Java Security Model e RMI

- Da Java 2 in poi la politica di sicurezza di Java impone all'utente di definire espressamente i *permessi* di cui deve disporre un'applicazione per poter essere eseguita.

- Tale insieme di permessi definiscono una *sandbox*, cioè uno spazio virtuale in cui eseguire l'applicazione.

- Per definizione la sandbox contiene il minor numero di permessi che consentano all'applicazione di essere eseguita.

- Questi permessi sono di solito elencati nei file di *policy*.

- Tali files di policy vengono creati da amministratori ed utenti con funzioni diverse:

  – chi sviluppa un'applicazione la distribuisce con un file di policy che elenca i permessi di cui l'applicazione ha bisogno per poter essere eseguita.

  - Chi amministra un sistema, utilizza invece i file di policy per definire cosa possono fare i programmi lanciati all'interno del sistema.

- Una caratteristica di tali permessi è che non sono concessi ad un'applicazione nel suo complesso ma bensì alle classi che compongono l'applicazione. La ragione di ciò è dovuta al caricamento dinamico di codice. In tal caso infatti, codici differenti, provenienti da coebasi differenti, possono avere permessi differenti.

Tipi di permessi

In Java 2 esistono nove tipi basilari di permessi:

1. AWT permissions
2. File permissions
3. Network permissions
4. Socket permissions

5. Property permissions
6. Reflection permissions
7. Runtime permissions
8. Security permissions
9. Serializable permissions

- Ciascuno di questi permessi rappresenta un modo differente in cui la sandbox può essere attaccata dall'esterno.
- Comunque, in una tipica applicazione RMI i permessi più importanti sono 1, 2, 4 e 5. Vediamoli un po' più in dettaglio.
- **AWT permissions**

  Questi permessi controllano l'accesso a risorse collegate con lo schermo e fanno fronte a tre possibili forme di attacchi:

– **Raggirare l'utente**. Del codice ostile potrebbe pretendere di instaurare un dialogo con l'utente sotto forma di registrazione a qualche servizio. In questo modo l'attacker potrebbe venire in possesso di informazioni riservate.

– **Cattura di informazioni statiche**. L'attacker potrebbe catturare l'informazione presente sullo schermo. Ad esempio, del codice ostile, caricato attraverso un mailer, potrebbe creare delle immagini GIF di ciò che appare sullo schermo dell'utente e spedirle all'attacker.

– **Raggirare l'applicazione**. Java 2 include un meccanismo chiamato robot che simula l'interazione di un utente con una applicazione. Ebbene, l'utente potrebbe lanciare un'applicazione e vedere sul suo schermo l'esecuzione di un'applicazione, del tutto simile alla sua, ma controllata da un robot gestito dal codice ostile.

Essendovi una varietà di possibili attacchi esistono 6 sottotipi di permessi: accessClipBoard, accessEventQueue, listenToAllAwtEvents, readDisplayPixels, showWindowWithoutWarningBanner, e createRobot.

Sotto vediamo un esempio in cui si consente a tutte le classi di mostrare finestre senza “warning banners”:

```
grant {
   permission java.awt.AWTPermission
              ``showWindowWithoutWarningBanner'';
};
```

- **File permission**

  Questi permessi abilitano le operazioni di *lettura*, *scrittura*, *esecuzione*, e *cancellazione* di files. Vi sono perciò quattro tipi di file permission: `read`, `write`, `execute`, `delete`. Ad esempio, vediamo un permesso che consenta a *tutte le classi* di leggere, scrivere, cancellare, ma non eseguire, tutti i files nella directory `C:/temp/` ed in tutte le sue sottodirectory:

```
grant {
   permission java.io.FilePermission ``C:/temp/-'' ,
                ``read, write, delete'';
}
```

Se avessi scritto invece:

```
grant {
   permission java.io.FilePermission ``C:/temp/*'' ,
                ``read, write, delete'';
}
```

allora i permessi sarebbero stati validi solo per i files all'interno della directory temp ma non all'interno delle sue sottodirectory.

- **Socket permission**.

  Tali permessi riguardano le operazioni su sockets. Esistono quattro operazioni basilari su sockets:

- *Risoluzione di un indirizzo*. Cioè la traduzione di un indirizzo leggibile in un indirizzo IP. Ad esempio, `www.oreilly.com` viene tradotto in 209.204.146.22.
- *Connessione*. Cioè la la creazione di una connessione socket da parte di un client verso un server.
- *Restare in ascolto di una connessione*. Ovvero quando un server socket ascolta per connessioni.
- *Accettare una connessione* Quello che accade dopo l'ascolto, cioè quando un istanza di `ServerSocket` crea una connessione attraverso un'istanza di `Socket`.

Per ciascuna di queste quattro operazioni esiste una socket permission: `resolve`, `connect`, `listen`, `accept`.

- `resolve` è in realtà implicato dagli altri permessi e si usa veramente poco.
- `connect` è richiesto da un client RMI per specificare host, domini, e porte a cui si può connettere.

- `listen` è richiesto dalle applicazioni che esportano un server RMI essenzialmente per indicare la porta su cui può ascoltare il Serversocket associato al server.
- `accept` è richiesto da un server RMI per specificare gli indirizzi URL dei client da cui può accettare connessioni e su quale porta. Più precisamente, dopo che una connessione è stata accettata da un server, ma prima che venga creata un'istanza di `Socket`, viene fatto un controllo per vedere se chi ha invocato `ServerSocket.accept` ha il permesso `accept` di accettare la connessione col client remoto.

Le socket permission richiedono di specificare un insieme di indirizzi di computers da cui si accetta una connessione ed un insieme di porte su cui si è disposti ad accettare la connessione. Gli indirizzi degli host possono essere forniti in uno dei seguenti modi:

- *Hostname.* Ad esempio `www.hipbone.com`.
- *Indirizzo IP.* Ad esempio 209.220.141.161.

– `localHost`. Ovvero la stringa `localHost` che specifica la macchina locale. Se non viene indicato nulla si utilizza il valore di default della variabile `localHost`.
– **.partial_domain_specification*. È possibile utilizzare * come una wildcard sul lato sinistro di un hostname. Ad esempio, `*.hipbone.com` e `*.com` sono specifiche valide. L'asterisco non può comunque essere usato come wildcard per indirizzi IP.

Le porte possono essere indicate o fornendo il numero specifico della porta oppure indicando una serie di porte:

– -2048: denota tutte le porte $\leq$ di 2048.
– 2048-: denota tutte le porte $\geq$ di 2048.
– 2048-4096: denote tutte le porte comprese tra 2048 e 4096.

Riportiamo un esempio in cui si consente a codice proveniente da un qualunque codebase di accettare connessioni da parte di qualunque macchina all'interno del dominio `*.oreilly.com` e su qualunque porta maggiore o uguale di 1024.

```
grant {
  permission java.net.SocketPermission
      ‘‘*.oreilly.com:1024-’’, ‘‘accept’’;
}
```

- **Property permission**
  - La classe `System` possiede un metodo `getProperties()` che ritorna le proprietà del sistema.
  - Il risultato dell'invocazione di tale metodo è un'istanza di `java.util.Properties` che contiene due cose: un insieme di coppie (nome, valore), contenente informazioni sulla JVM, e tutte le proprietà settate da comando con il flag -D.
  - Per default una classe proveniente da un'altra JVM non può leggere o modificare le proprietà di sistema. Comunque è possibile fornire tali permessi.
  - Le proprietà vengono specificate in uno dei seguenti tre modi: (i) con '*' per indicare tutte le proprietà, (ii) fornendo il nome

della proprietà, (iii) con una stringa seguita da '.*'. Ad esempio `groupname.*` individua tutte le proprietà che iniziano con `groupname`.

– Riportiamo un esempio in cui si fornisce a tutte le classi di leggere, ma non cambiare, un certo numero di proprietà:

```
grant {
  permission java.util.PropertyPermission
                   ''Java.version'', ''read'';
  permission java.util.PropertyPermission
                   ''Java.vendor'', ''read'';
  permission java.util.PropertyPermission
                   ''Java.vendor.url'', ''read'';
  permission java.util.PropertyPermission
                   ''Java.class.version'', ''read'';
}
```

IMPORTANTE: Una descrizione dei permessi più usati da RMI può essere trovata a pagina 91 del libro di Pitt e McNiff.

Java Security Manager

- La garanzia che vengano rispettati i permessi all'interno di una JVM fissati nei file di policy è fornita installando un'istanza della classe `SecurityManager`.

- Quando un programma tenta di eseguire un'operazione che richiede un esplicito permesso (ad esempio una lettura su un file o una connessione via socket) viene interrogata l'istanza del `SecurityManager`.

- Se però non è stato installato un SecurityManager allora le applicazioni hanno tutti i permessi.

- È possibile installare un solo SecurityManager all'interno di una JVM. Qualsiasi tentativo di installare un altro SecurityManager (o settare il corrente a null) è esso stesso soggetto alla presenza di un permesso `RuntimePermission(``setSecurityManager'')`.

- Il modo più semplice di installare un security manager è definendo la proprietà di sistema da linea di comando:

```
java -Djava.security.manager applicazione
```

In tal modo verrà creata ed installata un'istanza del SecurityManager prima che venga eseguita l'applicazione.

- È possibile anche specificare un particolare SecurityManager come in:

```
java -Djava.security.manager=java.rmi.RMISecurityManager
     applicazione
```

- `RMISecurityManager` è semplicemente una sottoclasse di `SecurityManager` che non aggiunge particolari funzionalità. Più che altro serve a ricordare al programmatore che sta usando un SecurityManager in ambito RMI.

- Un'altra possibilità è quella di lanciare il SecurityManager all'interno dell'applicazione (come abbiamo visto nei nostri esercizi) con il seguente comando:

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

Ma come funziona un SecurityManager?

- Ogni qual volta si esegue un'istruzione il SecurityManager invoca un metodo di `check` appropriato (ad esempio: `checkRead()`, `checkPropertiesAccess()`, `checkConnect()`, etc.) per controllare se esiste il relativo permesso nei file di policy.

- L'implementazione di questi metodi causa un'invocazione del SecurityManager. Tale invocazione può lanciare una `SecurityException` nel caso in cui si tenti di eseguire un'operazione per cui non si ha il permesso opportuno, altrimenti l'applicazione procede nell'esecuzione.

- Più precisamente quando si invoca uno dei metodi `check`, il SecurityManager controlla che *ogni classe* nello stack abbia il permesso appropriato. Controllare tutte le classi nello stack è l'unico modo per prevenire un attacco indiretto al sistema.

- Si noti che `SecurityException` è un'eccezione a runtime e quindi il compilatore non impone che venga catturata. È comunque cura del programmatore di catturare tale eccezione nei punti opportuni, come ad esempio la procedura main, o i metodi init, start e stop di un applet.

I tre file di policy

In una architettura Java esistono tre file di policy che vengono controllati dal Java Security Manager rispettando il seguente ordine:

1. Il "global policy file" che copre tutte le applicazioni eseguite da qualsiasi utente di un determinato host. Viene settato da chi configura il sistema e contiene permessi per tutte le JVM. In un sistema linux prende il nome *java.policy* e si trova in una directory del tipo:

   ```
   /usr/java/j2sdk1.4.0/jre/lib/security
   ```

   Guardate il vostro file di policy globale. Troverete `PropertyPermission` per le JVM ed il seguente SocketPermission:

   ```
   permission java.net.SocketPermission localhost:1024-,
                                 listen
   ```

necessario per esportare servers: consente a chiunque di ascoltare (ma non di accettare) su le porte maggiori o uguali alla 1024.

2. L' "user-specific policy file" che riguarda tutte le applicazioni lanciate da un utente. Sta dentro la vostra home e prende il nome *.java.policy* (con il punto iniziale). Controllate il vostro user-specific policy file. Se contiene

   ```
   permission java.security.AllPermission
   ```

   Cancellatelo!!! Altrimenti ogni vostra applicazione avrebbe tutti i permessi per default.

3. L' "application-specific policy file" cioè il file di policy specifico dell'applicazione che abbiamo visto nei nostri esempi.

Il debug di permessi: `java.security.debug`

- Il modo migliore per osservare il SecurityManager a lavoro è quello di settare la proprietà di sistema `java.security.debug`.

- Questa proprietà prende i seguenti quattro valori:
  - `all` Attiva tutte le opzioni di debugging.
  - `access` Visualizza una traccia a tutte le invocazioni del metodo `checkPermission()`. In tal modo è possibile vedere quali permessi il codice sta richiedendo, quali invocazioni vengono eseguite con successo e quali falliscono.
  - `policy` Visualizza informazioni sui files di policy: la loro locazione nel file system; i permessi che concedono; ed i certificati che usano per certificare il codice.

  Le seguenti opzioni possono essere usate insieme ad `access`, separate da una virgola:

  - `stack` Visualizza lo stack delle invocazioni, ogni volta che un permesso viene controllato.
  - `failure` Visualizza lo stack delle invocazioni, solo quando un permesso è negato.

– `domain` Visualizza il dominio di protezione relativo al permesso che si sta controllando.

Ad esempio:

```
-Djava.security.debug=access,failure
```

- Quindi settando la proprietà `java.security.debug` è possibile ottenere i permessi necessari all'esecuzione della vostra applicazione che dovete inserire nel file di policy.
- Per facilitare il compito di chi sviluppa un'applicazione, Java 2 supporta uno strumento, chiamato `policytool`, che consente di editare un file di policy dell'applicazione.

Struttura di un file di policy

- Un file di policy contiene uno o più *domini di protezione* a cui corrispondono comandi `grant` col seguente formato:

```
grant [signedBy Name] [codeBase URL] {
   // lista di permessi
};
```

- Un dominio di protezione consiste in un insieme di permessi concessi ad un codice sorgente.

- Un codice sorgente è individuato da un URL a cui è eventualmente associato un certificato digitale (tralasceremo i certificati).

- Un permesso invece è individuato da un *nome*, una *lista di azioni* (opzionale), un *target* (opzionale).

Vediamo adesso un esempio di file di policy:

```
grant {
  permission java.net.SocketPermission ‘‘*:80’’, ‘‘connect’’;
};

grant codeBase ‘‘http://www.supplierA.com/’’ {
  permission java.net.SocketPermission ‘‘*.univr.it:3000’’, ‘‘accept’’;
  permission java.io.FilePermission ‘‘/home/massimo/progetto/*’’, ‘‘read’’;
  permission java.util.PropertyPermission ‘‘*’’, ‘‘read’’;
};

grant codeBase ‘‘file://d:/classes’’ {
  permission java.awt.AWTPermission ‘‘showWindowWithoutWarningBanner’’;
  permission java.awt.AWTPermission ‘‘readDiplayPixels’’;
  permission java.net.SocketPermission ‘‘:1024-’’, ‘‘listen’’;
};
```

- Nel primo dominio di protezione si concede a tutte le classi, indipendentemente dalla loro provenienza, la possibilità di connettersi alla porta 80 (in cui in genere sta il webserver) di un qualunque host.

- Nel secondo si concede a tutte le classi provenienti dal dominio `http://www.supplierA.com/` i permessi per:
  - installare un server sulla porta 3000 accettando connessioni solo dai domini all'interno `*.univr.it`;
  - tutti i files in `/home/massimo/progetto/` ;
  - il permesso di leggere tutte le proprietà di sistema. Questa è una proprietà piuttosto azzardata che raramente si da.
- Il terzo pezzo concede a tutte le classi provenienti da una sottodirectory di `D:\classes` due permessi sullo schermo e la possibilità di ascoltare connessione su qualunque porta ≥ 1024.