)
  - E' protetto da password
    - ogni entry è protetta ed e' associata a degli alias
  - La corrispondente classe fornisce metodi per accedere al database tramite una SPI

Tools: keytool

- Utilità per creare coppie di chiavi e certificati firmati
- usa gli algoritmi forniti dal CSP
- crea e gestisce entry nel keystore
  - import ed export di certificati
  - genera richieste di certificazioni per la CA

Tools: policytool

- Utilità per creare e modificare file policy
  - interfaccia grafica per la creazione di policy entry
  - permette di
    - aggiungere o revocare permissions
      - specificando il *tipo* e il *target*
    - specificare azioni
    - specificare le firme autorizzate

Tools: jarsigner

- Utilità per firmare e verificare file JAR
- utilizza le informazioni contenute nel keystore
- usa algoritmi SHA e DSA o MD5 e RSA
- il file JAR generato ha due file aggiuntivi:
  - un file `manifest` con estensione `.sf`
    - per ogni sorgente in Jar lista lista il nome del file., il nome dell'algoritmo usato e il valore digest
  - un file di firma con estensione `.dsa`

Verifica con jarsigner

- `Jarsigner –verify example.jar`
  - verifica la firma dello stesso file sf
  - verifica ogni entry del file sf con la corrispondente del file manifest
  - calcola il digest per ogni file che ha una entry in sf e verifica i valori

Sicurezza & browser: Netscape

- Object Signing Tool
- Privileges sono chiamati capabilities e sono contenuti nel `capsapi_classes.zip` file
- La classe `Privilege Manager` gestisce le richieste per assegnare o revocare i privilegi (FileAccess,SendMail,Exit,Exec,Registry)
- Se un privilegio viene assegnato, esso dura per tutta la vita dell'applet

Sicurezza & browser: MS Explorer

- Il sistema divide i siti Web in quattro zone di sicurezza con diversi livelli:
  - Local, Trusted, Internet, Restricted
  - Security levels: High, Medium, Low, Custom
- Usa MS Authenticode certificates
- Usa SDK e lavora solo con CAB file
- Bisogna specificare il cab file nel tag html

Sicurezza & browser: JDK plugin

- Il Java plugin esegue gli applets invece della VM del browser
- Converte i tag HTML per gli applets
- Use l'ambiente JDK installato

Bibliografia

- G. Mc Graw, E. W. Felten
  "Securing Java"
  Wiley & Sons
- L. Gong
  "Inside Java 2 Platform Security"
  Addison Wesley