

Sicurezza ed Affidabilità dei sistemi Informatici
2006/2007 prof. A. Mazzeo, ing. L. Coppolino

Sicurezza in Java:

architettura e tecniche di attacco

Bifulco Roberto 885/269
Migliaccio Barbara 885/225

Prefazione

La sicurezza è un aspetto molto delicato dello sviluppo di ogni sistema. In questo testo si esaminano aspetti legati alla sicurezza nell'ambiente Java, sia dal punto di vista della realizzazione dei meccanismi per garantirla, sia analizzando alcuni attacchi e tecniche per aggirare tali meccanismi. Chiaramente, tali tecniche possono essere utilizzate in modo malizioso e potrebbe sembrare non conveniente divulgarle, tuttavia, prendendo spunto da [1], riportiamo un testo che consideriamo ancora valido nonostante la sua età:

“Many well-meaning persons suppose that the discussion respecting the means for baffling the supposed safety of locks offers a premium for dishonesty, by showing others how to be dishonest. This is a fallacy. Rogues are very keen in their profession, and already know much more than we can teach them respecting their several kinds of roguery. Rogues knew a good deal about lockpicking long before locksmiths discussed it among themselves [...] if there be harm, it will be much more than counterbalanced by good.” [2]

Indice generale

Prefazione.....	2
Introduzione.....	3
1 Panoramica su Java.....	3
1.1 Evoluzione della sicurezza in Java.....	4
1.1.1 Jdk 1.0.....	4
1.1.2 Jdk 1.1.....	4
1.1.3 Jdk 1.2 (Java 2.0).....	4
1.2 Ciclo di vita di un'applicazione Java.....	5
1.3 Modello di sicurezza in Java 2.0.....	6
1.3.1 Identificazione del codice.....	6
1.3.2 Permessi.....	7
1.3.3 Dominio di protezione.....	7
1.3.4 Politiche di sicurezza.....	8
1.3.5 Access controller.....	9
2 Tecniche di attacco.....	11
2.1 Errori nella realizzazione delle classi di sistema.....	11
2.2 Type confusion.....	12
2.3 Class spoofing.....	14
2.4 Tecniche di aumento dei privilegi.....	16
Conclusioni.....	18
Bibliografia.....	19

Introduzione

Java nasce come una piattaforma per eseguire applicazioni su piccoli dispositivi mobili e si trasforma in breve tempo in un ambiente che fa della comunicazione distribuita il suo punto di forza. L'indipendenza dal sistema operativo sottostante e la possibilità di eseguire codice preso da una fonte remota sono le sue peculiarità. E' proprio quest'ultima caratteristica che ne accentua la necessità di garantire un ambiente sicuro, che non permetta a codice dannoso di acquisire il controllo sul sistema. Per perseguire quest'ultimo obiettivo divenuto specifica portante dell'intera piattaforma Java, è stato sviluppato un complesso meccanismo di sicurezza. Tuttavia, nessun sistema è esente da errori, e Java non è un'eccezione. Nel corso del tempo si sono verificati diversi problemi che hanno minato la sicurezza dell'ambiente e l'hanno reso vulnerabile ad attacchi.

Questo testo introduce gli aspetti fondamentali del modello di sicurezza di Java e le sue vulnerabilità. Il documento è strutturato in due distinte parti. La prima parte presenta il modello di sicurezza partendo da una piccola digressione storica dell'evoluzione della sicurezza in Java, fino a giungere a presentare i meccanismi fondamentali dell'attuale modello di sicurezza. La seconda parte presenta alcune tecniche di attacco che sono state adoperate per bucare la sicurezza di Java. In questa sezione del documento, quando necessario, si approfondiscono alcune caratteristiche dell'ambiente che sono state volutamente tralasciate nella prima parte, per non complicare la trattazione.

1 Panoramica su Java

Java nasce come una piattaforma su cui sviluppare ed eseguire applicazioni in modo “sicuro”, in ambiente distribuito (dove quindi il codice eseguibile è *mobile* fra più sistemi). La piattaforma è formata sostanzialmente da tre componenti:

- Un linguaggio di programmazione che viene compilato in un formato intermedio, indipendente dalla piattaforma, detto *bytecode*
- La *Java Virtual Machine* (JVM) che esegue il *bytecode*
- Un ambiente di esecuzione che fa partire la JVM e fornisce le classi di sistema

Per sviluppare codice sicuro, Java cerca di sollevare il programmatore dall'occuparsi delle parti del programma che generano maggiori errori e debolezze:

- La gestione della memoria non è curata dal programmatore, ma è l'ambiente che si occupa di gestire lo spazio di indirizzamento dinamico, adoperando un *garbage collector* per deallocare la memoria.
- Non esiste un'aritmetica dei puntatori. Anche se tutti gli oggetti Java si utilizzano tramite puntatori, non è possibile eseguire operazioni sul valore di questi.
- Le condizioni di *overflow* su stringhe ed *array* sono controllate dall'ambiente.

In aggiunta a quanto detto, Java fa uso di variabili strettamente tipizzate (*type safety*), garantendo quindi maggiore affidabilità nel trattamento dei dati.

1.1 Evoluzione della sicurezza in Java

Nel corso della sua storia, Java ha subito diversi cambiamenti per adattarsi alle esigenze di sviluppatori ed utenti e per migliorare le sue caratteristiche. Come tutti gli altri aspetti di Java, anche il modello di sicurezza ha subito questo processo.

1.1.1 Jdk 1.0

Il modello di sicurezza della prima versione di Java è molto semplice. Tutto il codice eseguibile viene diviso in due categorie: *trusted* ed *untrusted*.

Per decidere a quale categoria appartiene un blocco di codice se ne esamina la provenienza: se il codice risiede localmente, quindi è presente sul *file system* della macchina che lo sta eseguendo, è considerato *trusted*. In tutti gli altri casi il codice è *untrusted*.

Java assegna due differenti ambienti di esecuzione per codice *trusted* e *untrusted*. Il primo è eseguito in un ambiente privo di limitazioni, e può quindi compiere qualsiasi operazione. Il secondo è eseguito in un ambiente estremamente limitato, dove tutte le operazioni potenzialmente dannose sono inibite. Questo ambiente limitato è noto col nome di *sandbox*, e l'intero modello di sicurezza del Jdk 1.0 è generalmente indicato come modello a *sandbox*.

1.1.2 Jdk 1.1

Col Jdk 1.1 si cerca di consentire al codice fidato proveniente da remoto la possibilità di eseguire maggiori operazioni. Per rendere il codice “fidato” si utilizza un meccanismo di crittografia a chiave asimmetrica. Se il codice è firmato, ed il suo firmatario è riconosciuto dall'ambiente come attendibile, allora il codice è considerato *trusted*.

In questo modo anche il codice remoto può essere eseguito fuori dalla *sandbox*, tuttavia persistono molte limitazioni:

- Non esiste granularità nell'assegnazione dei privilegi. Il codice è eseguito o nella *sandbox* o fuori da questa.
- Il codice presente sul *file system* locale è considerato sempre *trusted* e non controllato.

1.1.3 Jdk 1.2 (Java 2.0)

Con la versione 1.2 del Jdk, si sono risolte le limitazioni dei precedenti modelli di sicurezza. E' cambiata infatti l'ipotesi iniziale per cui tutto il codice locale veniva considerato *trusted*, ed i privilegi sono assegnati con grana fine, definendo delle *policy* che associano al codice, identificato dalla provenienza e dalla firma, dei permessi specifici.

1.2 Ciclo di vita di un'applicazione Java

Il ciclo di vita di un'applicazione Java integra una serie di processi utili ad assicurare la sicurezza del sistema (*Illustrazione 1.1*).

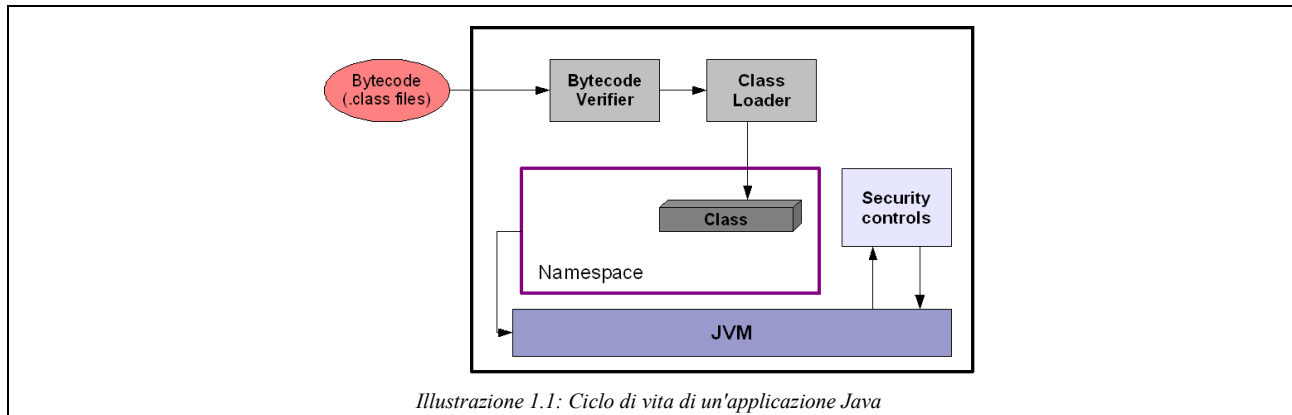


Illustrazione 1.1: Ciclo di vita di un'applicazione Java

Quando deve essere eseguito del *bytecode*, innanzitutto viene attivato un controllo, detto *bytecode verifier*, che controlla che sia rispettato il corretto formato e la struttura del *bytecode*. Sostanzialmente la verifica assicura che:

- Non si creino falsi puntatori.
- Non si violino restrizioni di accesso.
- Sia rispettato il tipo degli oggetti.
- Non ci siano *stack overflow*.
- Sia corretto il numero e il tipo dei parametri dei metodi.

Superata la verifica, il *bytecode* viene passato ad un *ClassLoader* perché venga creata nell'ambiente la classe. Il *ClassLoader* è utile a definire *namespace* separati, ma è anche un componente molto delicato, poiché è responsabile del caricamento di tutte le classi nel sistema (Per questa ragione ci sono forti limitazioni nella possibilità di definire *ClassLoader* personalizzati).

L'ambiente Java è dotato di un *ClassLoader* particolare, il *PrimordialClassLoader*, che ha il compito di caricare tutte le classi di sistema utilizzando i meccanismi di accesso, forniti dal sottostante sistema operativo, per fare il *bootstrap* della piattaforma.

Il *PrimordialClassLoader* è il primo ad essere caricato, tutti gli altri *ClassLoader* vengono caricati successivamente dal *PrimordialClassLoader*, poiché alcuni di questi sono fra le classi di sistema, oppure sono addirittura caricati da applicazioni d'utente che ne definiscono di propri (Chiaramente un *ClassLoader* definito dall'utente viene caricato esso stesso attraverso un altro *ClassLoader* appartenente invece al sistema, questo è dovuto al fatto che ogni *ClassLoader* è una classe).

L'algoritmo generale realizzato da un *ClassLoader* è il seguente:

1. Determina se la classe è già stata caricata.
2. Consulta il *PrimordialClassLoader* per verificare se la classe può essere caricata dal *classpath* di sistema.
3. Controlla che il *ClassLoader* abbia i permessi necessari.
4. Costruisce un oggetto della Classe partendo dal *bytecode* (trattato come *array* di *byte*).
5. Risolve le classi referenziate da quella appena caricata e ne verifica il *bytecode*.

Una volta caricata, la classe può procedere con tutte le operazioni che via via la interesseranno nel corso dell'esecuzione (Creare sue istanze, eseguire metodi, ecc.). Ogni volta che un'operazione potenzialmente dannosa deve essere eseguita, la JVM effettua un controllo sui permessi posseduti dalla particolare classe. Il controllo è effettuato tramite un'entità addetta alla sicurezza, che, sostanzialmente, identifica il codice e ne verifica i permessi di eseguire una determinata operazione.

Questa entità fino al JDK 1.1 era il *SecurityManager*, che semplicemente decideva se attribuire o no il codice alla *sandbox*. Dal JDK 1.2 si utilizza invece l'*AccessController* che aggiunge tutte le funzionalità per effettuare un controllo a grana fine dei permessi. Per quanto non più utilizzato, il *SecurityManager* esiste ancora nelle versioni successive al JDK 1.2, per la compatibilità all'indietro.

1.3 Modello di sicurezza in Java 2.0

Come anticipato, il modello di sicurezza di Java 2.0 introduce la possibilità di definire delle *policy* di accesso alle risorse con una grana fine, permettendo, allo stesso tempo, di trattare in modo più preciso il codice indipendentemente dalla sua localizzazione. Si presenta quindi un cambio delle ipotesi, rispetto alle versioni precedenti, riguardo al codice. Più precisamente, non si considera più fidato tutto il codice presente sul *file system* locale, ma il codice viene trattato esclusivamente in base alla combinazione della sua provenienza e della sua firma (con cifratura a chiave asimmetrica). Con queste informazioni, il sistema decide se concedere o meno i permessi di accesso alle risorse, al blocco di codice.

1.3.1 Identificazione del codice

La coppia di informazioni provenienza-firma costituisce la vera e propria identità del codice. All'interno dell'ambiente Java è utilizzata la classe *CodeSource* per tenere conto delle suddette informazioni collegate al codice. In *Illustrazione 1.2* è presentato il diagramma UML collegato alla classe *CodeSource*. La classe *Certificate* identifica il certificato associato al codice, con le rispettive chiavi pubbliche. La classe *URL* rappresenta la locazione del codice.

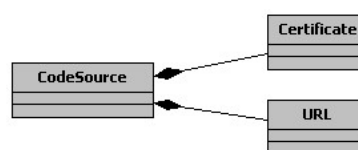


Illustrazione 1.2: *CodeSource* class

1.3.2 Permessi

L'oggetto fondamentale per regolare quello che un blocco di codice può o non può fare è il *Permission*. Java prevede la presenza di un permesso per ogni azione potenzialmente dannosa che può essere eseguita, inoltre, un utente può definire permessi personalizzati per le sue applicazioni, che vengono poi integrati nel modello di sicurezza Java. Quest'ultima possibilità consente di estendere ed adattare il modello di sicurezza a tutte le applicazioni.

Da un punto di vista pratico il *Permission* contiene due informazioni: *subject*, *actions*. Il *subject* è la risorsa cui il permesso fa riferimento, le *actions* sono le operazioni che il permesso concede su tale risorsa. Un blocco di codice che ha un *permission*, ha quindi la possibilità di eseguire le *actions* sul *subject* specificato dal *permission*.

In Java un *permission* è rappresentato da un'omonima classe *Permission*, il cui costruttore accetta due stringhe indicanti *subject* ed *actions*:

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

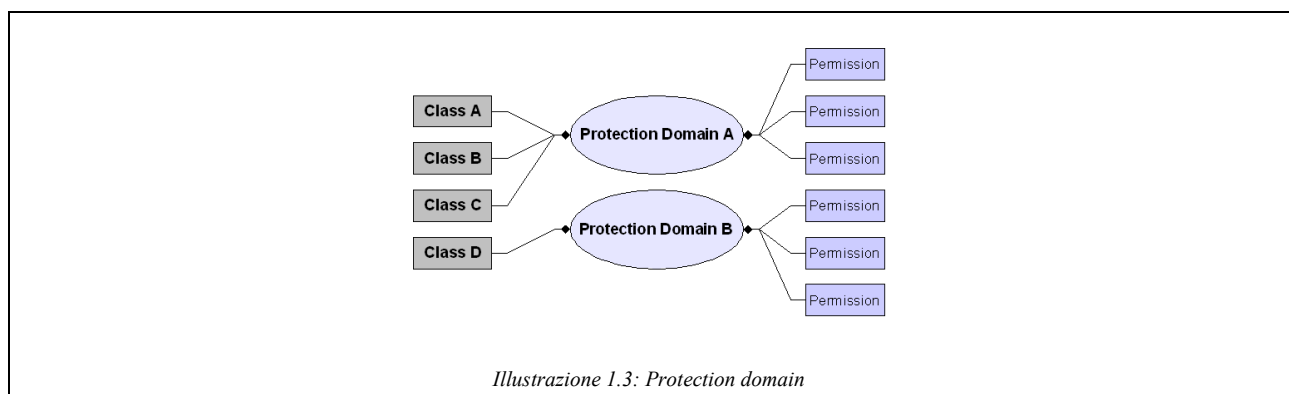
Dalla classe *Permission* derivano diverse altre classi che specializzano i permessi in base all'ambito in cui vengono adoperati. Alcuni esempi sono i *FilePermission*, *SocketPermission*, ecc.

La classe *Permission* contiene un metodo, *implies*, per semplificare la gestione dei permessi. Questo metodo esprime quali altri permessi sono automaticamente concessi quando viene concesso il permesso. Nell'esempio precedente, concedere un permesso su *"/tmp/*"* corrisponde a concedere permessi su tutti i *file* presenti all'interno della cartella, quindi, il metodo *implies* di quel permesso conterrà permessi di lettura per ciascuno dei *file* presenti nella cartella.

1.3.3 Dominio di protezione

Un concetto fondamentale per la sicurezza di un sistema è quello di dominio di protezione [3]. Se definiamo *principal* un'entità in un sistema a cui sono concessi dei permessi, allora, un *protection domain* è definito da un insieme di oggetti che sono direttamente accessibili ad un *principal*. In Java questo concetto si traduce in un insieme di classi le cui istanze hanno gli stessi permessi.

L'utilizzo dei domini di protezione è un utile meccanismo per raggruppare ed isolare unità di protezione. Ogni dominio di protezione, in Java, è sostanzialmente formato da due entità: *CodeSource* e l'insieme dei *permission* collegati allo specifico *CodeSource* (Illustrazione 1.3).



Un dominio di protezione si traduce nella pratica nella classe *ProtectionDomain*. Un oggetto *ProtectionDomain* contiene al suo interno un *CodeSource* che indica che a tale dominio appartengono tutte le classi provenienti dal *CodeSource* indicato, un insieme di permessi, ed un *array* di *principal*. L'*array* tiene conto degli utenti che attualmente stanno utilizzando il codice appartenente a quel dominio.

Java realizza due fondamentali domini di protezione: *system domain* ed *application domain*. Altri domini possono essere definiti dall'utente. E' anche possibile assicurare che i domini non di sistema non vedano la presenza degli altri domini non di sistema, ogni dominio infatti può essere dotato di un proprio *ClassLoader*.

1.3.4 Politiche di sicurezza

Una politica di sicurezza (*Policy*) non fa altro che esprimere quali permessi concedere e a chi concederli. La presenza delle *policy* consente di personalizzare il comportamento del sistema, per adattarlo a diversi scenari applicativi. Se ad esempio il sistema è collegato ad una rete che non ha accesso all'esterno e i cui componenti sono tutti fidati, si potrebbe decidere di adottare una *policy* che conceda tutti i diritti a tutti i blocchi di codice. Viceversa, un sistema esposto all'esterno potrebbe concedere un accesso molto limitato al codice non firmato, e fare una gerarchia fra le diverse firme, concedendo ad alcune più permessi e ad altre meno.

A *runtime* una *policy* viene rappresentata nell'ambiente Java da un'omonima classe *Policy*, contenente tutte le funzioni utili al sistema a verificare i permessi concessi e i proprietari di tali permessi.

L'archiviazione delle *policy* è fatta attraverso *file* binari o in ASCII, o anche tramite archiviazione in un database. E' definita una sintassi base per descrivere un *file* di *policy*, l'esempio successivo ne presenta la struttura:

```
grant signedBy "Roland" {
    permission java.io.FilePermission "C:\\users\\Cathy\\*", "read";
};
```

Nell'esempio si concede al codice firmato da *Roland* la lista di permessi inclusi fra le parentesi. Chiaramente *Roland* è un nome simbolico, e viene adoperato nel *file* di *policy* solo se precedentemente si è specificato un archivio contenente la risoluzione di tali nomi simbolici. Un esempio di dichiarazione di archivio è il seguente:

```
keystore "http://foo.bar.com/blah/.keystore";
```

Un *keystore* è appunto l'archivio sopracitato, che può essere contenuto in un altro *file* locale o remoto. Il contenuto di un *keystore* è una serie di *entry* contenenti la coppia (nome certificato, chiave), dove "nome certificato" è un nome simbolico.

Così come il codice può essere trattato in base alla firma, è possibile trattarlo anche in base alla provenienza, o in combinazione con entrambe le caratteristiche:

```
grant codeBase "http://java.sun.com/*", signedBy "Li" {
    permission java.io.FilePermission "/tmp/*", "read";
    permission java.io.SocketPermission "*", "connect";
};
```

Infine, è possibile concedere permessi non solo ad un blocco di codice, ma anche ad un *principal*:

```
grant principal javax.security.auth.x500.X500Principal "cn=Alice" {  
    permission java.io.FilePermission "/home/Alice", "read, write";  
};
```

Un *file* di configurazione presente nelle *directory* di sistema dell'ambiente Java mantiene traccia della posizione dei *file* di *policy*. Tale *file* consente di specificare la posizione di un *file* di *policy* di sistema predefinite e di *file* di *policy* di utente. Qualora non fosse specificata nessuna posizione, per nessun *file*, l'ambiente conserva al suo interno una *policy* predefinita che è quella del modello a *sandbox*.

Maggiori informazioni sulla sintassi dei *policy file* e sulla loro gestione sono presenti in [4].

1.3.5 Access controller

Fino ad ora si sono presentati dei concetti utili al trattamento della sicurezza nel sistema, ma non è stato ancora esaminata l'entità che effettivamente garantisce, funzionalmente, il rispetto dei domini di protezione e delle *policy*. Questa entità prende il nome di *Access Controller*.

Supponiamo che un'applicazione voglia richiamare un metodo M protetto (generalmente sono protetti tutti i metodi che sono riconducibili al dominio di protezione del sistema). Prima di eseguire il proprio compito, il metodo chiama, direttamente o indirettamente, l'oggetto *AccessController* per verificare che l'applicazione abbia i permessi necessari. Se i permessi sono presenti, il metodo viene eseguito correttamente, in caso contrario è sollevata una *SecurityException*.

Concettualmente il ruolo dell'*Access Controller* è molto semplice, deve solo verificare che un'applicazione abbia i permessi necessari a chiamare un metodo protetto.

Una prima complicazione che si presenta è la possibilità che un metodo protetto sia l'ultimo chiamato dopo una catena di chiamate. In questo caso, non è detto che i metodi appartengano tutti a classi facenti capo allo stesso dominio di protezione. La situazione generale è che un *thread* attraversi più domini di protezione nel corso della sua vita. E' necessario quindi stabilire una regola per decidere quali privilegi effettivamente assegnare al *thread*. L'approccio utilizzato è quello che massimizza la sicurezza: i permessi concessi sono solo quelli presenti in tutti i domini attraversati dal *thread*.

Ad esempio, se un *thread* attraversa i domini di protezione 1, 2, 3, che hanno rispettivamente i permessi a,b,c; a,c; a,d; allora il *thread* avrà come permesso il solo permesso "a" presente in tutti e tre i domini.

Supponendo quindi che un metodo protetto sia l'*m*-esima chiamata di una catena di chiamate ordinata da 1,2...*m*, l'algoritmo utilizzato dall'*Access Controller* per determinare se il metodo può essere eseguito è il seguente:

```
i=m;  
while(i>0){  
    if(il chiamante i-esimo non ha il permesso)  
        throw AccessControlException;  
    i = i-1;  
};
```

A quanto esposto va aggiunta un'eccezione. Per particolari esigenze, l'*Access Controller* mette a disposizione un meccanismo particolare in modo che alcuni metodi che lo necessitano possano ignorare la catena dei chiamanti e verificare i permessi solo dell'ultima classe che chiama il metodo.

Se infatti il progettista del metodo protetto lo ritiene necessario, può richiamare il metodo *doPrivileged()* dell'*Access Controller* che verifica che solo l'ultimo chiamante della catena abbia i permessi necessari. L'algoritmo precedente si modifica in:

```
i=m;
while(i>0){
    if(il chiamante i-esimo non ha il permesso)
        throw AccessControlException;
    else if( il chiamante i-esimo è privileged)
        return;
    i = i-1;
};
```

Quindi se il chiamante del metodo è *privileged*, non si continua a verificare i permessi di tutta la catena, ma l'*Access Controller* ritorna semplicemente dalla verifica senza sollevare eccezioni.

Molto spesso la catena dei chiamanti, di cui si è discusso precedentemente, viene chiamata *Access Control Context*. Il concetto di contesto è molto importante se si considera un caso non inusuale in cui un *thread* ne genera un altro. Poiché ogni *thread* ha uno *stack* proprio, il *thread* figlio non avrebbe alcuna catena di chiamanti reperibile dal punto di vista dell'*Access Controller*, in altre parole, si avrebbe un cambiamento di *Access Control Context*. Questa evenienza potrebbe causare facilmente errori nella gestione della sicurezza. Per questo motivo quando viene creato un *thread* figlio, questi eredita l'*Access Control Context* del padre. Sostanzialmente Java utilizza una classe *AccessControlContext* per incapsulare il contesto e passarlo al *thread* figlio come parte della sua struttura dati. L'*AccessController* quando deve verificare un permesso, controlla prima il contesto del *thread* e successivamente va a controllare l'eventuale contesto ereditato. L'algoritmo diviene quindi:

```
i=m;
while(i>0){
    if(il chiamante i-esimo non ha il permesso)
        throw AccessControlException;
    else if( il chiamante i-esimo è privileged)
        return;
    i = i-1;
};

inheritedContext.checkPermission(permission);
```

Dopo aver esposto l'algoritmo utilizzato per verificare la possibilità di chiamare un metodo, è bene specificare cosa effettivamente fa l'*Access Controller* per controllare la presenza di un permesso.

Come visto in precedenza, ogni classe è identificata da un *CodeSource*. L'*Access Controller* preleva questa informazione e la confronta con i domini di protezione per vedere a quale dominio appartiene la classe. Viene poi prelevato l'insieme dei permessi associati alla classe dalla *policy* in uso sul sistema. Se lo specifico permesso richiesto dal metodo protetto è presente fra quelli concessi alla classe, allora l'*Access Controller* non solleva eccezioni.

2 Tecniche di attacco

Il motivo principale per cui si compie un attacco è tentare di acquisire il controllo di un sistema, per poi avere libero accesso alle sue risorse, siano queste dati riservati (password e dati personali), risorse elaborative (il sistema può essere usato come piattaforma di lancio di un ulteriore attacco) o qualunque altra risorsa. Tutte le operazioni di un attacco vertono quindi attorno al tentativo di aumentare i privilegi di accesso che si hanno sul sistema. Nel caso di Java, come visto nella precedente sezione di questo testo, il sistema che concede privilegi è costituito dal *Java Runtime Environment*. Lo scopo di un attacco è quindi sfruttare alcune debolezze o difetti dell'ambiente per avere accesso alle entità che regolano la sicurezza dello stesso e modificarne i parametri.

Le tecniche esposte nel seguito sono ben note, almeno per l'idea su cui si basano, a quanti abbiano affrontato il problema della sicurezza (non solo in Java). Perché una delle tecniche esposte sia realmente efficace è necessario che l'ambiente Java presenti dei difetti nella sua realizzazione. Non si può infatti pensare di riuscire a portare a termine con successo un attacco, se il sistema attaccato non presenta difetti. In quest'ottica, le tecniche di attacco possono essere considerate come tecniche che portano alla luce le vulnerabilità (i difetti) del sistema. Chiaramente, la JVM è realizzata in varie forme ed in diversi modi, ad esempio la JVM realizzata dalla *SUN* è diversa da quella realizzata all'interno di *Internet Explorer* dalla *Microsoft*. Una interessante lista (non aggiornata) degli attacchi portati alle JVM è presente in [8].

2.1 Errori nella realizzazione delle classi di sistema

Il modo più semplice di rompere la sicurezza di un sistema è trovare errori nelle entità che realizzano la sicurezza. In Java queste entità sono le cosiddette classi di sistema (facendo riferimento alla prima sezione di questo testo, si possono citare *AccessController* o un *ClassLoader*).

Trovare un difetto in una di queste classi può dare la possibilità di modificare i dati da queste trattate e quindi, cambiare le modalità di esecuzione dell'ambiente. Nel seguito sono elencati gli errori più comuni che possono verificarsi. Ancora una volta, una specifica implementazione della JVM può presentare alcuni di questi errori, un'altra presentarne altri ed un'altra ancora non presentarne affatto, è chi compie l'attacco che deve essere in grado di far venire alla luce il difetto della JVM che sta attaccando e sfruttarlo, perché l'attacco abbia successo:

1. Ogni metodo potenzialmente pericoloso di una classe di sistema dovrebbe essere realizzato in modo che, prima dell'esecuzione delle sue funzioni, verifichi che il chiamante abbia i permessi necessari ad eseguirlo. Qualora questo non avvenisse per un difetto di realizzazione, non ci sarebbe nessun controllo di sicurezza ed il metodo sarebbe eseguibile da chiunque. Un attacco potrebbe consistere nella ricerca di un metodo che presenti questo difetto (sarebbe un errore grossolano, ma sempre possibile) e sfruttare i privilegi che garantisce l'uso di quel metodo per compiere operazioni illegali.
2. Se nella realizzazione di una classe di sistema non si progettano adeguatamente le limitazioni al codice, potrebbero crearsi delle situazioni in cui si violano le politiche di sicurezza. Ad esempio, se un metodo di una classe è dichiarato *public* ma non *final*, se la classe può essere estesa per ereditarietà, si può fare un *override* del metodo che potrebbe portare ad una falla nel sistema di sicurezza. Ad esempio, se il metodo di caricamento di una classe in un *ClassLoader* fosse soggetto a questa problematica, sarebbe possibile caricare

nel sistema praticamente qualsiasi tipo di codice. Generalmente tutti i metodi dell'*AccessController* sono i più bersagliati dalla ricerca di questi difetti.

3. Se una classe dipende strettamente dalla sua inizializzazione, è sempre presente un rischio che tali classi non siano propriamente inizializzate. Sostanzialmente questa problematica è presente quando non sono fatti appropriati controlli di sicurezza nel costruttore della classe.
4. Un'altra mina per la sicurezza del sistema è costituita dalle *inner class*. Il *bytecode* di Java non prevede un comando apposito per trattare le *inner class*, ma queste sono considerate classi a tutti gli effetti uguali alle altre. Tuttavia una *inner class* ha libero accesso ai metodi ed agli attributi privati della classe che la contiene. Perchè questo avvenga i compilatori generalmente cambiano la visibilità dei metodi della classe contenitore da *private* ad una visibilità di *package*.
5. Tutte le classi di Java possono implementare l'interfaccia *Cloneable* per indicare che il metodo *clone* della classe *Object* è legittimato ad essere eseguito. Il metodo *clone* si adopera per effettuare una copia esatta della classe. Se si definisce una sottoclasse e si implementa l'interfaccia *Cloneable* si può ottenere una copia della classe senza passare per nessuno dei suoi costruttori (evitando di fatto i controlli di sicurezza che questi effettuano), poiché si esegue una precisa copia dell'immagine di memoria. La soluzione di questo problema si ottiene specificando la classe come *uncloneable*. Per maggiori dettagli sull'uso del metodo *clone* si rimanda a [7].
6. Alcune classi di sistema possono essere dichiarate tramite l'apposita interfaccia *serializable*. Una classe serializzabile, viene trasformata in un *array* di byte per essere salvata. In questo modo, è possibile leggere il contenuto del *file* per recepire informazioni riservate. Allo stesso modo, quando una classe deve essere deserializzata per venire caricata nuovamente nell'ambiente di esecuzione, è possibile fornire un *file* opportunamente modificato in modo da impostare i campi della classe per violare le regole di sicurezza.
7. Se una classe di sistema restituisce un riferimento ad un *array* di dati interno, invece di una sua copia, sarebbe possibile modificare i dati di tale *array* influenzando i dati della classe di sistema.
8. Un ulteriore errore sufficientemente grossolano è l'esecuzione della verifica del tipo di una classe confrontando i nomi della classe e non i *class object*. In questo caso sarebbe semplice ingannare il controllo fornendo classi con un nome uguale a quella originale per rimpiazzarla.

Gli errori qui presentati sono chiaramente solo una parte di tutti quelli possibili, ma offrono una panoramica dei vari difetti che possono portare ad una penetrazione della sicurezza di un sistema.

2.2 Type confusion

E' stato già detto che il Java utilizza una programmazione fortemente tipizzata. La *type safety* di Java è una delle garanzie di sicurezza di tutto l'ambiente. Riuscire a sovvertire questa caratteristica, ossia riuscire a far risultare un oggetto di un tipo piuttosto che di un altro, consente di rompere la sicurezza dell'intero sistema. Generalmente, l'attacco prevede la creazione di un tipo personalizzato che consenta di fare una serie di operazioni particolari come ad esempio accedere a campi che in un'altra classe sarebbero privati. Viene poi eseguito, in qualche modo, un *casting* illecito dell'oggetto della classe sotto attacco nel tipo personalizzato precedentemente definito, e, quindi, è possibile accedere ai campi privati di tale oggetto, poiché Java non li riconosce più come tali.

Il controllo dei tipi di variabile fa parte della struttura stessa del codice Java e, come anticipato nella prima sezione di questo testo, il controllo di tali strutture è eseguito dal *Bytecode Verifier*.

Il *Bytecode Verifier* esegue quattro passi principali per controllare il *bytecode* prima di passarlo ad un *ClassLoader* (In realtà il quarto passo è eseguito dopo il caricamento della classe, quindi, il *Bytecode Verifier* non opera precisamente prima del *ClassLoader*, ma collabora con questo):

1. Eseguce controlli strutturali sul *bytecode*. Controlla che il *bytecode* contenga effettivamente la descrizione di una classe e che la lunghezza del file sia corretta. Sostanzialmente si controlla che il *bytecode* sia concorde alla definizione dei *.class file* di Java.
2. Il secondo passo si effettua dopo il *link* della classe. Eseguce controlli di tipo semantico sul rispetto dei tipi. Ogni descrittore di metodo, di campo, ecc. viene controllato per garantire che il tipo dichiarato corrisponda con quello reale. In aggiunta vengono controllati anche i meccanismi di ereditarietà (se ci sono classi definite specializzazioni di classi *final* o metodi *final* di cui viene fatto *override*).
3. Il passo tre è il più complesso poiché controlla il flusso di istruzioni generato dal *bytecode*, esaminando singolarmente le istruzioni di ogni metodo. L'esame è condotto analizzando il flusso dati di ciascuna sequenza di istruzioni presente nel *bytecode*, e simulando il *path* di esecuzione di ogni istruzione di un metodo.
4. Il quarto passo viene eseguito a *runtime* prima del processo di *dynamic linking* in cui i riferimenti simbolici presenti nel *bytecode* vengono risolti nei riferimenti diretti alle classi. Il *Bytecode Verifier* controlla che il riferimento risolto sia corretto e che la classe, il metodo o il campo puntato effettivamente esista e che la classe esistente sia effettivamente compatibile con il riferimento simbolico. E' molto importante notare che quando la JVM incontra una classe per la prima volta provvede al suo caricamento e all'esecuzione dei controlli, inoltre, il riferimento diretto alla classe viene conservato in memoria. Quando la JVM incontra una classe già caricata precedentemente, verifica la presenza del riferimento diretto in memoria e, se presente, non effettua di nuovo il controllo ma provvede a caricarla direttamente.

Nella maggior parte delle realizzazioni delle JVM, il controllo del *Bytecode Verifier*, per risparmiare tempo e incrementare le prestazioni, viene eseguito soltanto una volta prima del caricamento della classe. Eccezione a questa pratica è fatta per i controlli sulle istruzioni di accesso agli *array*, che vengono eseguiti anche a *runtime*.

Se viene rilevato un difetto di realizzazione nel *Bytecode Verifier*, che permette il *casting* fra due variabili di diverso tipo, è possibile portare a compimento un *type confusion attack*.

Scelta una classe obiettivo, l'attacco consiste nel creare una classe *fake* della prima. La classe *fake* è praticamente una copia esatta della prima classe, ad eccezion fatta che i campi privati della classe originale sono resi pubblici in quella falsa. In questo modo, facendo il *casting* della classe originale in quella falsa, è possibile accedere a tali campi e leggerli o modificarli a piacimento.

Nel seguito è presentato del codice di esempio, in cui si può notare che il campo *sec* della classe originale è privato, mentre in quella falsa è reso pubblico.

```

public class Original{
    private Security sec;

    private void methodA(){...}
    public void methodB(){...}
}

public class FakeOriginal{
    public Security sec;

    private void methodA(){...}
    public void methodB(){...}
}

```

Se un attacco del genere andasse a buon fine, sarebbe possibile, ad esempio, cambiare i valori di un oggetto che contiene *password* o politiche di sicurezza, garantendo di fatto il totale controllo del sistema all'attaccante.

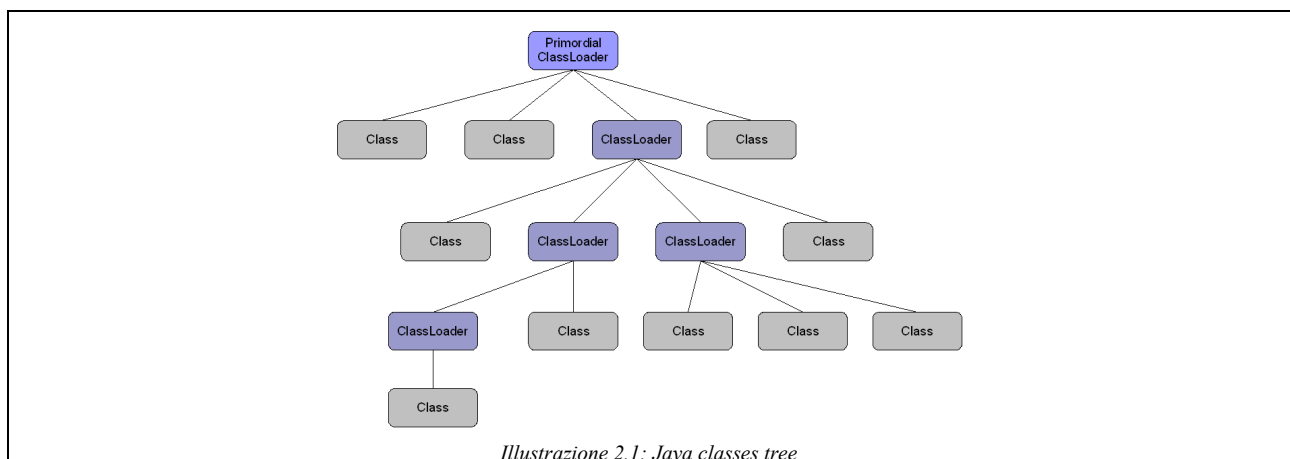
Difetti nel *Bytecode verifier* sono stati individuati diverse volte negli anni, ed in diverse implementazioni della JVM, questa tipologia di attacchi è quindi tutt'altro che improbabile.

2.3 Class spoofing

Un altro classico tipo di attacco è quello che tenta di sostituire una classe all'interno del sistema con un'altra appositamente creata. In questo genere di attacchi, il sistema è convinto di colloquiare con una classe sicura, ma in realtà sta operando con una classe costruita ad hoc per portare un attacco.

In Java il caricamento delle classi è compito dei *ClassLoader*. Tutte le classi caricate da un *ClassLoader* non vedono le classi caricate da un altro *ClassLoader*, in altre parole un *ClassLoader* definisce un proprio *namespace*. L'utilizzo di *namespace* separati è un metodo di sicurezza ulteriore che consente, fra le altre cose, di definire due classi con lo stesso nome, purché siano caricate da *ClassLoader* differenti, ossia, purché vengano definite in *namespace* diversi. Ogni *ClassLoader* assicura che un'istanza di una certa classe sia caricata solo una volta in seno a quel *ClassLoader*.

Il meccanismo appena descritto disegna un albero delle classi, in cui ogni *ClassLoader* è un nodo che può avere figli, ogni altra classe è un nodo dell'albero senza figli, la radice dell'albero è il *PrimordialClassLoader* (Illustrazione 2.1).



Come visto in precedenza, seppur con diverse limitazioni, un *ClassLoader* può essere ridefinito dall'utente. E, quindi, si può immaginare di operare in modo da creare le condizioni per un attacco.

Sostanzialmente si punta a creare confusione fra le classi appartenenti a diversi *namespace* ma con nomi uguali.

Supponiamo, ad esempio, di utilizzare le due classi definite nel seguito:

```
public class Spoofed{
    public Object var;
}
public class Spoofed{
    public MyClass var;
}
```

Vogliamo tentare di creare confusione fra queste due classi, in modo che il sistema ne utilizzi una quando è invece convinto di utilizzare l'altra.

Per farlo è necessario definire due ulteriori classi:

```
public class Dummy{
    Spoofed value;
}
public class Bridge{
    public void A(Dummy dummy, MyClass myclass){
        dummy.value = myclass;
    }
}
```

Supponendo di utilizzare per l'attacco due *ClassLoader*, CL1 e CL2, lo scenario è il seguente:

1. Vengono caricate dal CL1 le classi *Dummy* e *MyClass*. Il caricamento della classe *Dummy*, chiaramente, implica il caricamento anche della classe *Spoofed*.

Simultaneamente viene caricata dal CL2 la classe *Bridge*, che a sua volta contiene all'interno le dichiarazioni delle classi *Dummy* (e quindi anche *Spoofed*), *MyClass*.

La dichiarazione delle classi è fatta in modo che le classi *Dummy* e *MyClass* abbiano la stessa rappresentazione interna nella JVM, mentre la classe *Spoofed* nel *namespace* del CL1 è definita con un campo di tipo *Object*, nel *namespace* del CL2 è definita con un campo di tipo *MyClass*.

Per fare questo, il metodo *loadClass* del CL2 è ridefinito nel seguente modo:

```

public synchronized Class loadClass(String name, boolean resolve){
    Class c = null;
    if(name.equals("Dummy") return dummy_cl;
    else if(name.equals("MyClass") return myclass_cl;
    else if(name.equals("Spoofed") c=defineClass("Spoofed",spoofed_def ...);
    else c = findSystemClass(name);
    if(resolve) resolveClass(c);
    return c;
}

```

Il risultato che si ottiene è quindi una condivisione delle classi *MyClass* e *Dummy* fra i *namespace* di CL1 e CL2, ma una differente definizione per la classe *Spoofed* (Illustrazione 2.2).

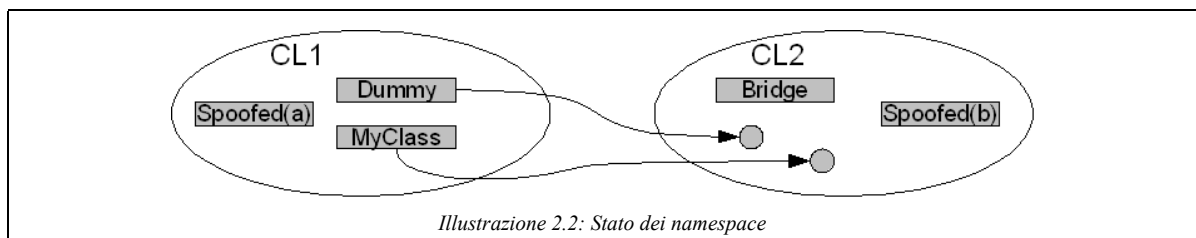


Illustrazione 2.2: Stato dei namespace

2. Nel *namespace* del CL2 vengono create le istanze delle classi *Bridge* e *Dummy*.
3. Se si chiama il metodo A della classe *Bridge*, passando come parametri la classe *Dummy* e la classe *MyClass*, avremo un *casting* di un oggetto *MyClass* in un oggetto *Object*, creando di fatto *type confusion*.

Il risultato appena esposto viene raggiunto poiché nel *namespace* di CL1 la classe *Dummy* ha un campo di tipo *Spoofed* così come nel CL2. Tuttavia, per quanto le classi *Dummy* siano effettivamente uguali, le classi *Spoofed* non sono tali nei due *namespace*. Inoltre, la classe *Dummy* ignora l'esistenza di diverse versioni della classe *Spoofed* e, quindi, quando riceve come argomento una classe *Spoofed* per impostare il valore del suo campo, non solleva eccezioni di tipo non compatibile.

Molte implementazioni della JVM, per evitare questa problematica non consentono di fare *overriding* del metodo *loadClass* del *ClassLoader*. Tuttavia, alcune implementazioni della JVM da parte di Microsoft non impongono questa limitazione.

2.4 Tecniche di aumento dei privilegi

Le tecniche presentate fino ad ora hanno lo scopo di aggirare le meccaniche del sistema per compiere operazioni altrimenti vietate. Chiaramente, creare un *type confusion* fra due classi create da un utente ha poco senso e nessuna utilità. Un attacco del genere viene invece generalmente usato prendendo come obiettivo le classi di sistema. Lo scopo è accedere alle parti protette di tali classi per poi concedere all'attaccante privilegi aggiuntivi sul sistema.

Le tecniche presentate nei precedenti paragrafi sono quindi un semplice mezzo per giungere allo scopo centrale dell'attacco che è l'aumento dei privilegi sul sistema.

Effettuare un aumento dei privilegi è un'operazione tutt'altro che banale, e, quindi, non è possibile definire un metodo unico di compierla. Nel seguito si presenta un approccio concettuale alle operazioni necessarie per ottenere maggiori privilegi. In particolare partiremo dall'ipotesi che è presente un difetto in un qualche componente della JVM e che sia stato trovato il metodo di fare l'*exploit* di tale difetto, tramite un attacco di tipo *type confusion*.

Il codice presentato è da intendersi puramente concettuale e non rappresentativo di una reale implementazione, per quanto si rifaccia ad esempi concreti (nella fattispecie alla JVM del *web browser* Netscape 4.x, esempio preso da [9]):

```

PrivilegeManager pm = PrivilegeManager.getPrivilegeManager();

VerifierBug bug = new VerifierBug();
MyPrivilegeManager mpm = bug.cast2MyPrivilegeManager(pm);

Target target = Target.findTarget("SuperUser");
Privilege priv = Privilege.findPrivilege(Privilege.ALLOWED,
                                         CPrivilege.FOREVER);

PrivilegeTable privtab = new PrivilegeTable();
privtab.put(target,priv);

Principal principal = PrivilegeManager.getMyPrincipals()[0];
mpm.itsPrinToPrivTable.put(principal,privtab);

try {
    ClassLoader cl=getClass().getClassLoader();
    Class c1.loadClass("Beyond");
    c.newInstance();
} catch (Throwable e) {}

```

Il codice presentato è concettualmente molto semplice. Viene presa un'istanza della classe che controlla i privilegi di sistema, e, sfruttando un *bug*, si effettua un attacco di *type confusion* facendo un *casting* illecito della classe di sistema ad una classe definita dall'attaccante (presumibilmente tale classe sarà uguale a quella di sistema con la differenza che tutti i campi sono dichiarati *public*).

Le righe successive non fanno altro che risalire all'utente che corrisponde all'attaccante, definire una tabella dei privilegi, con tutti i privilegi assegnati, ed associarla a tale utente.

Per quanto l'utente attaccante abbia ora tutti i privilegi, sono necessarie ancora diverse azioni. Infatti, il sistema conserva ancora traccia dei vecchi privilegi in tutti i *frame* dello *stack* della classe che ha condotto l'attacco. E' quindi necessario che l'attaccante definisca una nuova classe, dopo aver ottenuto i privilegi, in modo che tale classe faccia effettivamente riferimento ai nuovi privilegi.

Questo spiega la presenza del successivo blocco *try-catch* che carica la classe *Beyond*. In realtà, ancora non è completa l'acquisizione dei privilegi, poiché dopo averli assegnati, è molto spesso necessario attivarli. Si adopera solitamente un'apposita funzione come quella mostrata nel seguito:

```

PrivilegeManager.enablePrivilege("SuperUser");

```

Compiuta quest'ultima operazione, l'attacco è portato a termine con successo, e l'attaccante prende il controllo del sistema bersaglio.

Conclusioni

L'ambiente Java è nato per garantire la sicurezza delle applicazioni. Il risultato ottenuto, da un'analisi compiuta dopo diversi anni di pieno utilizzo di Java, è sufficientemente vicino a quelli che erano i propositi iniziali che spinsero alla creazione dell'ambiente.

Tuttavia, non è un insegnamento recente quello che recita “nulla è perfetto”, ed anche Java risente di alcuni difetti che ne compromettono la sicurezza. In queste pagine sono state presentate le problematiche principali che si sono presentate nel tempo e che effettivamente hanno causato problemi di sicurezza e, talvolta, danni. Non è presente una lista delle attuali debolezze di Java e dei modi per sfruttarle per ovvi motivi, inoltre, tali informazioni sono chiaramente riservate e fin quando problemi del genere non vengono risolti, è sempre preferibile non divulgarli.

In definitiva Java si presenta più sicuro di molti altri ambienti, quantomeno perché garantisce l'assenza di molti degli errori che nel passato hanno causato falle alla sicurezza. Il sistema è ben progettato ed efficiente nella sua gestione. Le problematiche che si sono verificate nel tempo sono il frutto di errori, talvolta anche banali, di realizzazione di alcuni dei componenti. Errori spesso anche molto difficili da individuare. In conclusione, Java si presenta come un ambiente robusto, le cui rare problematiche risiedono in sporadici errori di realizzazione delle componenti che curano la sicurezza. Lo sforzo per compiere un attacco è tuttavia alto e non alla portata di molti.

Bibliografia

- [1] Anderson – *Security Engineering* – WILEY 2001
- [2] Tomlinson – *Rudimentary treatise on the construction of locks* – 1853
- [3] Saltzer, Schroeder – *The Protection of Information in Computer Systems* – 1975
- [4] Sun Microsystems – *Java Security Architecture, Revision 1.2* – 2002
- [5] Fraticelli – *JDK 1.2: Modello di Sicurezza* – www.mokabyte.it 1998
- [6] Puliti – *Gestione della sicurezza con il JDK 1.2* – www.mokabyte.it 1998
- [7] Sun Microsystems – *Java 2 Platform Documentation*.
- [8] McGraw, Felten – *Securing Java* – WILEY 1999
- [9] LSD Research Group – *Java and Java Virtual Machine security vulnerabilities and their exploitation techniques* – 2002
- [10] Gollmann – *Computer Security 2nd edition* – WILEY 2006
- [11] Sun Microsystems – *Java security overview* – 2005
- [12] Casati – *Tecnologie per l'esecuzione di codice distribuito attraverso WWW* – 2002

Febbraio 2007
Università Federico II di Napoli
Sicurezza ed Affidabilità dei Sistemi Informatici