

Sicurezza: Java Security

Henry Muccini
University of L'Aquila
e-mail: muccini@univaq.it



Indice:

- Introduzione
- Problemi con le applet
- Java features for security
- Soluzioni: dal JDK1.0 al JDK1.2
- Applet rights
- Implementazione della sicurezza nel JDK1.2
- Tools per la sicurezza in Java

Introduzione:

Il linguaggio Java, nato nel 1995, e' divenuto oggi uno dei linguaggi di programmazione piu' popolari. Cio' che lo ha reso tanto famoso e' l'*indipendenza dalla piattaforma Hw*; un programmatore puo' scrivere un programma in Java senza preoccuparsi, in generale, della piattaforma sulla quale esso dovra' girare. Un utilizzatore finale potra' utilizzare il codice interpretato utilizzando la Java Virtual Machine (JVM) appropriata per il suo sistema operativo. Questa indipendenza dalla piattaforma ha portato alla scrittura di codice Java che possa girare su qualsiasi piattaforma. In particolare, sono nate le Applet Java che sono codice Java che puo' risiedere su di un server e puo' essere scaricato ed eseguito su di una macchina client. Il collegamento tra macchina client e macchina server puo' essere realizzato tramite il Web. Quello che accade e' che collegandosi ad una pagina Web, delle applet Java possono essere scaricate ed eseguite sulla nostra macchina. Chi ci garantisce che questo codice non esegua operazioni illecite sulla nostra macchina? Quali sono le operazioni che un applet puo' eseguire sul client? Come ci si puo' proteggere da operazioni non volute?

Questi sono alcuni degli interrogativi che troveranno risposta nel seguito.

Problemi con le applet:

Un applet, una volta scaricata sul client, puo' eseguire delle operazioni non consentite. In particolare esistono 4 tipologie principali di attacco:

ATTACK CLASS	EXPLANATION AND CONSEQUENCES	JAVA DEFENSE
System Modification	The most severe class of attacks. Applets that implement such attacks are attack applets. Consequences of these attacks: severe.	Strong
Invasion of Privacy	If you value your privacy, this attack class may be particularly odious. They are implemented by malicious applets. Include mail forging. Consequences of these attacks: moderate.	Strong
Denial of Service	Also serious but not severely so, these attacks can bring a machine to a standstill. Also implemented by malicious applets. May require reboot. Consequences of these attacks: moderate.	Weak
Antagonism	Merely annoying, this attack class is the most commonly encountered. Implemented by malicious applets. May	Weak

	require restart of browser. Consequences of these attacks: light to moderate.	
--	--	--

Con il termine “malicious applet” indicheremo nel seguito un’applet scritta con l’obiettivo di attaccare il sistema locale eseguendo azioni non consentite. Per evitare che tali applet arrechino danni al sistema, Java prevede dei meccanismi di protezione. Inoltre i browsers permettono di disabilitare l’esecuzione di applet o controllare la loro provenienza e firma. Un’altra soluzione al problema e’ data da dei tools in grado di controllare che l’applet non arrechi danni al sistema, ma tali tools possono naturalmente essere basati solo su problemi noti.

Antagonism:

L’obiettivo di alcune malicious applet e’ quello di arrecare fastidio, ad esempio suonando continuamente dei pezzi o mostrando figure oscene. Un esempio di tale applet e’ basata su un’idea di Mark Fools e consiste nel mostrare a video un’immagine con un suono. Questo sembrerebbe un comportamento normale. Il problema si verifica nel momento in cui si passa ad un’altra pagina Web ed il suono continua a sentirsi.

Come e’ possibile! Semplice; l’applet ha lanciato un thread che non si ferma mai! Seguendo questo approccio e’ possibile creare delle applet che, ridefinendo il metodo stop(), eseguano un’azione all’infinito.

Per maggior dettagli consultare il testo “Securing Java”.

Denial of Service:

“Se non riesco ad entrare nel tuo computer, faccio in modo che neanche tu possa usarlo!!!”. Questa e’ la filosofia che e’ dietro il Denial of Service. Attacchi di questo tipo consumano quindi risorse portando il sistema a non essere piu’ utilizzabile. Tipicamente si consuma tutto il tempo di CPU, si alloca tutta la memoria disponibile, si occupa tutto lo spazio video o si porta il sistema ad attendere eventi che mai si verificheranno.

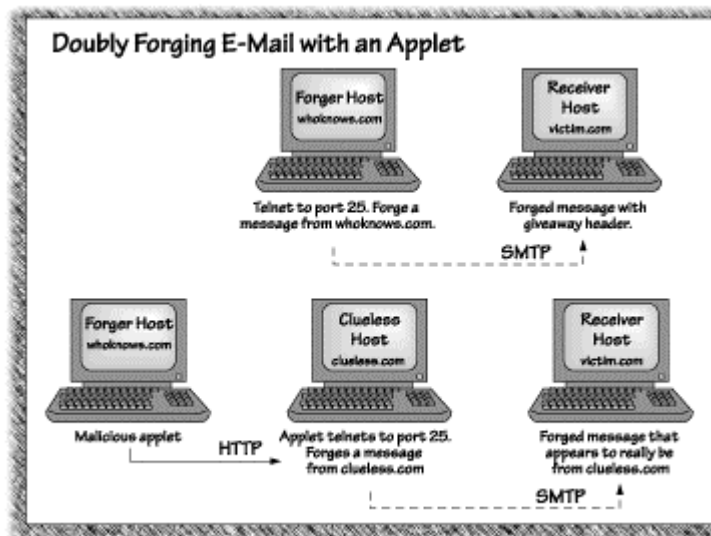
La caratteristica di questi attacchi e’ che sono molto semplici da realizzare e hanno un alta percentuale di successi. Partendo, ad esempio dall’attacco presentato in precedenza, risulta essere molto semplice applicarlo per creare un Denial of Service:

- 1) Si crea un applet che lancia un thread con massima priorita’;
- 2) Si ridefinisce il metodo stop();
- 3) Fa in modo che l’applet si presenti come innocua e tipica (visualizza una picture o delle immagini animate);
- 4) Fa’ partire il thread con un certo delay, in modo che possa esser piu’ difficile scovarlo;
- 5) Quando il thread parte, facciamo in modo che esegua calcoli all’infinito per occupare risorse (ad esempio la CPU)

Invasion of Privacy:

Un esempio tipico della violazione della privacy e’ dato dal *Forging Mail*; tale violazione consiste nel mandare e-mails spacciandosi per altre persone. Solitamente tale attacco viene diretto alla porta 25 di una macchina, al protocollo SMTP. Cio’ che accade e’ che comunque la mail falsificata puo’

essere riconosciuta analizzando alcune informazioni nell'header del messaggio; in particolare, nell'header viene visualizzato il mittente del messaggio e l'host dal quale viene spedito. Utilizzando un applet si puo' realizzare un *double forging* attach; con tale tipo di attacco si ha che la mail fasulla appare del tutto lecita!!! Questo accade perche' l'applet lavora sulla macchina di colui che subisce l'attacco.



System Modification:

Questa tipologia di attacchi viene portata tramite *attack applet*, cioe' un applet che sfrutti dei banchi del linguaggio o della virtual machine. I danni sono molto piu' pesanti rispetto a quelli delle malicious applet; infatti e' possibile modificare dati nell'hard disk, rivelare informazioni private a terze persone, infettare la macchina con virus, ... Applicando uno degli attacchi discussi un hacker puo' acquisire il pieno controllo della macchina.

DATE	ATTACK APPLLET
February 1996	Jumping the Firewall
March 1996	Slash and Burn
March 1996	Applets Running Wild
May 1996	Casting Caution to the Wind
June 1996	Tag-Team Applets
June 1996	You're Not My Type
July 1996	Casting Caution to the Wind (reprise)
August 1996	Big Attacks Come in Small Packages
February 1997	Steal This IP Number
February 1997	Cache Cramming
March 1997	Virtual Voodoo
April 1997	The Magic Coat
May 1997	Verifying the Verifier
July 1997	The Vacuum Bug
August 1997	Look Over There
July 1998	Beat the System

February 1996: Drew Dean, Edward Felten, and Dan Wallach at Princeton discovered a flaw in Java's networking software, affecting Netscape Navigator 2.0. This flaw was postulated independently by Steve Gibbons. It could be exploited to launch security attacks on other machines on a private network. This flaw was fixed in Netscape Navigator 2.01. The resulting attack is called Jumping the Firewall. See [page 147](#). This attack resulted in one of the two Java-related CERT alerts [CERT, 1996a].

March 1996: David Hopwood at Oxford University found a flaw that allows an attack that tricks Java into treating the attacker's applet as trusted code. This flaw allowed full system penetration. It affected Netscape Navigator 2.01 and was fixed in Netscape Navigator 2.02. The resulting attack is called Slash and Burn. See [page 153](#).

March 1996: The Princeton team (Dean, Felten, and Wallach) found a bug in the Java byte code Verifier and a flaw in the class-loading mechanism. Together, these allowed full system penetration. This problem affected Netscape Navigator 2.01 and was fixed in Netscape Navigator 2.02. The resulting attack is called Applets Running Wild. See [page 158](#). This attack resulted in the second of the two Java-related CERT alerts [CERT, 1996b].

May 1996: Independent consultant Tom Cargill, working with the Princeton team (Dirk Balfanz, Dean, Felten, and Wallach) found a flaw in the implementation of the Java interpreter. This flaw allowed full system penetration. It affected Netscape Navigator 2.02 and Microsoft Internet Explorer 3.0beta1, and was fixed in Navigator 3.0beta3 and Explorer 3.0beta2. The resulting attack is called Casting Caution to the Wind. See [page 163](#).

June 1996: Hopwood found another flaw in the interpreter that again allowed full system penetration. This flaw affected Netscape Navigator 3.0beta3 and was fixed in Navigator 3.0beta4. The resulting attack is called Tag-Team Applets. See [page 165](#).

June 1996: Balfanz, Dean, and Felten found a flaw in Java's implementation of array types that allowed full system penetration. This flaw affected Netscape Navigator 3.0beta5 and was fixed in Navigator 3.0beta6. The resulting attack is called You're Not my Type. See [page 156](#).

July 1996: Cargill, Balfanz, Dean, and Felten found another implementation flaw in the Java interpreter. This flaw allowed an attacker to mount some attacks on network services on other private-network machines. This flaw affected Netscape Navigator 3.0beta5 and was fixed in Navigator 3.0beta6. This attack is also called Casting Caution to the Wind. See [page 163](#).

August 1996: Balfanz and Felten found a flaw in Microsoft's Java implementation. The flaw allowed code in an attack applet to become a member of a security-critical Java package, thus gaining the ability to change various security parameters. This in turn gives the applet full access to the target machine's files and the network. This flaw affected Microsoft Internet Explorer 3.0beta3 and was fixed in Explorer 3.0beta4. The resulting attack is called Big Attacks Come in Small Packages. See [page 167](#).

February 1997: A pair of British hacker/consultants, Ben Laurie and Major Malfunction, discovered two separate flaws, one that worked only against Netscape Navigator 3.x and the other that worked only against Microsoft Internet Explorer 3.x. The Microsoft flaw was much more serious than the Netscape flaw. The fact that the attacks were browser specific serves to emphasize that different vendors have different approaches to implementing Java security. To be sure, both vendors are coding to the same specification, but there are many ways to skin a cat. The two attacks are called, Steal This IP Number and Cache Cramming, respectively. See pages [169](#) and [171](#).

March 1997: Security researchers at JavaSoft discovered a subtle error in the JVM and immediately patched it. This illustrates how serious JavaSoft is about security. The error was deep inside the Virtual Machine code and affected all browsers. The attack is called Virtual Voodoo. See [page 172](#).

April 1997: Soon after the release of JDK 1.1, the Princeton team discovered a serious flaw in the newly implemented code-signing system. The flaw allowed an attack applet to take on maximum privilege allocated inside a system. Because only the HotJava browser supported JDK 1.1, it was the only browser affected by the flaw. This attack is called The Magic Coat. See [page 172](#).

May 1997: The Kimera group at the University of Washington, led by Professor Brian Bershad, created a byte code Verifier of their own using formal methods and strict software engineering processes. In order to test their implementation, the Kimera group created an automatic test harness that generated millions of tests by mutation. They then tested their Verifier against commercially available Verifiers. In the course of testing, they identified 27 errors in various commercial Verifiers. One error was severe enough to warrant a patch. The others were patched in later releases of the JDK. This set of attacks is called Verifying the Verifier. See [page 174](#).

July 1997: Sun's response to the Verifier errors in May (which downplayed their severity) prompted the Kimera group to create an exploit that attacked Netscape Navigator 3.x. The attack was able to gain access to protected information in the browser's memory space, including private keys used in setting up secure socket layer (SSL) sessions. This attack is called the Vacuum Bug. See [page 177](#).

August 1997: Creative Concepts consultant Ben Mesander discovered a weakness in the way HTTP redirect operations were handled by Java. The weakness was particularly obvious in Microsoft Internet Explorer 3.x but was also present (in a subtler fashion) in Netscape Navigator 3.x. This attack is called Look Over There. See [page 178](#).

July 1998: Balfanz, Dean, Felten, and Wallach found a security flaw in the ClassLoader implementation of JDK 1.1 and JDK 1.2beta3 that allows a malicious ClassLoader to override the definition of built-in "system" types like `java.lang.Class`. When properly abused, this can lead to a type confusion attack that completely compromises security. The flaw requires use of a secondary flaw to gain a foothold. Mark LaDue had earlier discovered such a flaw in

Netscape 4.0x's Security Manager. Thus, Netscape 4.0x is exploitable, while other Java VMs (Sun's and Microsoft's) do not appear to be exploitable even though they contain the same flaw. This attack is called Beat the System. See page [182](#).

Una lista di malicious applet:

LaDue's malicious applets perform the following hostile activities:

- Play a sound file forever (our NoisyApplet is adapted from this one).
- Kill a browser with a CPU-hogging attack.
- Consume all available memory on your machine.
- Spin endless threads to consume resources.
- Display many hundreds of large black windows.
- Combine many denial-of-service attacks (windows, threads, and sounds) into one payload.
- Pop a fake dialog box requesting sensitive information (username and password).
- Surreptitiously perform remote calculation and report results back to the server.
- Forge mail.
- Kill all applet threads (except for self).
- Send your browser to a URL over and over again.
- Obtain your username.
- Fill all disk space available to the browser.
- Create an AppletClassLoader (a good staging ground for more serious attacks).
- Exercise mystery methods (undocumented but available) that crash a browser.
- Misuse native methods through the Java API, resulting in a crash.
- Deny legitimate use of the audio system by retaining control over it.
- Steal information about the SystemPrincipal and create an impostor.
- Determine exactly which plugins a browser has with help from JavaScript.
- Steal information from a Java Wallet (including username and password).
- Carry out some social engineering in order to rewire the Help button of the Java Wallet.
- Cause a modem connected to an arbitrary serial port to dial.

Java features for security:

I primi concetti base della sicurezza sono quelli che ormai i programmatori Java hanno sentito ripetere fino alla noia:

- **L'esistenza di un ambiente chiamato *Java Virtual Machine*.** E' questo l'ambiente dove il programma java (compilato solo all'80% deve solo essere interpretato al 20%) girerebbe.

Questo ambiente si incarica di verificare il codice prima di mandarlo in esecuzione, individuando eventuale codice maligno ed impedendogli di andare a ricoprire aree di sistema riservate.

- **La rimozione dell'aritmetica dei puntatori.**

Spesso l'estrema potenza di un linguaggio aumenta la complessità del programma aumentando quindi la generazione degli errori e quindi peggiorando la sua manutenibilità.

Java è stato costruito sulla base del linguaggio C++ estremamente potente, alleggerendolo dell'aritmetica dei puntatori. Questo significa che una volta assegnate delle aree di memoria ai dati, non è più possibile cambiarle modificando il valore del puntatore sotto le variabili. Così non è possibile che il programma ad un certo momento vada a puntare aree di memorie protette la cui violazione genererebbe un crash di sistema.

- **Verifica errori più profonda al momento della compilazione.**

Il compilatore oltre che effettuare una normale verifica sintattica del codice, ha l'incarico di verificare se determinate regole di programmazione sono state rispettate, costringendo il programmatore a codificare in un determinato modo. Per esempio sono considerati errori i punti di programma che non possono in nessun caso essere

raggiunti, o la mancanza della gestione di eccezioni (condizioni di errore) in seguito all'uso di particolari istruzioni che potrebbero sollevare quell'eccezione (per esempio nel caso di lettura di un file, è necessario gestire l'eccezione *IOException*, sollevata in seguito per esempio di file non trovato).

Anche questo diminuisce gli errori al momento del run-time rendendo quindi il programma più sicuro.

- **Garbage Collection.**

Un'altra caratteristica di Java che contribuisce alla sicurezza è la presenza di una *Garbage Collection* automatica. La *Garbage Collection* è la possibilità dell'ambiente di run-time di rilasciare automaticamente memoria non più utilizzata da un certo periodo di tempo. La *Garbage Collection* controlla tutte le aree di memoria entrando in funzione periodicamente e rilascia una determinata porzione di memoria quando questa non è più puntata o i puntamenti sono tutti interni ad essa (per esempio se si manda in esecuzione di un metodo, la memoria allocata da questo metodo, viene rilasciata al momento di fine esecuzione del metodo).

La memoria liberata è così disponibile al riutilizzo da parte del programma per altre esigenze rendendo il programma stesso più efficiente.

- **Altre caratteristiche del linguaggio.**

Esistono molte altre caratteristiche che contribuiscono a rendere sicuro il linguaggio Java. Queste includono per esempio dei test al momento della compilazione sul meccanismo di *casting* (processo di conversione da un oggetto ad un altro). Per esempio è possibile effettuare un cast da un array di Object ad un array di String solo se gli elementi dell'array sono effettivamente String.

Inoltre Java è dotato di modificatori pubblici, protetti e privati dei dati e dei metodi che contribuiscono a controllare la visibilità di questi. Dall'uso appropriato di questi è possibile tenere restringere gli accessi a dati, metodi e classi interne.

Un altro modificatore importate è il *final* che disabilita la possibilità in java di ridefinire classi e metodi e quindi impedisce che determinate classi importanti vengano ridefinite malignamente.

Fin qui abbiamo elencato le caratteristiche proprie del linguaggio che forniscono dei livelli di sicurezza piuttosto automatici.

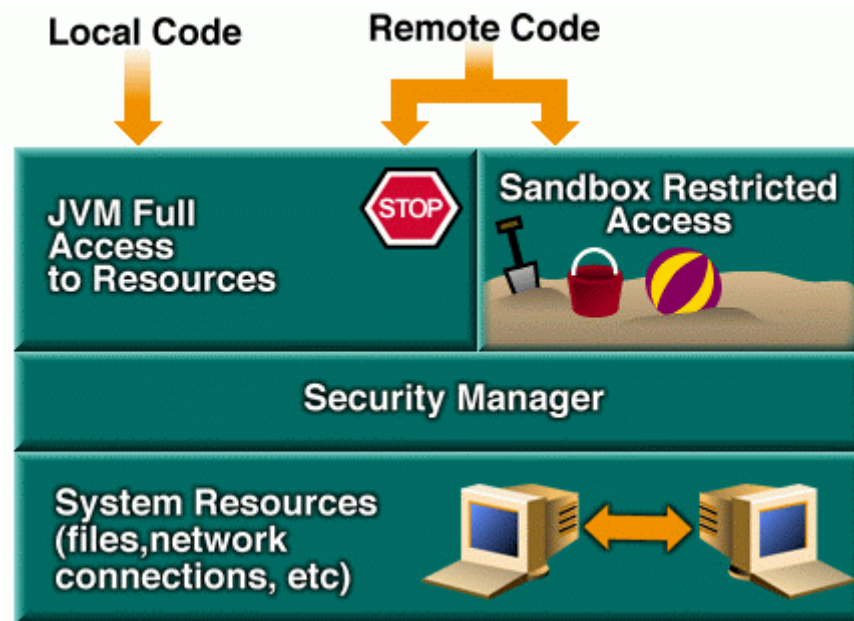
Soluzioni: dal JDK1.0 al JDK1.2:

Mostreremo nel seguito come la piattaforma Java sia evoluta nel tempo per permettere una gestione sempre più "precisa" (fine) degli accessi. L'obiettivo è quello di consentire operazioni sull'host (lettura o scrittura di files, connessioni ad altri hosts, ...) solo a del codice che sia "autenticato".

Vedremo come nel modello iniziale esistevano solo due classi di diritti (tutto o nulla) mentre l'evoluzione attuale permette di assegnare anche solo alcuni diritti.

JDK1.0

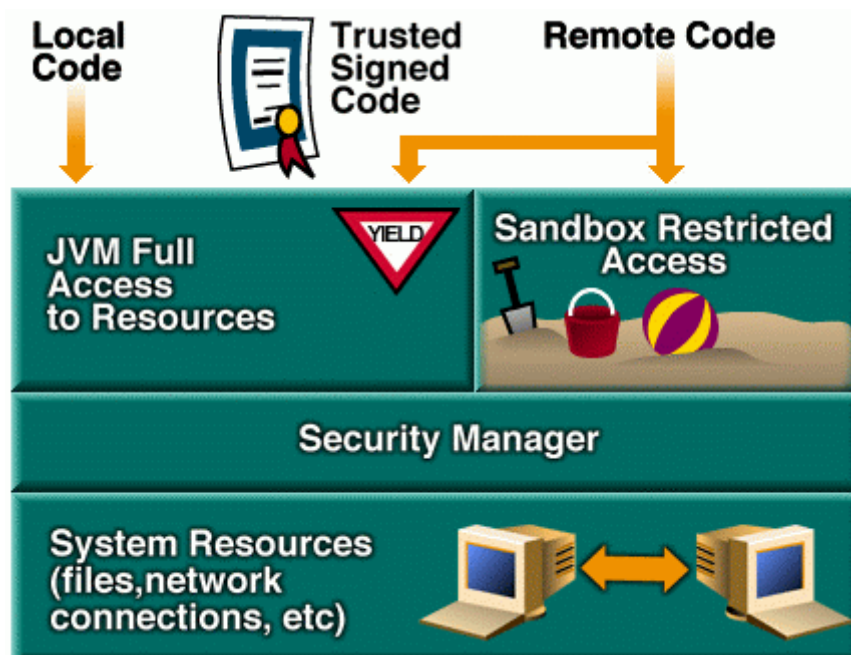
Nella versione del JDK1.0 esisteva un modello noto come "sandbox" in cui il *local code* (quello locale) aveva il diritto di utilizzare appieno le risorse del sistema mentre il *remote code* (applet) poteva girare solo sotto una sandbox, cioè un'area dedicata alle applet all'interno della quale l'applet può tutto. Al di fuori di tale area, però, l'applet non ha nessun diritto; non può leggere o alterare dati al di fuori dell'area, non può comunicare attraverso le sandbox. Il *Security Manager* (come anche nelle altre versioni) determina a quali risorse si ha il diritto di accesso, sia nel caso Full Access sia per la Restricted Area.



JDK1.1:

Viene introdotto il concetto di *Signed applet*: una applet, anche se scaricata dal Web (remote code) puo' avere il diritto di accedere alle stesse risorse del local code nel caso in cui sia trusted. Questo tipo di politica ha richiesto l'introduzione di chiavi per la firma del codice. Un Java Archive (JAR) file viene introdotto per contenere l'applet e tutte le informazioni relative. Tale JAR viene quindi firmato e spedito all'host di destinazione. Il target host controlla la firma: se la verifica porta a successo, l'applet ha pieno accesso alle risorse di sistema altrimenti viene confinato nella sandbox.

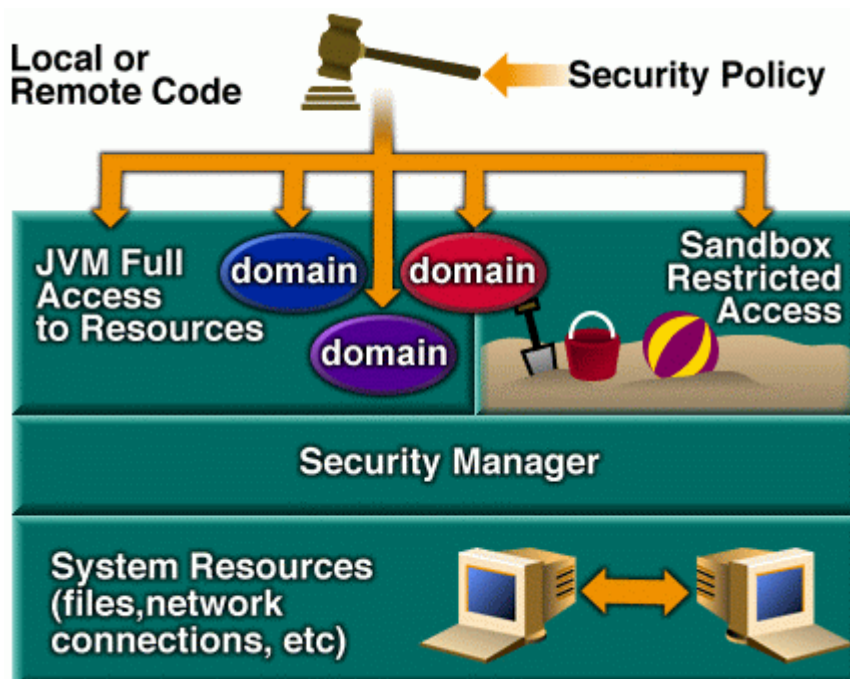
Affinche' i nuovi servizi di autenticazione fossero facilmente implementabili e' stata aggiunta alla prima release del JDK Security API JDK1.1 la **Java Cryptography architecture (JCA)** per le funzionalita' di crittografia e autenticazione.



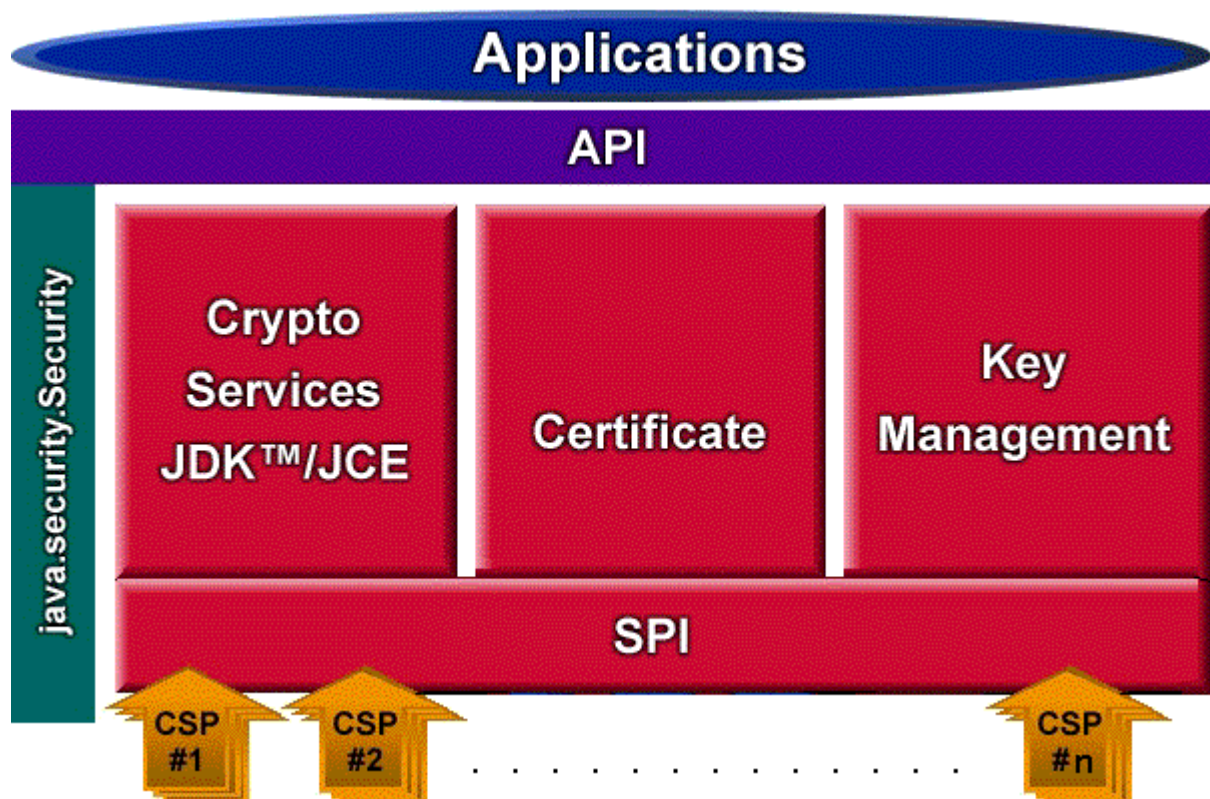
JDK1.2:

Rappresenta un passo di raffinamento ulteriore in cui local o remote code vengono gestiti nella stessa maniera: ad entrambi sono assegnati dei **permissions** (vedi sezione su ????????) che specificano quali risorse il codice puo' usare (ad esempio, lettura o scrittura di un particolare file o connessione con un particolare host). Quando un applet (locale o remote) viene caricata su di un target host, la security **policy** definisce l'insieme dei permessi da assegnare ad un codice proveniente da vari host e firmati da vari signers. Tutte le classi con gli stessi diritti di accesso vengono racchiusi in un **dominio**. Cio' vuol dire che, fondamentalmente, ogni applet puo' appartenere ad un proprio dominio e puo' usare una porzione delle risorse di sistema.

Se nessuna particolare policy viene definita (per default) l'applet NON puo' accedere a nessuna delle risorse del sistema: files, network facilities, screens, keyboards, disk drivers e printers. Anche comunicazioni fra Java programs sono disabilitate.



Con il passaggio dal JDK1.1 al JDK1.2, anche la JCA e' stata arricchita per gestire la creazione e manutenzione delle chiavi, per la gestione del key factory e per la certificazione. La prossima figura illustra i vari moduli che compongono il JCA:

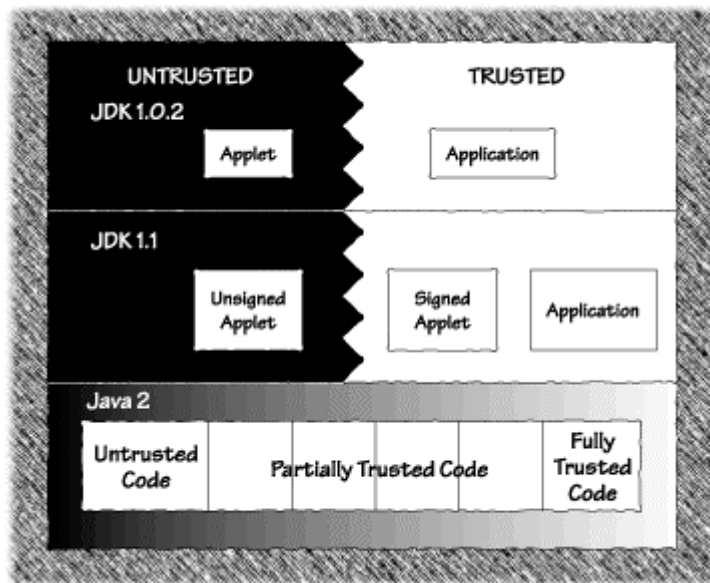


Il **JCE** (Java Cryptography Extension) estende il JDK per includere APIs per eseguire encryption, scambio di chiavi e message authentication, creando un platform-independent cryptography API. Una classe richiede dei servizi di crittografia che poi devono essere implementati dal gestore del **service provider interface (SPI)**. Ogni classe ha un SPI class che definisce quale particolare algoritmo di crittografia verra' usato (ad esempio, DSA or RSA).

La componente **Certificate** mette a disposizione delle classi ed interfacce per la gestione di certificati in formato X.509 v3 o in PGP o SDSI.

Il **Key management** mette a disposizione classi o interfacce per gestire il repository delle chiavi e dei certificati

Riassumendo:



Applet rights:

Vediamo ora cosa un applet puo' e cosa un applet NON puo' fare una volta scaricata sulla nostra macchina. Per default un applet NON puo':

- leggere o scrivere files sul file system del client
- fare connessioni di rete, eccetto con l'host di origine
- eseguire altri programmi sul client
- caricare librerie
- definire metodi nativi

In particolare:

files: un applet puo' leggere o scrivere su di un file solo se le viene fornito tale permesso, iscrivendola presso una "access control list" (vedi sezione ??? per dettagli);

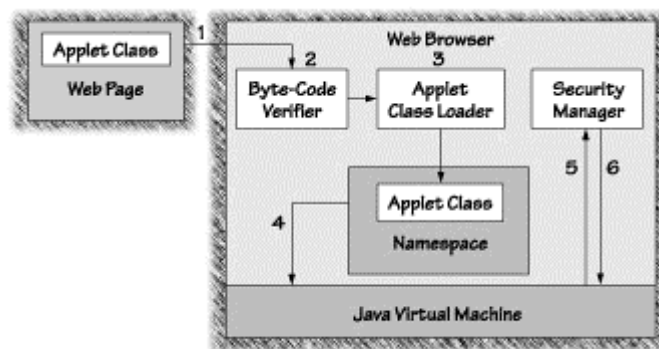
system properties: le applet sono abilitate a leggere alcune proprieta' di sistema e disabilitate per altre. Se il browser e' Java-enabled non c'e' modo di nascondere le proprieta' enabled.

Enabled	Disabled
Java version number	Java installation directory Java classpath User account name User home directory User's current working directory
Java vendor-specific string	
Java vendor URL	
Java class version number	
Operating system name	
Operating system architecture	
Operating system version	
File separator (eg, "/")	
Path separator (eg, ":")	
Line separator	

network connections: un applet puo' aprire una connessione solo con l'host che fornisce i .class file
run other programs: non e' permesso... assolutamente. Non e' neanche permessa la manipolazione di thread non relativi all'applet stessa

Implementazione della sicurezza nel JDK1.2:

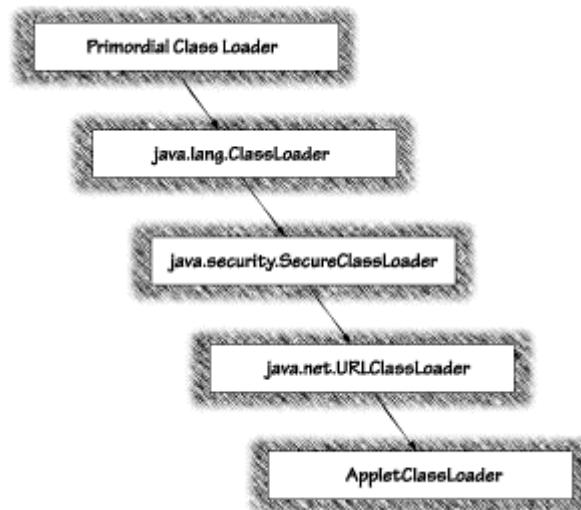
Prima di analizzare i packages Java utilizzati per implementare la sicurezza, mostriamo quelli che sono gli oggetti della JVM devoluti al caricamento ed al controllo delle applet. In particolare descrivere i compiti del ClassLoader (il modulo che carica il codice java) e del Security Manager (che come piu' volte mostrato serve per applicare le politiche di filtraggio)



ClassLoaders. La responsabilità di verificare e caricare il codice è della classe *ClassLoader* il quale risulta essere uno dei moduli più importanti per la sicurezza. Compito del *ClassLoader* è di determinare quali classi possono essere aggiunte per girare su di un ambiente Java e parte del loro lavoro è di assicurarsi che parti importanti del Java runtime environment non siano rimpiazzati da codice alterato. Il *ClassLoader* ha fondamentalmente due funzionalità:

- 1) Il *ClassLoader* deve trovare il byte code necessario alla VM per una particolare classe; tale byte code può essere caricato dal disco locale o dalla rete.
- 2) Il *ClassLoader* definisce i namespaces accessibili da differenti classi e le regole che mettono in relazione tali namespaces. I namespaces permettono alle classi Java di vedere differenti viste dell'ambiente, in funzione dal loro luogo di origine (local file, remote file, firewall remote file, ...). La gestione dei namespaces è di importanza critica per la sicurezza e diversi banchi sono stati trovati proprio nella gestione di tali oggetti.

Molte implementazioni della VM hanno utilizzato differenti class loaders per caricare codice da locazioni differenti. Come conseguenza, a tutto il codice caricato dallo stesso class loader vengono assegnate le stesse security policies. Con l'avvento del JDK1.1, sia l'informazione sull'origine del codice che quelle sul firmatario vengono utilizzate per fissare la policy. Solo *ClassLoader* che ha caricato il codice ne conosce la provenienza. In particolare, esistono diversi *ClassLoaders*:



Il Primordial Class Loader ha il compito di caricare le classi delle API di Java. Permette di realizzare il bootstrap. Inoltre il Primordial carica tutte le classi che sono locate nel ClassPath. I Class Loader Objects (le altre classi di ClassLoader) caricano codice Java ma non eseguono funzioni di bootstrap. In particolare gli AppletClassLoader sono responsabili del caricamento di classi dal browser e sono definiti dai produttori di browser Java-enabled.

Cosa accadrebbe se, un applet caricata dall'AppletClassLoader installasse un nuovo AppletClassLoader? Naturalmente si avrebbe che codice remote verrebbe trattato con diritti diversi da quelli originali, dando luogo ad un attacco sul sistema.

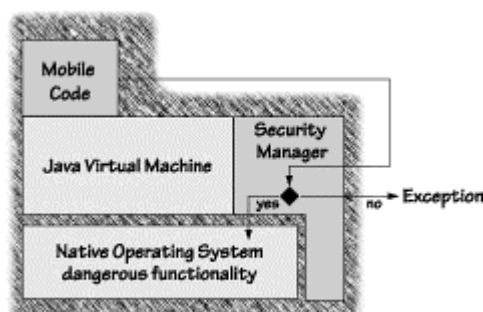
Per evitare tali problemi, si fa' in modo che l'AppletClassLoader carichi l'applet in un proprio namespace, in modo che l'applet possa lavorare solo sulle proprie classi e non su classi di sistema o di altri namespace.

I passi che vengono seguiti dal ClassLoader per caricare del byte code sono i seguenti:

- Determine whether the class has been loaded before. If so, return the previously loaded class.
- Consult the Primordial Class Loader to attempt to load the class from the CLASSPATH. This prevents external classes from spoofing trusted Java classes.
- See whether the Class Loader is allowed to create the class being loaded. The Security Manager makes this decision. If not, throw a security exception.
- Read the class file into an array of bytes. The way this happens differs according to particular class loaders. Some class loaders may load classes from a local database. Others may load classes across the network.
- Construct a Class object and its methods from the class file.
- Resolve classes immediately referenced by the class before it is used. These classes include classes used by static initializers of the class and any classes that the class extends.
- Check the class file with the Verifier.

Il SecurityManager. Permette di gestire la sicurezza delle applicazioni tramite l'estensione della classe *java.lang.SecurityManager*. Il compito del Security Manager e' di tenere traccia di quegli oggetti ai quali e' permesso compiere operazioni pericolose sul sistema. Un Security Manager ha quindi il compito di evitare l'esecuzione di molte operazioni a codice untrusted permettendole invece a codice trusted.

The Java library's use of the Security Manager works as follows:



- A Java program makes a call to a potentially dangerous operation in the Java API.
- The Java API code asks the Security Manager whether the operation should be allowed.
- The Security Manager throws a SecurityException back to the Java API if the operation is denied. This exception propagates back to the Java program.
- If the operation is permitted, the Security Manager call returns without throwing an exception, and the Java API performs the requested dangerous operation and returns normally.

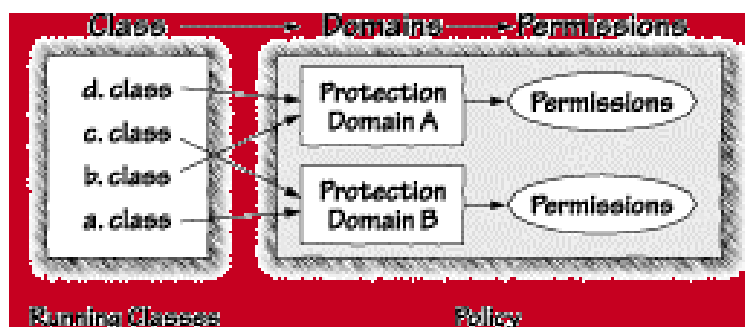
The Security Manager has the following duties:

- Prevent installation of new class loaders. The job of class loaders is to keep the namespaces properly organized. Because security checks are requested by classes in the Java library, applets must be prevented from spoofing the library classes.
- Protect threads and thread groups from each other. (Unfortunately, JDK 1.0 implementations of this piece of policy do not function properly. Some malicious applets, discussed in Chapter 4, have been written to take advantage of this. Better thread protection is provided in JDK 1.1 and above.)
- Control the execution of other application programs.
- Control the ability to shut down the VM.
- Control access to other application processes.
- Control access to system resources such as print queues, clipboards, event queues, system properties, and windows.
- Control file system operations such as read, write, and delete. Access to local files is strictly controlled.
- Control network socket operations such as connect and accept.
- Control access to Java packages (or groups of classes), including access to security enforcement classes.

The job of the Security Manager has been deeply affected by many of the new code-signing and access-control features now found in the Java security architecture (see [Chapter 3](#)). Using encryption-based authentication methods, the Security Manager in concert with other mechanisms can set up much more sophisticated rules for trusted, partially trusted, and untrusted applets.

Implementazione della sicurezza

Nella versione del JDK1.2 viene introdotto il concetto di Protection Domains, Permission e Properties. L'idea fondamentale è che ad ogni code proveniente da un certo URL e con una certa firma viene associato un protection Domain;



Ad ogni Domain vengono assegnati dei Permissions. Una Policy contiene i vari Permissions relativi a files da diversi URLs e con diverse firme.

Cosa accade quando una applicazione necessita l'uso di diversi Domains? È cruciale che se un dominio con Permission più bassi (DOMlow) comunica con uno con Permission più alti (DOMhigh), il primo NON acquisti permessi più alti di quelli a lui assegnati o non usi il DOMhigh con permessi di natura high. Si ha la regola generale che dice che

- Il permesso di un thread che usi diversi Domini è l'intersezione dei permessi di tutti i domini attraversati dall'esecuzione del thread

- Il permesso di Trust viene concesso solo se tutti i Domini sono Trust

Implementazione dei Permissions

Esistono fondamentalmente 3 classi per la gestione dei Permissions:

- 1) Permission
- 2) PermissionCollection
- 3) Permissions

Java.security.Permission e' la superclasse astratta di tutti i Permission. Usa due parametri : il "target name" per specificare il nome della risorsa per la quale il permesso e' concesso (ad esempio, un file) ed l'"action", cioe' l'operazione permessa sul target (ad esempio, read o write su di un file).

Java.security.PermissionCollection rappresenta una collezione di Permission objects per una singola categoria (avremo PermissionCollection di FilePermission, un'altra per i SocketPermission, ...)

Java.security.Permissions rappresenta una collezione di collezioni di Permission objects, cioe' una collezione eterogenea di Permissions.

Permissions include:

```
java.io.FilePermission for file system access
java.net.SocketPermission for network access
java.lang.PropertyPermission for Java properties
java.lang.RuntimePermission for access to runtime system resources
java.security.NetPermission for authentication
java.awt.AWTPermission for access to graphical resources such as windows
```

Java.io.FilePermission:

- Si usano due parametri: target e action

Fissano i permessi di lettura, scrittura ed esecuzione su files o intere directory. Ad esempio si puo' avere:

```
import java.io.FilePermission;
```

```
FilePermission p = new FilePermission("*", "read, write, execute");
FilePermission p = new FilePermission("/home/public/", "read");
FilePermission p = new FilePermission("/home/public/pub.txt", "read");
FilePermission p = new FilePermission("<<ALL FILES>>", "read");
```

Java.net.SocketPermission

- Si usano due parametri: target e action
- Concetto di **porta**
- Le azioni sulle socket includono **accept, connect, listen, resolve.**

```
import java.net.SocketPermission;
```

```
SocketPermission p = new SocketPermission ("*.com", "accept");
SocketPermission p = new SocketPermission ("*.sun.com", "connect");
SocketPermission p = new SocketPermission ("dwssrv.dm.univaq.it:80", "listen");
```

Java.security.BasicPermissions

- Estende la classe Permission
- Non si usa il parametro "action" definito in Permission; cio' porta ad avere un livello di descrizione piu' astratto, visto che si dice *su cosa si puo' operare*, senza dire *per farci cosa*
- Alcune sottoclassi sono java.lang.RuntimePermission, java.security.SecurityPermission, java.util.PropertyPermission e java.net.NetPermission.

Java.security.AllPermission

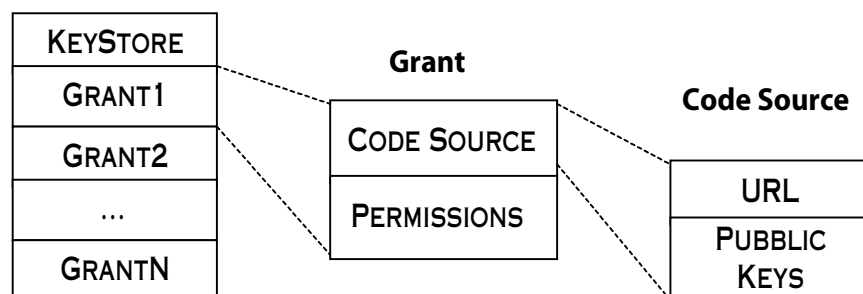
Abilita tutti i permessi e va' usato in maniera oculata e solo per classi trusted e di sicura provenienza.

Implementazione della Policy

Si specifica quali permessi sono disponibili per codice che proviene da vari source. L'oggetto Policy valuta la global policy e ritorna un *Permission object* che specifica i permessi concessi al codice da quel sito source.

Esiste un Policy Configuration File che solitamente e' memorizzato su di un file e che contiene un dbase di chiavi con i loro certificati ed un insieme di grant, ognuno dei quali contiene l'indirizzo del code source ed i permissions associati ad esso.

Policy Configuration File



La sintassi per assegnare un grant e' la seguente:

```
grant [signedBy "signer_names"] [, codeBase "URL"] {
```

```
    permission permission_class_name [ "target_name"
        [, "action" ] [, SignedBy "signer_names"];

    permission ...
};
```

Ed ecco un esempio:

```
grant codeBase "http://dwssrv.dm.univaq.it", signedBy "muccini" {
    permission java.io.FilePermission "/public/*", "read";
    permission java.io.FilePermission "/private/*", "read", signedBy "student";
    permission java.io.SocketPermission "*", "connect";
};
```

Nel momento in cui viene caricata una nuova pagina da un sito (code source), il meccanismo di sicurezza consulta il *policy object* per controllare quali permessi possono essere assegnati ai files. Il policy file puo' essere composto tramite un editore testuale od appositi tools.

Implementazione dei Domini

Un dominio e' costituito da tutte le classi con stesso codeSource e stesse keys. Se due classi hanno stessi permessi ma differenti codeSource, allora sono in differenti domini.

Le classi *java.security.ProtectionDomain* e *java.security.AccessController* implementano il concetto di Dominio e controllo di accesso.

La classe *java.security.AccessController* decide se concedere il permesso all'uso di risorse critiche ed eventualmente marca il codice come Privilegiato. Ogni code che richiede l'accesso ad una risorsa di sistema deve invocare metodi di questa classe.

Java Security Tools

I principali tools di sicurezza per Java sono:

- Keytool;
- Jar;
- Jarsigner;
- Policytool.

Tali tool sono disponibili per il jdk1.2 e sono stati realizzati con l'obiettivo di facilitare alcune operazioni relative alla sicurezza.

In dettaglio:

Keytool

Lanciando il comando “keytool” vengono mandati in esecuzione il Key and Certificate Management Tool che permette di:

- Creare chiavi (pubblica e privata);
- Creare delle richieste per certificati;
- Diffondere la propria public key;
- Gestire il keystore.

Le chiavi ed i certificati sono utilizzati per firmare applicazioni ed applet e vengono memorizzati in un database protetto noto con il nome di *keystore*.

Il formato del comando e' il seguente:

keytool <command><options>

Per ulteriori informazioni <http://java.sun.com/docs/books/tutorial/security1.2/summary/tools.html>,
<http://java.sun.com/docs/books/tutorial/security1.2/toolsign/index.html>

Java Archive Tool (jar)

Il formato Jar permette di memorizzare diversi files in un unico file di archivio, zippandone il contenuto. Viene solitamente utilizzato per memorizzare classi Java e risorse ausiliare associate ad applet o applicazioni. In questo modo diversi files possono essere downloadati in un'unica transizione HTTP, evitando di richiedere una nuova connessione per ogni file e risparmiando notevolmente sui tempi di download.

Nel momento in cui del codice deve essere *firmato*, esso viene posto in un Jar file (usando il *jar* tool) che viene poi firmato (usando il *jarsigner* tool).

Il comando e' il seguente:

jar [cf] <jar-file> <input-file(s)>

Per ulteriori informazioni <http://java.sun.com/docs/books/tutorial/security1.2/summary/tools.html>,
<http://java.sun.com/docs/books/tutorial/jar/index.html>

Jarsigner Tool

Permette di firmare dei Jar files e di verificare firme apposte su signed Jar files.

Il Jarsigner accede al keystore (creato con il keytool) quando ha bisogno di trovare una chiave privata ed il suo certificato. Naturalmente il keystore e' protetto da password e per poter accedere a tale informazione bisogna avere gli appositi diritti.

Il comando e' il seguente:

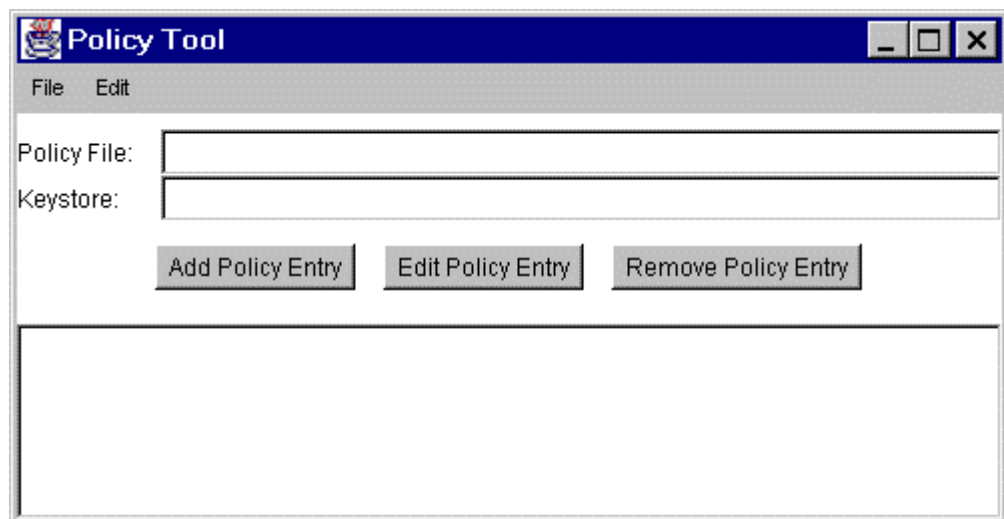
jarsigner <options> <jar-file> <alias>

Per ulteriori informazioni <http://java.sun.com/docs/books/tutorial/security1.2/summary/tools.html>,

Policy Tool

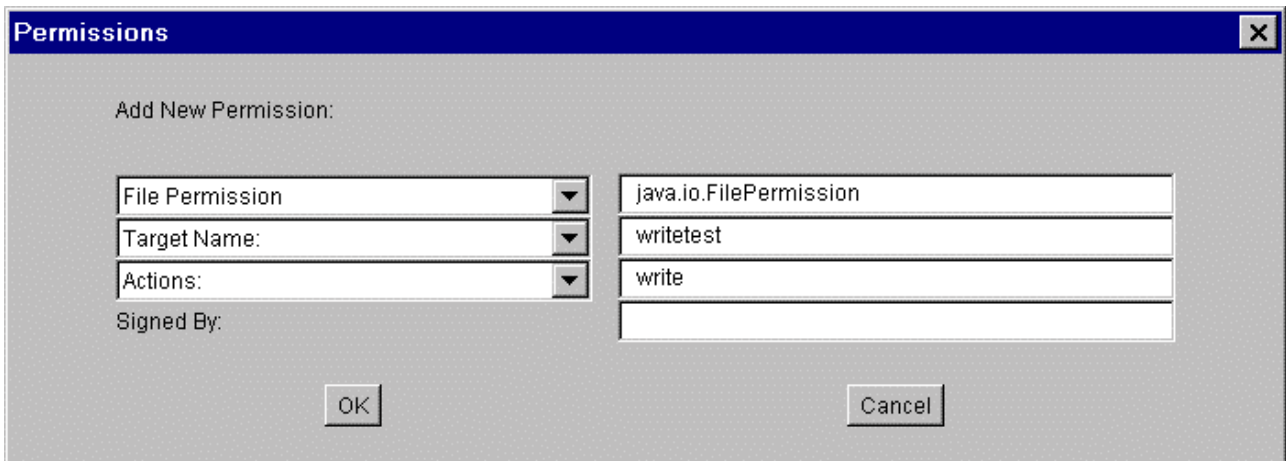
Utilizzando tale tool e' possibile creare e modificare il file di configurazione delle policy. E' fornito di una interfaccia grafica che permette di velocizzare le operazioni.

Nel momento in cui viene lanciato il comando "policytool", il tool cerca il policy file (memorizzato nella home e con estensione .java.policy); se esiste, riporta nella finestra le informazioni attuali altrimenti presenta una finestra vuota come la seguente:

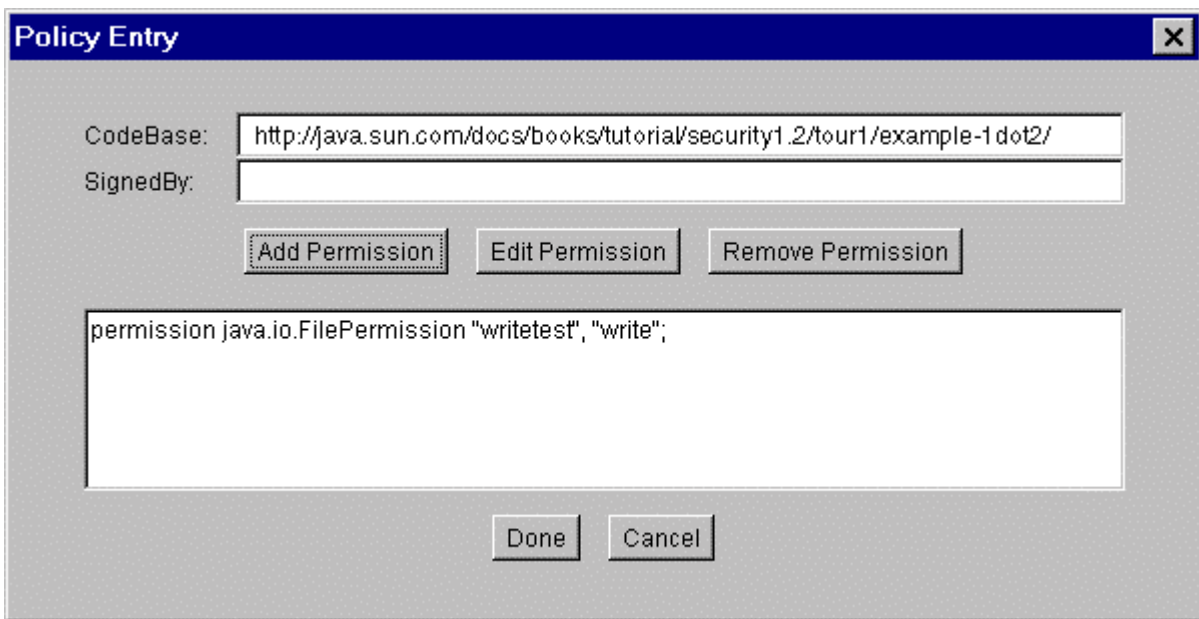


Premendo il pulsante "Add Policy Entry" e' possibile aggiungere una nuova policy specificando dei permessi per codice proveniente da particolari code source e firmati da particolari entita'.

Ad esempio, volendo assegnare un File Permission al codice proveniente dal sito <http://java.sun.com/docs/books/tutorial/security1.2/tour1/example-1dot2/> e senza alcuna firma, si dovra' prima di tutto riempire la schermata seguente



per fissare il tipo di permesso, il target e l'action per poi fissare l'indirizzo del code source e la signature:



Biblio degli articoli di riferimento

°°°Sito Javasoft sull'architettura e sulle classi per la sicurezza

<http://java.sun.com/products/jdk/1.3/docs/guide/security/spec/security-specTOC.fm.html>

Le Faq su applet security

<http://java.sun.com/sfaq/>

°°°Edward Felten Home Page – Java Security group

<http://www.cs.princeton.edu/sip/>

Articolo

<http://www.mokabyte.it/0197/security.htm>

Articolo sulle Javacard

<http://www.mokabyte.it/2000/01/javacard.htm>

Sicurezza in Java: la classe SecurityManager

<http://www.in-it.com/java/99-01.htm>

Is the Only Really Secure Computer a Disconnected Computer?

<http://developer.java.sun.com/developer/technicalArticles/Security/ReallySecure/index.html>

Lesson: Security Features Overview

<http://java.sun.com/docs/books/tutorial/security1.2/overview/index.html>

Lesson: Quick Tour of Controlling Applets

<http://java.sun.com/docs/books/tutorial/security1.2/tour1/>

Tools Summary

<http://java.sun.com/docs/books/tutorial/security1.2/summary/tools.html>

°°°Libro: Securing Java

<http://www.securingjava.com/>

SITO DI RIFERIMENTO:

<http://java.sun.com/security/>

<http://java.sun.com/products/jdk/1.2/docs/guide/security/index.html>

Hostile Applets

The Hostile Applets Home Page at www.rstcorp.com/hostile-applets

DigiCrime at www.digicrime.com

The Java Security Hotlist: Hostile Applets and Other Toys at

www.rstcorp.com/javasecurity/applets.html

WILEY

Securing JAVA

Getting Down to
Business with
Mobile Code

Gary McGraw
Ed Felten



<http://www.securingjava.com/>