Università degli Studi di Perugia
LS Informatica

Valentina Franzoni

I NUOVI PARADIGMI PER I L CONTROLLO DEGLI ACCESSI

Seminario per il corso di Sicurezza Informatica
(Prof. S. Bistarelli)

PREMESSA

Internet e il Web hanno portato un gran numero di persone non esperte di sicurezza in contatto con nuove applicazioni di IT.

In molte applicazioni, come in quelle per il commercio elettronico, c'è spesso bisogno di permettere l'accesso al sistema ad entità sconosciute.

Siamo davanti a un caso di un cambiamento considerevole del modo in cui Internet viene usato e dobbiamo domandarci se i vecchi paradigmi siano adatti alle nuove necessità o se siano necessarie nuove politiche (*policy*), dove non sia sufficiente un parziale rafforzamento dei vecchi meccanismi di sicurezza.

Certamente ad oggi questa fase di transizione è ancora *in fieri* e dunque, anche dove sono emerse nuove concezioni della situazione, è ancora necessario sperimentare bene prima di trarre conclusioni su quali siano le strategie da seguire, per ottimizzare la sicurezza in questi nuovi ambienti.

Per la nostra trattazione, ci baseremo principalmente sui nuovi meccanismi di controllo degli accessi basati sul codice, per poi analizzare alcuni semplici esempi di implementazione per un sistema valido e completo, sebbene sempre migliorabile.

INTRODUZIONE

Il primo paradigma per l'accesso a computer distribuiti è nato con l'architettura client-server: un client vuole effettuare un calcolo su un server e il server lo autentica per proteggere se stesso.

Con l'emergere del *World Wide Web*, questa architettura è cambiata sotto vari aspetti. La computazione dal server passa al client: quando un client cerca una pagina web, il browser esegue i programmi integrati nella pagina stessa.

Anche le modalità dell'accesso sono cambiate, dal momento che non si chiede più di essere autorizzati solo alle operazioni fondamentali di lettura, scrittura ed esecuzione, ma ad eseguire interi programmi: i server eseguono script che vengono dai client, mentre i client ricevono applet dai server e possono archiviare il loro stato di sessione nei cookie. Entrambi devono essere protetti.

La separazione tra programmi e dati non esiste più. Il codice eseguibile, come le applet, è integrato in documenti che creano pagine interattive in grado di processare l'input dell'utente. Poiché la computazione è passata al lato client, i server possano mantenere libere le risorse, per fornire allo stesso tempo più servizi.

In questa situazione, spesso il client deve prendere decisioni sulle politiche di sicurezza e gli utenti diventano amministratori di sicurezza.

Quando il software era distribuito tramite floppy disk, accettare floppy da sorgenti arbitrarie poteva essere causa di infezioni. La tipologia di problema per il codice mobile è la stessa: i client devono essere protetti dal codice ed esso deve essere protetto dai client sui quali viene eseguito.

Il Web non ha creato dunque problemi di sicurezza totalmente nuovi, ma ha introdotto un cambiamento del contesto, per cui è necessario ripensare i paradigmi di controllo degli accessi.

IL CONTROLLO DEGLI ACCESSI BASATO SULLE IDENTITA'

Il soggetto (*principal*: utente o servizio) manda una richiesta d'accesso che deve essere autenticata. Le politiche (*policy*) di sicurezza autorizzano il soggetto ad accedere ad un oggetto.

Il modello originario prevedeva l'assunzione che i soggetti corrispondessero a identità conosciute: su questo si basa l'IBAC (*Identity-Based Access Control*), implementato in Unix e Windows, che ha origine in organizzazioni chiuse come università o laboratori di ricerca. Questo controllo degli accessi prevede per definizione che le identità delle persone siano verificate, poiché le regole di accesso sono locali: tali regole sono archiviate nelle ACL (*Access Control List*) insieme agli oggetti stessi.
Le credenziali sono le UID (*User ID*) in Unix e le SID (*Security ID*) in Windows. L'esecuzione delle regole è centralizzata e il reference monitor non deve consultare terze parti per prendere una decisione: le politiche di sicurezza risultano automatizzate.

IL CONTROLLO DEGLI ACCESSI BASATO SUL CODICE

Nel controllo degli accessi basato sul codice (*Code-Based Access Control*) abbiamo accesso anche senza autenticazione o, meglio, l'autenticazione è vista come verifica di ogni credenziale (*evidence*) fornita esternamente ed associata ad una richiesta di accesso.
Possiamo però associare anche elementi locali con una richiesta. Ad esempio, nella sessione corrente potremo dover prendere decisioni sull'accesso in base all'ora e alla data corrente: l'autenticazione dovrà quindi prevedere anche la verifica di una tale associazione.

In un secondo momento, dovremo trovare se una delle nostre policy si applica alla richiesta e controllare se le credenziali disponibili sono sufficienti a soddisfarla, cioè se tutti i permessi richiesti sono inclusi tra i permessi garantiti. Questa fase può essere ancora chiamata "autorizzazione", ma non significherà più l'identificazione di chi ha i titoli per accedere ad una risorsa, ma di quali credenziali sono richieste per l'accesso.

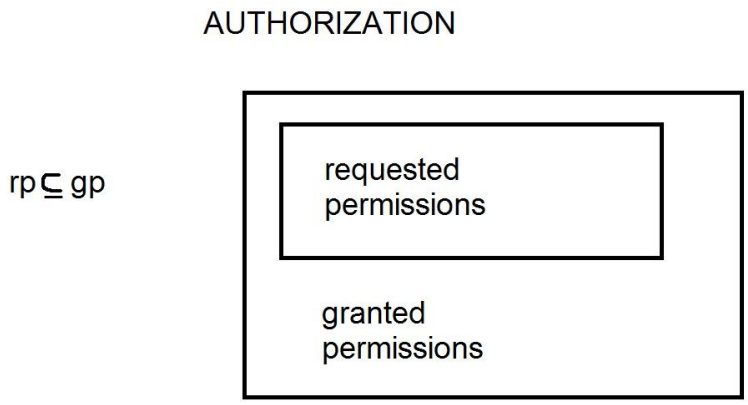

Figura 1: Autorizzazione.

Nella pratica, un PAP (*Policy Administration Point*) crea le policy di accesso; un PDP (*Policy Decision Point*) valuta quale politica è applicabile e prende una decisione sull'autorizzazione; un PEP (*Policy Enforcement Point*) applica il controllo degli accessi, mandando una richiesta al PDP ed eseguendo la decisione ricevuta; un PIP (*Policy Information Point*) fornisce le credenziali al PDP.

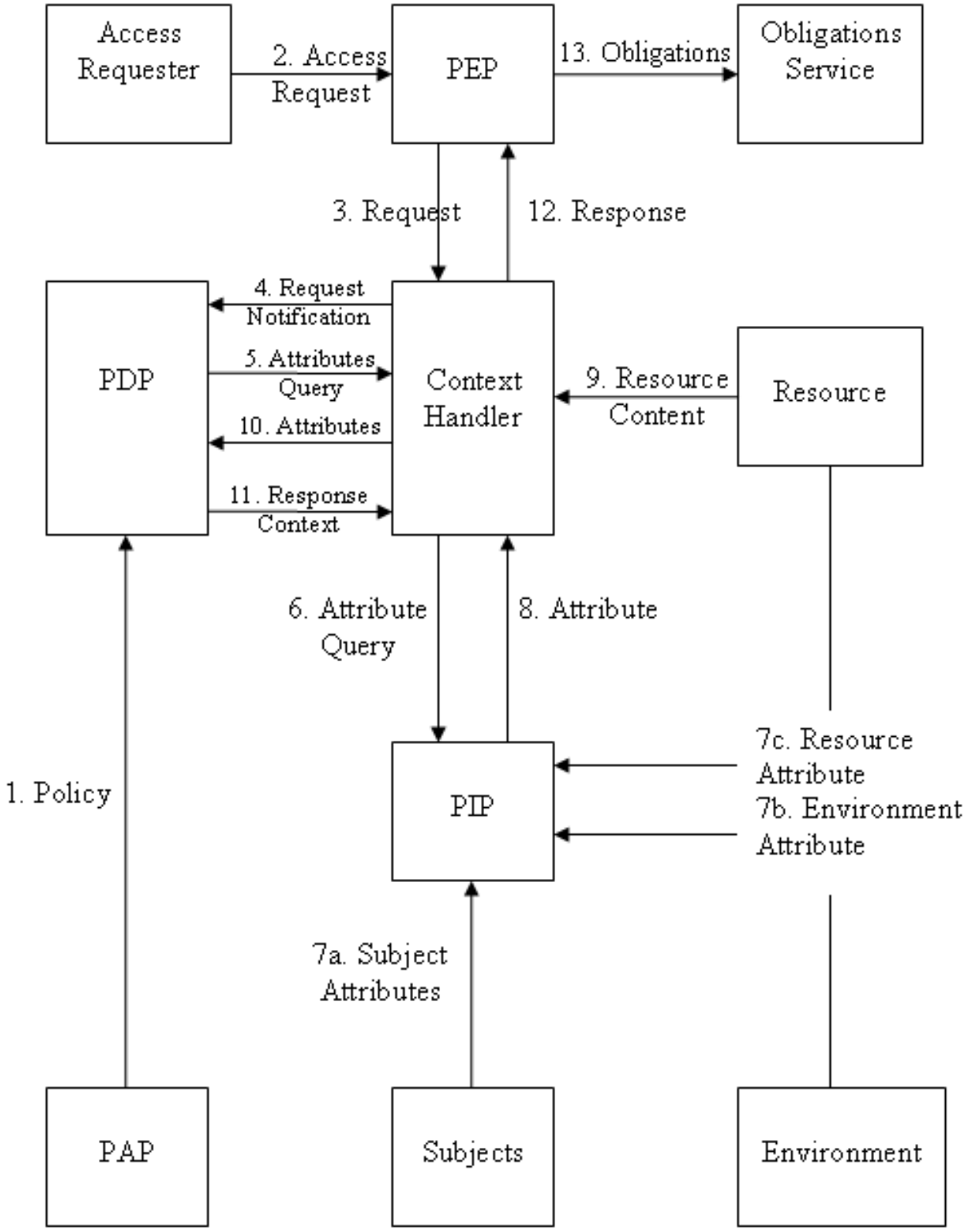

Figura 2: flusso di dati di XAMCL: il PDP controlla che il set di credenziali fornito dal PIP soddisfa le richieste del PEP secondo la politica locale di accesso del PAP.

UN AMBIENTE APERTO

Siamo passati da un sistema chiuso a un ambiente aperto (*Open Environment*).
Le entità con cui abbiamo a che fare non solo non saranno sempre identificabili o localizzabili fisicamente, ma potrà anche verificarsi la situazione in cui non abbiamo l'autorità giurisdizionale per identificarle o intervenire: in questo caso l'identificazione degli utenti non è utile per le politiche di sicurezza.
Allo stesso tempo, i *reference monitor* non ricevono solo richieste di lettura/scrittura/esecuzione, ma interi programmi (*applet*, *script*).

La *Java Sandbox*, (il sistema di sicurezza di Java) mostra il primo modello per controllare gli accessi basandosi sul codice anziché sulle identità. Le credenziali da controllare sono:

- il luogo d'origine del codice, con la possibilità di impostare politiche differenti per codice archiviato localmente o in remoto, ovvero sulla base dell'URL da cui proviene;
- la firma del codice, cioè se l'autore è certificato e da chi, per assegnargli eventualmente diversi privilegi;
- l'identificazione del codice in base a una lista conosciuta, tramite il controllo di un hash, in particolare per le applicazioni di home banking;
- la verifica di particolari proprietà del codice, come i file che può scrivere i siti a cui si può connettere: in questo caso l'autore dovrà fornire gli elementi richiesti e il reference monitor avrà il compito di controllarli al momento della decisione sull'accesso.

STACK INSPECTION

Come nei sistemi IBAC i soggetti rappresentano gli utenti, nei sistemi CBAC i soggetti possono essere parti di codice.
Quando i soggetti sono più d'uno, come nel caso di un software che ne richiama un altro, bisogna decidere quali permessi concedere: se quelli associabili all'uno o quelli associabili all'altro. Come sempre la tattica è quella di rimanere più rigidi possibile, ovvero di concedere solo le autorizzazioni che risultino assegnabili sia all'uno che all'altro soggetto. Per analizzare queste credenziali, viene effettuato uno *stack walk*, che restituisce l'intersezione di tutte le funzioni richiamate nello stack.

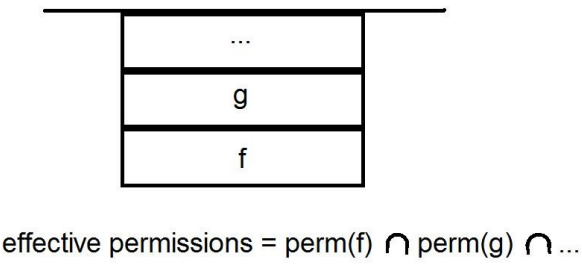

Figura 3: Stack Walk.

Sia la *Java Virtual Machine* che il *.NET Common Language Runtime* (i verificatori di *bytecode* integrati nei due linguaggi) effettuano questo tipo di procedura, con la possibilità per una funzione di delegare esplicitamente ad un'altra i propri permessi.

Lo stack walk però non è sufficiente a garantire una corretta gestione dei permessi. In alcuni casi, infatti, non tutti i parametri rilevanti per la sicurezza sono sempre memorizzati nello stack, come

per la ricorsione di coda: un chiamante di fiducia potrebbe anche vedersi cancellare un permesso già affermato.
Per evitare questo inconveniente è efficace un'implementazione che tenga traccia dei diritti in maniera proattiva: i permessi statici correnti verranno associati con ogni unità di codice al momento del caricamento e aggiornati automaticamente con l'intersezione dei due insiemi. Questo metodo viene chiamato HBAC (*History-Based Access Control*).

LA SICUREZZA IN JAVA

Java è un linguaggio ben pensato per la sicurezza, perché impedisce ai programmatori di causare alcuni gravi problemi di protezione.

Innanzitutto Java implementa la sicurezza dei tipi (è *type safe*) a tempo di compilazione. Il sorgente viene compilato in un bytecode simile a linguaggio assembly ma indipendente dalla macchina, che viene archiviato in classi. Per quanto riguarda le applicazioni web, il bytecode viene interpretato nel client da una JVM (*Java Virtual Machine*) integrata nel browser e dipendente dalla piattaforma, che lo trasforma in istruzioni specifiche per la macchina.

Il *Byte Code Verifier* della JVM è in grado di verificare il bytecode tramite controlli sintattici, verifiche di teoremi e analisi dei flussi di dati, implementando un'ulteriore verifica dei tipi prima che il codice sia eseguito. In questa fase viene controllato che i file siano nel formato corretto e che non ci sia uno stack overflow. I metodi vengono chiamati coi loro argomenti ed i tipi appropriati e si controlla che anche i risultati siano dei tipi previsti, senza conversioni illegali di tipi. I campi privati non devono essere accessibili dall'esterno dell'oggetto.

Il *Class Loader*, attivato dalla JVM, carica le classi aggiuntive richieste.
Il *Bootstrap Class Loader* carica le classi che formano il core della JVM, in dipendenza dalla piattaforma e non ha limitazioni di accesso al filesystem e alle risorse di sistema. L'*Extension Class Loader* carica le classi dai pacchetti opzionali installati. L'*Application Class Loader* si occupa delle classi definite dall'utente.
Quando un Class Loader carica la classe, le assegna un dominio protetto. I controlli dei collegamenti vengono effettuati dalla JVM una sola volta, a differenza di quelli a tempo di esecuzione.
Ogni classe sarà caricata in un Class Loader con un nome differente, ma è possibile anche definire politiche personalizzate o assegnare particolari privilegi a classi che provengono da una certa fonte.
Al termine del controllo, il Class Loader invoca il Byte Code Verifier.

Il *Security Manager* è il reference monitor di Java, che impone un controllo di accesso alle risorse di sistema e protegge le aree di memoria a cui il programma può accedere invocando l'*Access Controller*, che implementa un algoritmo di decisione comune a tutti i permessi ed esegue lo stack walk.

Una volta controllato, il bytecode può essere o non essere eseguito secondo la policy, in base alla quale si confrontano le credenziali fornite con quelle richieste. Una volta creati, i domini protetti sono file che non possono essere cambiati né sovrascritti: ogni file contiene almeno un valore di chiave e una lista di permessi, assegnabili a una lista di credenziali o a una parte di codice o ad una lista di firmatari.
Java infatti permette anche la firma del codice, che può costituire una credenziale in base alla quale il client può decidere di effettuare delle scelte anziché altre.

Questo è un esempio di implementazione di una policy in Java:

```
package javasec.samples.ch05;
import java.security.*;
import java.util.*;

public class MyPolicy extends Policy {

// This inner class defines a simple set of permissions:
// either everything is allowed (the implies( ) method always
// returns true, and the collection contains an AllPermission
// object) or everything is prohibited (the implies( )
// method always returns false and the collection is empty).

   static class SimplePermissions extends PermissionCollection {
       boolean allow;
       Permissions perms;

       SimplePermissions(boolean b) {
              allow = b;
              perms = new Permissions( );
              if (allow)
                     perms.add(new AllPermission( ));
       }

       public void add(Permission p) {
              if (isReadOnly( ))
                     throw new SecurityException(
                     "Can't add to this collection");
                     perms.add(p);
       }

       public Enumeration elements( ) {
              return perms.elements( );
       }

       public boolean implies(Permission p) {
              return allow;
       }

   }

   // We never change the policy

   public void refresh( ) {}

   // If the code was loaded from a file, return a collection
   // that implies all permissions. Otherwise, return an
   // empty collection (one that implies no permissions).

   public PermissionCollection getPermissions(CodeSource cs) {
       if (cs.getLocation().getProtocol( ).equals("file"))
              return new SimplePermissions(true);
              return new SimplePermissions(false);
   }
}
```

Per quanto riguarda le *Applet*, il sistema di sicurezza di Java non permette che accedano al filesystem dell'utente, né che ottengano alcuna informazione su nome, password, indirizzo e-mail e configurazioni di macchina. Non possono riconfigurare la JVM, né sostituire le loro classi alle classi del sistema. In pratica le applet possono connettersi solo con il server dal quale provengono.

Inoltre Java non permette l'uso di puntatori, in modo che non sia possibile manipolare direttamente gli indirizzi di memoria.
Tutto questo sistema di sicurezza costituisce la *Java Sandbox* di *Java 2*.

Nella versione *Java 1*, c'era una divisione molto netta, tra codice *unrestricted* e *restricted*: le applet non firmate erano considerate restricted, al contrario di quelle firmate e del codice locale. In questo modo non erano possibili policy flessibili e alcuni programmi, come quelli di home banking, non potevano essere autenticati.
Nel 1999 fu scoperto inoltre un bug molto grave nella JVM di Netscape Communicator: in alcune circostanze la JVM non riusciva a verificare tutto il bytecode. Questo avrebbe permesso a eventuali hacker di eseguire codice che violasse le politiche di sicurezza dei tipi, creando un *attacco di confusione dei tipi*. Dal momento che le variabili vengono inizializzate prima di essere usate, con questo attacco si riesce ad associare in memoria due etichette di tipi incompatibili allo stesso oggetto. L'unica soluzione valida era disabilitare la JVM nel browser.

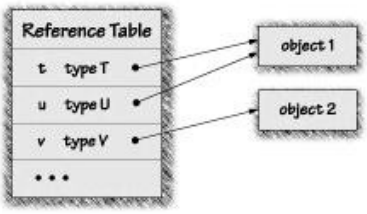

Figura 4: attacco di confusione di tipi.

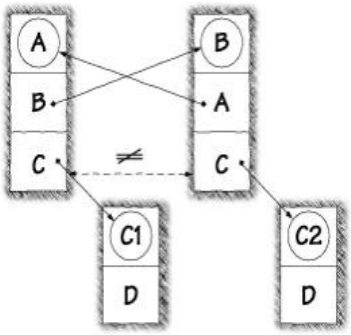

Figura 5: le classi A e B si riferiscono al nome di classe C. A alloca un oggetto di tipo C e lo passa a B, che vi accede in base alla propria definizione.

LA SICUREZZA IN .NET

.NET è, come Java, un linguaggio di programmazione type-safe. In .NET la gestione della sicurezza è affidata principalmente al CLR (*Common Language Runtime*), un sistema architetturalmente simile alla JVM, che supporta diversi linguaggi come il Visual Basic, C++, C#, etc., dei quali il CLS (*Common Language Specification*) definisce i requisiti.

Il codice scritto in uno dei linguaggi .NET viene compilato in un codice MSIL (*Microsoft Intermediate Language*), simile al bytecode di Java, che viene passato al compilatore JIT (*Just In Time*) appena prima dell'esecuzione. Il CLR carica ed esegue il codice, esegue i controlli di sicurezza e la *garbage collection*.

Il codice *managed* viene compilato per girare sul framework .NET ed è controllato dal CLR, mentre il codice nativo *unmanaged* viene compilato in linguaggio macchina per l'esecuzione su una

specifica piattaforma hardware. Questo codice non passa per il controllo del CLR e funziona ad un livello più basso. Le chiamate dal codice managed al codice nativo sono critiche per la sicurezza, poiché sono possibili e non garantiscono l'applicazione dei controlli del CLR..

L'unità logica di codice MSIL è l'*assembly* e contiene metadati che includono per ogni classe le informazioni su nome, visibilità, sottoclassi e interfacce implementate, oltre a metodi, campi e proprietà.

Anche .NET implementa il CBAC, assegnando privilegi sulla base dell'identità del codice, stabilita tramite le credenziali degli assembly, calcolate a tempo di esecuzione o già note, come l'URL di provenienza del codice, l'hash dell'assembly o la firma del codice.
Le credenziali possono essere fornite dall'host che lancia l'assembly, come il browser, oppure dal codice *managed* che richiama altro codice, o ancora dallo stesso assembly, che provvede da se stesso a fornire le credenziali richieste, usando classi specifiche per le varie applicazioni. In ogni caso l'assembly non può sovrascrivere le credenziali fornite dall'host.

Come per Java, i permessi vengono archiviati in domini di applicazione isolati, con una chiave aggiunta ai metadati e una firma digitale apposta a tempo di compilazione e poi verificata dal framework .NET per evitare attacchi all'integrità e falsificazione dell'identità (*spoofing*).
I permessi sono assegnati agli assembly in base alle politiche di sicurezza. I permessi richiesti vengono specificati aggiungendoli al codice. A tempo di esecuzione verrà effettuato uno stack walk.

Le condizioni e i permessi associati vengono gestiti tramite gruppi e gerarchie, con quattro diversi livelli di sicurezza: Enterprise, Machine, User e Application, che consistono in set di permessi. Per ogni livello vengono valutate le credenziali richieste e i risultati vengono ancora una volta intersecati, per ottenere l'autorizzazione finale.

I COOKIE

Il protocollo HTTP non prevede di per sé una modalità in cui si possa tenere traccia di dati durante la navigazione nel passaggio tra una sessione e l'altra, poiché tutte le richieste effettuate vengono trattate come eventi indipendenti, anche quando vengono dallo stesso client. Tutti gli strumenti associati hanno dunque bisogno di essere costantemente ripetuti nel tempo.
Per esempio, se per l'accesso a un sito web è necessaria una password, ad ogni click sulla pagina questa password dovrà essere ritrasmessa. HTTP è in grado di mantenere tali dati nel browser per la durata della sessione, ma in caso di interruzione -voluta o accidentale-, come anche quando si vogliano implementare delle transazioni che consistono di più passi, è necessario ricorrere ai *cookie*.

I cookie (in inglese, “biscotti”) non sono altro che file di dati, di solito conservati nel client. Trattandosi di dati e non di programmi, i cookie di per sé non possono violare l'integrità del sistema, né carpire informazioni confidenziali per passarle al server.
Inoltre solo al dominio del server che ne ha richiesto la creazione è data l'autorizzazione a leggerli e quindi il server ha accesso solo ai cookie specifici per il suo dominio.
I cookie possono essere impostati *sessione per sessione*, nel qual caso rimangono in RAM e scadono alla chiusura del browser o in base a un tempo preimpostato; oppure possono essere *permanenti* e dunque risiedere nel disco fisso, in una cartella facilmente identificabile chiamata solitamente “Cookies”, sotto forma di file di testo.

Tre sono i tipi di attacchi ai cookie:

1. da parte del server, che potrebbe violare la privacy del client creandone un profilo;
2. da parte del client, che potrebbe modificare i cookie, ad esempio per avere benefici dal server come per raccolte punti o per accedere a risorse protette (attacco di *cookie poisoning*);
3. da parte di terze parti, che potrebbero impadronirsi dei cookie di un altro utente (attacco di *sniffing*), per poi impersonificarlo con il server (attacco di *spoofing*). Questa combinazione dei due attacchi citati viene chiamata *snarfing*.

Figura 6: frammento di una sessione con l'utilizzo di cookie ricostruita da PeepNet.

Nell'esempio in figura 6 vediamo un frammento di una ricostruzione, fatta dal programma di hacking PeepNet, di una sessione che utilizza l'autenticazione tramite ID per accedere a pagine web personalizzate. Nella voce **Cookie:** vediamo il token fornito nella richiesta HTTP inviata al server. La parte importante è **cuid=**, dove viene riportato l'ID usato per identificare l'utente sul sito www.victim.net . Una volta intercettata questa richiesta, l'hacker potrà creare il proprio account sul sito victim.net e poi sostituire nel proprio cookie la voce **cuid** con quella del cookie intercettato, per assumere l'identità della vittima e accedere alle sue pagine personali.

Le contromisure che si possono adottare per bloccare gli attacchi sono diverse. Alcuni browser permettono di bloccare del tutto i cookie o di eliminarli al termine della sessione di navigazione. Esistono anche programmi specifici, che possono essere impostati per avvisare il client quando il server richiede la creazione di cookie e per monitorare le richieste del server in tal senso.
Quando visitiamo un sito che usa i cookie per l'autenticazione, possiamo anche verificare che venga utilizzato un protocollo SSL o pari per criptare il primo invio di nome utente e password, in modo che non appaiano in chiaro in programmi come PeepNet.
Allo stesso modo anche il server può proteggersi usando la crittografia.

SPKI

Nei sistemi decentralizzati di controllo degli accessi, possiamo proteggere l'integrità delle regole di accesso tramite la crittografia.
SPKI (*Simple Public Key Infrastructure*) supporta gli schemi di autorizzazione che funzionano senza il controllo delle identità.
Il sistema si basa sul postulato che i nomi usati per il controllo degli accessi abbiano essenzialmente una valenza locale e quindi i diritti di accesso debbano derivare non tanto dai nomi quanto da altri attributi. Da questo punto di vista, i nomi sono considerati soltanto puntatori per accedere alle autorizzazioni. Una politica di sicurezza definisce le regole di un dato dominio.
I nomi hanno valenza locale e non sono necessariamente unici globalmente quindi, per riferirci ai nomi da un dominio locale diverso, è necessario avere identificatori globali unici, che saranno coppie di chiavi pubbliche e private generate random. Queste coppie hanno una probabilità vicina alla certezza di essere uniche, come anche l'hash della chiave pubblica: sarà quindi la chiave pubblica, o il suo hash, a costituire l'identificatore unico di un dominio e la principale credenziale per l'accesso.

I diritti di accesso vengono infatti collegati direttamente alle chiavi pubbliche tramite dei certificati, che contengono esplicitamente almeno un emittente e un soggetto e possono includere deleghe, autorizzazioni e condizioni di validità. L'emittente firma il certificato ed autorizza il soggetto, definendo la policy. Il soggetto è tipicamente una chiave pubblica, il suo hash o il suo nome.
La chiave per la verifica del certificato viene archiviata nelle ACL, che non sono firmate e non hanno emittente, poiché vengono archiviate localmente e mai trasmesse.

I diritti d'accesso sono assegnati a chiavi pubbliche: i possessori delle relative chiavi private possono delegare alcuni dei loro diritti ad altri soggetti, emettendo dei certificati per quei diritti.
L'emittente può delegare soltanto i diritti che possiede e specificare se tali diritti possono essere delegati ulteriormente.

L'algoritmo per assegnare le autorizzazioni è espresso in termini di 5-uple nella forma:
<emittente, soggetto autorizzato, delega, autorizzazione, validità> dove **delega** è un flag, **autorizzazione** definisce i diritti specifici per l'applicazione e **validità** è una coppia di date che esprimono l'inizio (non prima di) e la fine (non dopo di) dell'autorizzazione.

Ne forniamo per finire un semplice esempio, che riguarda un'autorizzazione delegabile assegnata in un certificato da un emittente1 ad un soggetto1 e di un'altra da un emittente2 a un soggetto2. Nel codice in figura 7 vediamo anche la seguente verifica della validità globale del certificato, che restituisce l'autorizzazione finale, sempre in forma di 5-upla.

```
Input: <Issuer1,Subject1,D1,Auth1,Val1>
       <Issuer2,Subject2,D2,Auth2,Val2>

IF Subject1 =Issuer2 AND D1 = TRUE
   THEN output <Issuer1, Subject2, D2,
                AIntersect(Auth1,Auth2),
                VIntersect(Val1,Val2)>
```

Figura 7: esempio di SPKI.

BIBLIOGRAFIA

D. Gollmann, *Computer Security* (Wiley 2003)
S. McClure – J. Scambray – G. Kurtz, *Hacker! 4.0* (Apogeo 2003)
S. Oaks, *Java Security 2nd edition* (O'Reilly 2001)
A. De Santis, *Applet Maliziosi* (Università degli Studi di Salerno, 2000)