UNIVERSITÀ DI PISA

FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI

CORSO DI LAUREA SPECIALISTICA IN INFORMATICA

ANALISI DEGLI ATTACCHI CONTRO UNA INFORMATION INFRASTRUCTURE ATTRAVERSO DIPENDENZE ED EVOLUZIONI

Tesi di Laurea Specialistica

Mario Chilosi

Relatore: Prof. Fabrizio Baiardi

Controrelatore: Prof. Fabio Gadducci

ANNO ACCADEMICO 2006-2007

Abstract

L'analisi del rischio è utile per studiare minacce e vulnerabilità di una infrastruttura informatica e per capire come investire un insieme limitato di risorse a difesa delle aree più a rischio.
Un componente dell'infrastruttura è la rappresentazione astratta di una parte del sistema.
Gli utenti del sistema acquisiscono diritti sui componenti per mezzo degli attacchi o delle dipendenze. Le dipendenze riguardano i tre principali attributi di sicurezza: confidenzialità, integrità e disponibilità dei dati; sono usate per modellare l'infrastruttura a più livelli di astrazione, per dedurre strategie di attacchi, o per definire piani per la mitigazione del rischio.
Lo studio delle dipendenze aiuta a condurre alcune fasi dell'analisi del rischio.
In questo studio una dipendenza è una implicazione logica tra due diritti. Ogni dipendenza è caratterizzata dai componenti sorgente, da uno di destinazione e da un attributo di sicurezza per ogni componente. Acquisire un diritto da parte dell'utente significa controllare un particolare attributo di sicurezza di un componente dell'infrastruttura. Le dipendenze note del sistema possono essere sfruttate per aumentare l'insieme dei diritti di ogni minaccia, a partire dall'insieme dei privilegi iniziali e dai diritti acquisiti grazie agli attacchi eseguiti contro il sistema.
Il concetto di dipendenza è una informazione spesso trascurata, mentre nella maggior parte dei casi capire la struttura logica e fisica dell'infrastruttura, e quindi i legami tra i componenti, consente di intervenire tempestivamente per eliminare le vulnerabilità del sistema in conseguenza degli attacchi delle minacce.
Scopo di questa tesi è fornire uno strumento che consente di simulare gli scenari possibili in conseguenza degli attacchi eseguiti dalle minacce contro il sistema.
Partendo da un elenco di informazioni sul sistema, utenti con diritti e risorse iniziali, dipendenze tra gli attributi dei componenti, obiettivi delle minacce, attacchi sfruttabili con i relativi goal, diritti e risorse necessarie per ogni attacco, quello che si vuole ottenere è una simulazione del comportamento del sistema, modellando tutti i possibili attacchi che ogni minaccia è in grado di eseguire contro l'infrastruttura. Più precisamente lo strumento simula, per ogni minaccia, tutte le sequenze di attacchi, dette evoluzioni, eseguiti per raggiungere l'obiettivo prefissato.
Il comportamento del sistema viene rappresentato mediante il grafo delle evoluzioni, i cui nodi rappresentano gli stati del sistema, ossia un insieme di coppie <utente,diritto>, e gli archi corrispondono agli attacchi delle minacce. Una evoluzione utente è un cammino fra due stati, quello in cui la minaccia possiede i diritti iniziali e quello finale in cui ha raggiunto i privilegi di un suo obiettivo.

Keywords: componenti, vulnerabilità, attacchi, minacce, diritti, obiettivi, evoluzioni, stati goal, ipergrafo delle dipendenze, grafo delle evoluzioni.

RINGRAZIAMENTI

Mia Madre, mio Padre, mio Fratello, i miei Nonni...
Grazie per... la Pazienza!
(...Saltuariamente reciproca!)

Tu, che sei andato/a "a scovare" questi umili e personali ringraziamenti,
sicuramente sei annoverato/a tra la stretta cerchia di Amici.
(...O forse no?! ;-p)

Gaspare, Manuel, Mirco, Pietro, Riccardo...
I compagni di studio con i quali ho seguito i corsi e preparato gli esami.

Il mio relatore, prof. Fabrizio Baiardi
per il continuo supporto durante la preparazione e la stesura della tesi.

(aMoilrisChio)

INDICE

Abstract 1

Introduzione 7
Descrizione di base 7
Le dipendenze 7
Le evoluzioni 7
Lo strumento proposto 8
Struttura della tesi 8

CAPITOLO 1 : Analisi del problema 10

1.1 Astrazione dell'infrastruttura 10
1.2 Minacce, vulnerabilità, attacchi 11
1.3 Modellazione dell'infrastruttura 13
1.3.1 L'ipergrafo delle dipendenze 14
1.3.2 Il grafo delle evoluzioni 16

CAPITOLO 2 : Tecnologia impiegata 19

2.1 La scelta della programmazione logica 19
2.2 Prolog: caratteristiche principali 23

CAPITOLO 3 : Sviluppo del software 26

3.1 Definizione dello strumento 26
3.1.1 Input necessari e rappresentazione dell'ipergrafo delle dipendenze 27
3.2 Output ottenuti 29
3.2.1 Rappresentazione del grafo delle evoluzioni 32

CAPITOLO 4 : Simulazioni effettuate 36

4.1 *Caso di studio 1: simulazione di uno scenario base* 36
4.2 *Caso di studio 2: simulazione di uno scenario reale 1* 41
4.3 *Caso di studio 3: simulazione di uno scenario reale 2* 46

CAPITOLO 5 : Conclusioni e sviluppi futuri 50

Appendice: PSC - Prolog Source Code 53

Bibliografia 72

INTRODUZIONE

Descrizione di base

I concetti principali che hanno guidato il lavoro descritto in questa tesi sono le **dipendenze** e le **evoluzioni** di una infrastruttura informatica, i componenti, gli utenti, i diritti e gli attacchi contro il sistema.
Questi concetti hanno fornito le basi per la progettazione e la realizzazione di uno strumento innovativo nel campo della sicurezza informatica il cui scopo è quello di modellare dei piani di attacco per poter studiare il comportamento del sistema in conseguenza di tutti gli attacchi eseguiti da parte delle minacce.

Le dipendenze

Un componente dell'infrastruttura è la rappresentazione astratta di una parte del sistema; esempi possono essere host, server, router o firewall all'interno di una rete, o di una rete di reti, ma anche oggetti, file, database.
Tra questi componenti esistono delle relazioni che riguardano i tre principali attributi di sicurezza: confidenzialità, integrità e disponibilità.
Per confidenzialità (c) si intende la possibilità di leggere dati su quel componente; integrità (i) significa poter scrivere informazioni; il controllo della disponibilità (availability, a) implica la capacità di gestione del componente (i.e. stabilire gli utenti che possono invocare operazioni del componente).

Acquisire un diritto (o privilegio) da parte dell'utente significa controllare un particolare attributo di sicurezza di un componente dell'infrastruttura. La relazione tra componenti è vista come dipendenza quando possedere un diritto implica l'acquisizione di un altro diritto:

- possiedo A;
- so che A implica B;
- allora ottengo B.

Un esempio di dipendenza tra diritti potrebbe essere il seguente:

- un server web (oggetto) gestisce i dati di un form;
- un utente può scrivere dati in questo form grazie al metodo "scrivi_dati1", quindi possiede dei privilegi sul metodo "scrivi_dati1";
- un database (altro oggetto) con un metodo "scrivi_dati2" per scrivere i dati nei record.

L'utente non può scrivere direttamente sul database, quindi non ha esplicitamente il diritto sul metodo "scrivi_dati2". Di fatto, quando l'utente scrive i dati nel form comunicando con il web server, sta scrivendo anche nel database, quindi acquisisce, per proprietà transitiva, il diritto di scrittura nel database.

L'**ipergrafo delle dipendenze** fornisce una descrizione formale del sistema: i nodi rappresentano i componenti dell'infrastruttura, gli iperarchi (il termine indica che una dipendenza può avere più sorgenti, come discusso nel dettaglio in 1.3.1) sono le dipendenze note tra gli attributi di sicurezza dei componenti. A partire da questa rappresentazione si possono studiare gli effetti che un determinato attacco può causare nel sistema, per esempio l'acquisizione di nuovi diritti da parte della minaccia che esegue l'attacco, fino all'eventuale conseguimento di un obiettivo della minaccia.

Le evoluzioni

Le minacce che agiscono nel sistema hanno degli obiettivi, ossia degli insiemi di diritti da raggiungere (diritti goal).

Un attacco può portare al diretto raggiungimento dell'obiettivo da parte della minaccia che lo esegue. In altre situazioni sono necessarie delle catene di attacchi per raggiungere tutti i diritti di un obiettivo della minaccia.
Uno stato del sistema è un insieme di diritti per ogni minaccia, prima e/o dopo l'esecuzione di un determinato attacco. Nel grafo delle evoluzioni i nodi rappresentano gli stati, mentre gli archi rappresentano gli attacchi delle minacce che causano la transizione di stato.
Una evoluzione è il cammino che porta l'utente dallo stato in cui possiede i soli diritti iniziali a quello in cui ha raggiunto l'obiettivo cercato. Una evoluzione evidenzia attraverso quale percorso un utente raggiunge il suo obiettivo, con quanti passi e grazie a quali attacchi.
Il **grafo delle evoluzioni** mostra tutte le possibili evoluzioni di ogni minaccia del sistema.

Lo strumento proposto

Partendo da un elenco di input sul sistema, utenti con diritti e risorse iniziali, dipendenze note tra gli attributi di sicurezza dei componenti del sistema, obiettivi delle minacce, vulnerabilità tra i componenti, attacchi noti e relativo goal, diritti e risorse necessarie per ogni attacco eseguito dalle minacce, si vuole studiare il comportamento delle minacce dell'infrastruttura e, di conseguenza, le situazioni che si possono verificare nel sistema.
Gli input di cui sopra servono per descrivere formalmente l'infrastruttura prima dell'esecuzione degli attacchi da parte delle minacce e sono gli unici dati da inserire manualmente nella simulazione. Da essi è possibile ottenere la rappresentazione dell'ipergrafo delle dipendenze.
La fase di analisi deve prevedere uno studio iniziale del contesto informatico; spesso però non si ottengono risultati significativi proprio perché non si pone attenzione alle condizioni iniziali del sistema, agli attori dell'infrastruttura e alle relazioni che esistono tra di essi.
Per ogni minaccia vengono descritti e formalizzati, attraverso la rappresentazione del grafo delle evoluzioni, tutti gli attacchi eseguiti nel sistema per raggiungere il proprio obiettivo a partire da un insieme di diritti iniziali.
Inoltre si deducono tutte le implicazioni che può avere un attacco contro il sistema, ottenendo tutte le possibili configurazioni del sistema via via che le minacce eseguono gli attacchi.
Una simulazione, quindi, di tutti gli ipotetici scenari dell'infrastruttura che si possono verificare in conseguenza di minacce, dipendenze ed attacchi. Uno strumento utile all'amministratore di rete o all'analista esterno per capire dove si dovrà intervenire.

Struttura della tesi

Nei prossimi quattro capitoli vengono approfonditi i concetti presentati in questa introduzione. La tesi è così strutturata:

- **Capitolo 1: Analisi del problema**
  Vengono definiti tutti i termini e i concetti chiave per la modellazione di una information infrastructure. Vengono inoltre presentati e discussi l'ipergrafo delle dipendenze e il grafo delle evoluzioni.

- **Capitolo 2: Tecnologia impiegata**
  Si espongono le motivazioni principali dell'utilizzo della programmazione logica per lo

  sviluppo dello strumento e si analizzano le caratteristiche principali del linguaggio adottato.

- **Capitolo 3: Sviluppo del software**
  Si descrive nel dettaglio lo strumento software proposto, i moduli e gli algoritmi principali e la rappresentazione dell'ipergrafo delle dipendenze e del grafo delle evoluzioni discussi nel primo capitolo.

- **Capitolo 4: Simulazioni effettuate**
  Vengono presentate tre simulazioni in cui sono analizzate tipiche situazioni di realtà aziendali medio-piccole.

- Seguono considerazioni conclusive, codici sorgenti del progetto e riferimenti bibliografici.

CAPITOLO 1 : ANALISI DEL PROBLEMA

1.1 Astrazione dell'infrastruttura

In questo capitolo vengono definiti tutti i termini usati nella tesi.
Viene presentato un nuovo approccio che aiuti a condurre l'analisi del rischio di un sistema (Risk Analysis), grazie alla realizzazione di uno strumento che simula strategie alternative di attacchi contro una infrastruttura informatica formata da componenti che condividono informazioni e/o risorse.

> **Definizione 1.** Un componente è la rappresentazione astratta di una parte del sistema.
>
> **Definizione 2.** Un diritto (o privilegio) equivale alla possibilità di invocare una o più operazioni del componente.

Il sistema è composto da oggetti, o componenti. La capacità di invocare operazioni genera il controllo di uno o più attributi di sicurezza: confidenzialità (lettura dati), integrità (scrittura dati), disponibilità (controllo del componente).
Componenti e diritti rappresentano una nozione sicuramente semplificata rispetto al significato che potrebbero assumere nella realtà, nel nostro contesto ciò che importa è solo descrivere le relazioni tra componenti e, quindi, le implicazioni tra diritti.

La prima modellazione presentata in questa tesi è la rappresentazione di tutte le dipendenze tra i componenti del sistema (paragrafo 1.3.1). Considerare il sistema nel suo complesso, senza tralasciare nessuna informazione, è la vera difficoltà dell'analista.
In seguito non si parlerà più di dipendenze fra componenti, ma direttamente di dipendenze fra diritti, dato che, per come lo abbiamo definito, un diritto è la possibilità di invocare una operazione del componente generando il controllo di uno o più attributi di sicurezza.

Il concetto di **dipendenza** tra diritti può essere così riassunto:

- una minaccia può usufruire del diritto A o perché appartiene al suo insieme dei diritti iniziali o perché viene raggiunto tramite l'esecuzione di un attacco;

- possedere un diritto (o in alcuni casi un determinato insieme di diritti), può consentire l'acquisizione di un altro diritto tramite dipendenza: se la minaccia U possiede il diritto A e, dalle dipendenze note del sistema, un secondo diritto B dipende da A, allora U è in grado sfruttare sia il diritto A che il diritto B; in altri termini, a partire dal solo diritto iniziale A, la minaccia acquisisce, per proprietà transitiva, il diritto B.

La seconda modellazione presentata è la rappresentazione di tutti gli attacchi che le minacce eseguono contro il sistema per raggiungere l'obiettivo prefissato. Questo aspetto viene ampiamente discusso in 1.3.2.

1.2 Minacce, vulnerabilità, attacchi

Uno dei punti principali per la definizione della politica di sicurezza di un sistema è la gestione dei diritti di ogni utente e la definizione delle risorse di cui può usufruire. Studiare il ruolo di ogni utente (o minaccia) all'interno di una infrastruttura informatica permette di capire la criticità di ogni risorsa e l'uso accettabile che se ne può fare, oppure di gestire le regole per il controllo degli accessi al sistema, o di stabilire tutti i comportamenti leciti o illeciti all'interno del sistema e di prevedere sanzioni in caso di violazioni alla politica di sicurezza.

Una **minaccia**, nel nostro contesto, è un utente con determinati diritti e risorse. I termini utenti e minacce sono utilizzati in questa tesi come sinonimi. Le minacce hanno degli obiettivi precisi da raggiungere, ossia particolari insiemi di diritti goal. Per raggiungere questi obiettivi le minacce eseguono degli attacchi contro il sistema.

> **Definizione 3.** Un attacco è l'azione di una minaccia per acquisire determinati privilegi.
>
> **Definizione 4.** Una vulnerabilità è un difetto del componente che abilita un attacco.
>
> **Definizione 5.** Una contromisura è un intervento sul sistema che consente di limitare l'insieme dei diritti di una minaccia per impedire l'esecuzione di un attacco contro il sistema.

La valutazione della sicurezza di un sistema inizia dall'analisi del rischio (Risk Analysis), utile per studiare minacce e vulnerabilità di una infrastruttura informatica e per capire come investire un insieme limitato di risorse a difesa delle aree più a rischio. Nella Risk Analysis si associano le vulnerabilità dei componenti del sistema informatico alle conseguenze che possono provocare nell'infrastruttura [16,17,18,23,42].
Il rischio è la perdita media dovuta all'attacco A contro il sistema; si ottiene moltiplicando il valore dell'impatto I, che rappresenta la conseguenza (perdita media) dell'attacco A contro il sistema, per la probabilità P che l'attacco A abbia successo:

> **Definizione 6.** Rischio = P(A_Succ) x I(A).

Nel caso di più attacchi il rischio complessivo del sistema è dato dalla media dei valori dei rischi relativi calcolati per ogni attacco contro il sistema.
Calcolare il valore del rischio, al fine di capire quali sono le vulnerabilità da eliminare [46], è un approccio sicuramente corretto, ma presenta diversi problemi.
Ad esempio l'associazione vulnerabilità-impatto risulta troppo spesso informale, soprattutto quando si cerca di stimare la probabilità che un determinato attacco abbia successo nel sistema [23].
Un altro problema è la mancanza di dati reali su cui effettuare il calcolo della probabilità che l'attacco abbia successo poiché è necessario uno storico di eventi come percentuali, tipologie di attacchi subiti e tempi necessari perché una certa vulnerabilità possa essere sfruttata. Di fatto la probabilità è un fattore che l'analista troppo spesso inserisce secondo la sua esperienza personale.
Attualmente nel web si possono reperire numerosi strumenti per determinare le vulnerabilità dei componenti di un sistema. Il problema è che la maggior parte di essi riconosce solo le vulnerabilità più diffuse e soprattutto che questi strumenti difficilmente tengono conto delle implicazioni delle vulnerabilità nel complesso del sistema, non considerando il fatto che una macchina possa essere o meno marginale all'interno della architettura [16,18].

La nostra proposta mira a fornire una metodologia che leghi le vulnerabilità ai componenti del sistema informatico da analizzare e che permetta di lavorare con uno storico limitato, avendo a disposizione dati certi sulla infrastruttura [29,30].

Ciò che vogliamo ottenere è una rappresentazione delle conseguenze di tutti gli attacchi possibili nel sistema, visualizzare l'insieme dei privilegi che ogni utente è riuscito ad ottenere e, quindi, capire quali attacchi sono stati necessari per il conseguimento degli obiettivi delle minacce.
Non si tratta, quindi, di un modello per scoprire né le vulnerabilità, né tanto meno le contromisure, bensì di uno strumento che simula scenari di attacco contro una infrastruttura informatica a partire dagli input noti del sistema e dalle ipotesi dell'analista [29,30].

In particolare l'attacco deve sfruttare una vulnerabilità del componente; per essere eseguito la minaccia deve possedere particolari diritti e risorse. L'attacco fornisce determinati diritti, utili al raggiungimento diretto dell'obiettivo della minaccia o per poter eseguire dei nuovi attacchi.
Gli attacchi **sequenziali** sono quei particolari attacchi necessari per conseguire un obiettivo della minaccia eseguibili a partire da precedenti attacchi eseguiti dalla minaccia stessa. In questo esempio gli attacchi a2 e a3 sono due attacchi sequenziali:

- la minaccia u1 esegue l'attacco a1;
- acquisisce i diritti X e Y;
- grazie all'acquisizione del diritto Y può eseguire a2;
- esegue a2 ed ottiene il diritto Z;
- grazie a Z può ora eseguire l'attacco a3;
- esegue a3, acquisisce i diritti J,K;
- i diritti J e K sono i diritti goal dell'obiettivo della minaccia u1 denominato goal1.

Tipico scenario è quello del malintenzionato esterno all'architettura che sfrutta l'identità di un utente per attaccare un nodo del sistema, ma anche quello di un utente con account limitato che cerca di violare la politica di sicurezza interna per acquisire privilegi di cui non può disporre, oppure ancora il caso dell'utente inesperto che inconsapevolmente crea delle vie di accesso al sistema.

Il non differenziare utenti del sistema e minacce che effettivamente eseguono gli attacchi è sicuramente una semplificazione, ma permette di focalizzare l'attenzione sulle operazioni dei componenti del sistema che si possono invocare: un utilizzo dei diritti non consentito dalla politica di sicurezza del sistema può causare effetti imprevisti e provocare un attacco a quei componenti dell'infrastruttura che presentano delle vulnerabilità.

Si suppone, quindi, che le vulnerabilità del sistema siano note, tali da poter essere sfruttate dalle minacce per attaccare i componenti che presentano vulnerabilità. Interessa, infatti, quali sono le conseguenze di una vulnerabilità in termini di acquisizione di diritti, e non da quale bug del componente è provocata.

Abbiamo detto che ogni minaccia viene caratterizzata da un insieme di diritti iniziali.
Le dipendenze note del sistema consentono ad un utente di espandere il proprio insieme dei diritti iniziali.

Vediamo un esempio:

- un server web (oggetto) gestisce i dati di un form;
- l'utente può scrivere dati in questo form grazie al metodo "scrivi_dati1", quindi possiede dei privilegi sul metodo.
- altro oggetto: un database con un altro metodo "scrivi_dati2" per scrivere i dati nei record.
- l'utente non può scrivere direttamente sul database, quindi non ha esplicitamente il diritto su questo secondo metodo.

Di fatto, quando l'utente scrive i dati nel form comunicando con il web server, sta scrivendo anche nel database, quindi acquisisce, per proprietà transitiva, il diritto di poter scrivere nel database.

Proprietà transitiva tra le dipendenze del sistema:

- se il diritto A dipende dal diritto B e B dipende da C, allora A dipende da C.

Grazie a questa proprietà è possibile studiare come le conseguenze di un attacco si propagano all'interno del sistema. In altri termini la proprietà transitiva permette l'espansione dell'insieme dei privilegi della minaccia che esegue l'attacco.

> **Definizione 7.** L'insieme dei diritti legali è costituito dai diritti acquisiti grazie alla chiusura transitiva (TC: Transitive Closure) tra l'insieme dei diritti iniziali e le dipendenze note nel sistema.

L'insieme dei diritti di un utente si espande seguendo il seguente schema:

- diritti iniziali;
- diritti legali;
- attacco => diritti obiettivo quell'attacco;
- TC;
- nuovo attacco;
- TC;

fino all'esaurimento della catena di attacchi che la minaccia è in grado di eseguire, ottenendo così l'insieme dei diritti estesi.

Abbiamo appena introdotto due concetti fondamentali per condurre la simulazione di un piano di attacco: la propagazione delle conseguenze di un attacco di una minaccia sull'intero sistema e la possibilità di generare sequenze di attacchi da parte delle minacce. Questi aspetti verranno discussi nei paragrafi 1.3.1 e 1.3.2 per la rappresentazione dell'ipergrafo delle dipendenze e del grafo delle evoluzioni.

L'attacco A ha successo nel sistema se:

- A è eseguito dalla minaccia U sul componente C;
- C possiede determinate vulnerabilità;
- U possiede tutti i diritti necessari per eseguire A;
- U controlla tutte le risorse necessarie per eseguire A.

Fornire una rappresentazione formale del comportamento della minaccia che esegue il singolo attacco non fornisce alcuna informazione rispetto a quelle che già si possedevano. L'aspetto più difficile da capire da parte di un analista risulta l'implicazione sul sistema di ogni attacco delle minacce [32,33,37,39]. Nel nostro contesto questo aspetto si concretizza nell'acquisizione di nuovi diritti, eventualmente necessari alle minacce per eseguire nuovi attacchi e conseguire gli obiettivi prefissati.

Riepilogando, vediamo la minaccia come un qualsiasi utente che, consapevolmente o no, ha interesse ad attaccare il sistema e che possiede determinate risorse che gli consentono di effettuare attacchi.

La minaccia raggiunge il proprio obiettivo sfruttando le vulnerabilità dei componenti che è in grado di controllare ed eseguendo una particolare sequenza di attacchi detta evoluzione, come vedremo in 1.3.2.

1.3 Modellazione dell'infrastruttura

La modellazione di quanto descritto nei precedenti paragrafi consiste nella rappresentazione di due grafi, quello delle dipendenze (trattasi di ipergrafo) e quello delle evoluzioni.
Vedremo nei paragrafi 3.1.1 e 3.2.1 le scelte implementative per rappresentare i due grafi di cui sopra, prima però è necessario fornire alcune definizioni:

> **Definizione 8.** Dati un insieme V di nodi e un insieme E di archi un grafo G è un insieme G = (V, E).

Ogni arco e ∈ E connette due vertici u ∈ V e v ∈ V detti estremi dell'arco; per questo motivo spesso un arco e viene identificato con la coppia formata dai suoi estremi <u,v>.
L'insieme V è chiamato insieme dei vertici (o insieme dei nodi) di G ed i suoi elementi sono detti vertici (o nodi). L'insieme E è chiamato insieme degli spigoli (o insieme degli archi) di G ed i suoi elementi sono detti spigoli (o archi). Un grafo può quindi essere rappresentato come un insieme di vertici V ed un insieme di coppie E, dove gli elementi di ciascuna coppia appartengono a V.

> **Definizione 9.** In un grafo orientato, un cammino <v0, v1, ..., vk> forma un ciclo se v0 = vk ed il cammino contiene almeno uno spigolo. Il ciclo è semplice, se v1,v2,...,vk sono distinti.

> **Definizione 10.** Un grafo senza cicli è aciclico. Spesso si usano le prime lettere del nome inglese Direct Acyclic Graph (DAG) per indicare un grafo orientato aciclico

Come generalizzazione del concetto di grafo si può ottenere quello di ipergrafo.

> **Definizione 11.** Un ipergrafo I è un insieme I = (V,E), ove V è un insieme finito di nodi e E è un insieme di sottoinsiemi di V (detti iperarchi) di cardinalità ≥ 2. Se E contiene solo sottoinsiemi di V di cardinalità 2 allora I è un grafo.

1.3.1 L'ipergrafo delle dipendenze

> **Definizione 12.** L'ipergrafo delle dipendenze è l'insieme dei nodi e degli iperarchi che rappresentano rispettivamente i componenti del sistema e le dipendenze che esistono tra di essi.

Questo particolare grafo descrive le conoscenze esplicite che l'analista possiede sul sistema.
Il principio adottato nello studio è quello di evitare la "Security Through Obscurity". L'idea alla base di questo concetto è che mantenendo segrete le informazioni di un sistema, si offrono pochi vantaggi agli eventuali attaccanti. Recentemente, il principio opposto, cioè la "Full Disclosure", ha preso il sopravvento, diventando uno dei requisiti fondamentali per un sistema sicuro.

Nel nostro contesto assumiamo che una minaccia abbia la stessa visibilità delle informazioni rispetto a chi difende il sistema; in altri termini ha a disposizione tutte le informazioni rappresentate dall'ipergrafo delle dipendenze, ipergrafo in quanto una dipendenza può avere più diritti sorgenti, come si può osservare dalla Figura1.
Importante notare che, a priori, non si dovrebbero porre limiti al numero dei diritti dal quale un diritto dipende; tuttavia, nello sviluppo dello strumento, si è deciso di concentrarsi più sugli attacchi delle minacce contro il sistema che sulle particolari configurazioni delle dipendenze (vedi paragrafo 3.1.1), considerando quindi dipendenze relativamente semplici composte da al più quattro diritti sorgenti.

Ogni minaccia del sistema espande il proprio insieme di privilegi sfruttando la chiusura transitiva tra l'insieme dei diritti iniziali e le dipendenze note del sistema, fino ad ottenere l'insieme dei diritti legali, definiti in 1.2.

Lo schema seguente chiarisce questo concetto:

- diritti iniziali della minaccia M : {a,b,c,d};

- dipendenze note del sistema:
  diritto a implica diritto d;
  diritti (c AND d) implicano diritto e;

- la minaccia M a partire da {a,b,c,d} espande il proprio insieme a {a,b,c,d,e} (la prima dipendenza non aggiunge informazione in quanto il diritto d era già in possesso della minaccia M).

Come da definizione in 1.1, possedere un diritto significa poter invocare una operazione del componente generando il controllo di uno o più attributi di sicurezza. L'ipergrafo delle dipendenze evidenzia tutte quelle relazioni che permettono di acquisire diritti su un componente a partire da quelli posseduti sui componenti sorgente della dipendenza.

La Figura1 mostra una possibile configurazione del sistema attraverso l'ipergrafo delle dipendenze appena descritto:

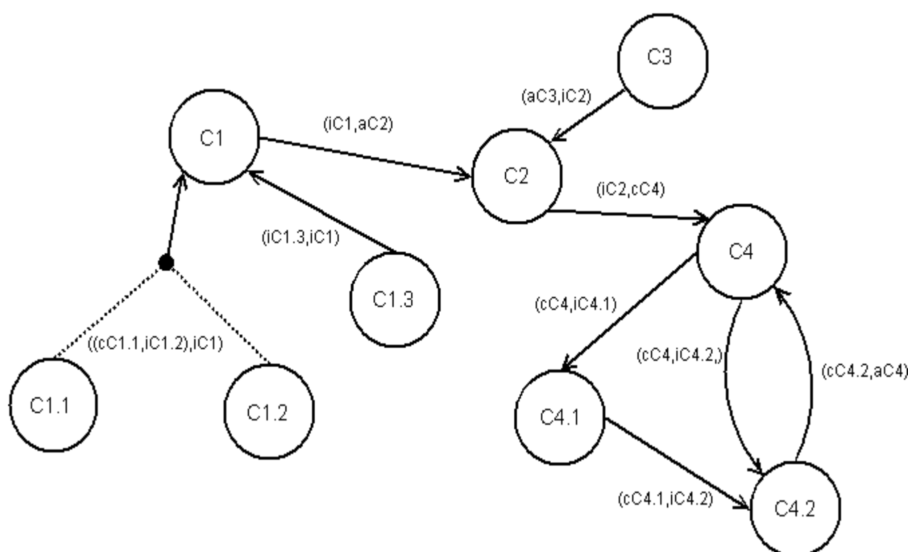

Figura1: Ipergrafo delle dipendenze

Le etichette sugli iperarchi indicano la dipendenza tra due operazioni sugli attributi di sicurezza dei componenti, o, più semplicemente, la dipendenza tra due diritti.
Per esempio (iC2,cC4) significa che possedere il diritto di scrittura (integrità) su C2 implica il poter leggere informazioni (confidenzialità) sul componente C4.
Inoltre possiamo notare il caso di una dipendenza multipla tra diritti (iperarco): ((cC1.1,iC1.2),iC1). Questa dipendenza permette di rappresentare l'informazione che se una minaccia è in grado di leggere informazioni su C1.1 e di scrivere su C1.2, allora raggiunge anche il diritto di scrittura sul componente C1.

Grazie alla rappresentazione dell'ipergrafo delle dipendenze è possibile inoltre capire l'importanza strategica di alcuni diritti per le minacce. Ad esempio:

- la minaccia U per raggiungere il privilegio X deve possedere i diritti Y,Z;

Ipotizziamo che il diritto X sia la possibilità di effettuare acquisti su un sito di commercio elettronico (integrità sul database a cui il sito si collega); questo privilegio viene raggiunto se si è in grado di leggere informazioni sul server che gestisce gli username degli utenti (diritto Y) e sul file criptato che memorizza le password relative (diritto Z).
Ipotizzando che una minaccia, per raggiungere tutti i diritti sorgenti della dipendenza che permette di acquisire il privilegio X (Y e Z in questo esempio), esegue numerosi attacchi, questa dipendenza evidenzia quanto il diritto X sia importante per la minaccia e quindi pericoloso per la sicurezza del sistema.

Infine si osserva che l'ipergrafo delle dipendenze non rispetta la definizione di DAG, come invece vedremo per quanto riguarda il grafo delle evoluzioni.

Per esempio nella Figura1 sono ammessi iperarchi che formano cicli tra i componenti C4 e C4.2.
In particolare le due dipendenze che causano il ciclo sono (cC4,iC4.2) e (cC4.2,aC4) ed evidenziano l'informazione che dalla confidenzialità su C4 si arriva all'integrità di C4.2 e che invece dalla confidenzialità su C4.2 si ottiene il controllo del componente C4 (availability). In questo esempio il ciclo non riguarda gli stessi attributi di sicurezza su entrambe i componenti, ma può verificarsi la situazione di due dipendenze che introducono ridondanza tra diritti, per esempio (cC1,cC2) e (cC2,cC1). La prima dipendenza indica che se una minaccia possiede cC1 allora acquisisce cC2, la seconda dipendenza esprime l'opposta informazione.
La dipendenza (X,Y) è una implicazione da leggersi solo da sinistra verso destra, in quanto una minaccia potrebbe avere tra i suoi diritti il privilegio Y senza però possedere X (è il caso in cui Y fa parte dell'insieme dei diritti iniziali oppure è acquisito grazie ad un attacco).
Il ciclo causato dalle dipendenze (X,Y) e (Y,X) esprime l'ovvia informazione che sia X sia Y saranno entrambi acquisiti dalla minaccia che può sfruttare l'una o l'altra dipendenza.

1.3.2 Il grafo delle evoluzioni

Prima di discutere il grafo delle evoluzioni è necessario definire i termini stato ed evoluzione.

Definizione 13. Uno stato del sistema è un insieme di coppie <minaccia,diritto>.

Lo stato iniziale contiene tutte le minacce con i relativi diritti legali (da Definizione 6 quei diritti acquisiti dopo il calcolo della chiusura transitiva tra l'insieme dei diritti iniziali e le dipendenze note

del sistema). La transizione di stato è provocata dall'esecuzione dell'attacco da parte della minaccia. Lo stato successivo conterrà tutte le coppie <minaccia,diritto> dello stato precedente più le nuove coppie composte dalla minaccia che ha eseguito l'attacco e i nuovi diritti, frutto dell'attacco o delle dipendenze del sistema.

> **Definizione 14.** Lo stato goal (goalState) è il primo stato che contiene tutti i diritti di un obiettivo della minaccia (diritti goal).

> **Definizione 15.** Due stati sono equivalenti se contengono identiche coppie <utente,diritto>.

> **Definizione 16.** Il grafo delle evoluzioni è l'insieme dei nodi e degli archi che rappresentano rispettivamente gli stati del sistema e le transizioni possibili (attacchi delle minacce).

> **Definizione 17.** Una evoluzione del sistema (o evoluzione utente) è un cammino del grafo delle evoluzioni dallo stato iniziale, in cui una minaccia possiede i soli diritti legali, allo stato goal.

Il grafo delle evoluzioni mostra i cammini (sequenze di attacchi) delle minacce per raggiungere i relativi obiettivi. I nodi rappresentano gli stati del sistema, gli archi le transizioni di stato, ossia gli attacchi che le minacce compiono per raggiungere un nuovo stato [29].
Ogni cammino del grafo che termina in un goalState rappresenta una evoluzione utente.
Importante notare che non vengono considerate valide quelle transizioni di stato che non aggiungono informazione rispetto allo stato precedente: un attacco che non fornisce alcun nuovo diritto non serve a nessuna minaccia, e quindi non deve essere preso in considerazione.

Abbiamo introdotto il concetto di **monotonicità**: la funzione che calcola le evoluzioni del sistema è una funzione monotona.
Nel nostro contesto $S1 < S2$ significa che lo stato S1 è stato generato prima dello stato S2 e quindi, per definizione di funzione monotona, $f(S1) < f(S2)$ significa che l'insieme dei diritti per la minaccia che causa la transizione in S2 deve essere maggiore all'insieme contenuto nel precedente stato S1.

Grazie alla proprietà appena discussa ogni stato di un cammino del grafo delle evoluzioni non può essere uguale allo stato che lo precede. Questo è fondamentale per evitare che venga eseguita sempre la stessa azione. Se, infatti, le pre-condizioni dell'attacco A sono soddisfatte all'istante T, lo saranno anche dopo la sua applicazione all'istante T+1 (ipotesi valida proprio perché la funzione che calcola le evoluzioni è monotona). In questo modo si esclude che l'attacco A sia nuovamente eseguito dato che tutte le sue post-condizioni (diritti acquisiti in conseguenza dell'attacco A) sono già state acquisite dall'utente.

Il concetto di evoluzione evidenzia come, per ogni minaccia del sistema, non sia più sufficiente il singolo attacco e la relativa chiusura transitiva per raggiungere un suo obiettivo. La chiusura transitiva aggiunge una serie di nuovi diritti necessari per eseguire un nuovo attacco, e così via, fin tanto che la minaccia non abbia eseguito ogni possibile attacco.
A questo punto o la minaccia ha raggiunto il suo obiettivo (goalState), quindi abbiamo una situazione che ci interessa modellata dalla particolare evoluzione nel grafo, oppure quella minaccia non ha una rilevanza critica per la sicurezza del sistema.

Sotto l'ipotesi che una minaccia del sistema possa solo acquisire privilegi, il grafo delle evoluzioni è un grafo orientato aciclico (DAG).

In un contesto in cui si possono adottare delle contromisure durante l'esecuzione degli attacchi da parte delle minacce contro il sistema, può verificarsi il caso in cui una minaccia invece di espandere il proprio insieme dei diritti subisca un decremento numerico. Ciò corrisponderebbe ad un arco all'indietro nel grafo delle evoluzioni, che significa l'inevitabile generazione di un ciclo. Questa situazione per il momento non è stata considerata, ma potrebbe essere approfondita negli eventuali sviluppi futuri dello strumento.

La Figura2 mostra un esempio del grafo delle evoluzioni appena descritto:

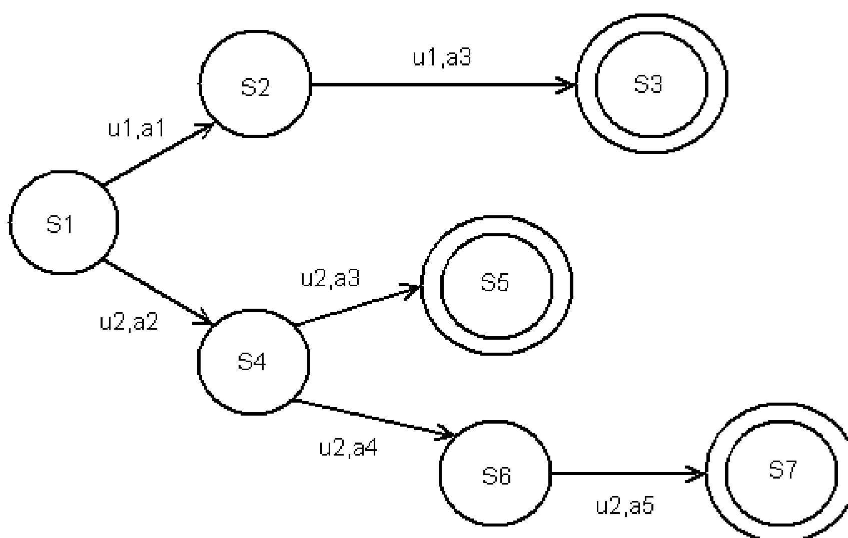

Figura2: Grafo delle evoluzioni

L'etichetta sugli archi evidenzia la minaccia e il relativo attacco sfruttato. I nodi doppiamente cerchiati rappresentano gli stati goal delle minacce (goalState).
Nel paragrafo 3.2.1 vengono presentate le scelte implementative che hanno permesso di rappresentare le informazioni contenute nel grafo delle evoluzioni.

CAPITOLO 2 : TECNOLOGIA IMPIEGATA

In questo capitolo vengono presentate le principali differenze tra programmazione imperativa e programmazione logica; si evidenziano le motivazioni che hanno indotto all'utilizzo della programmazione logica per lo sviluppo dello strumento; vengono inoltre analizzate le caratteristiche principali del linguaggio adottato: il Prolog.

2.1 La scelta della programmazione logica

Programmazione procedurale (C, Pascal, Basic, etc.) e programmazione ad oggetti (Java, C++, etc.) costituiscono due modi diversi per risolvere problemi, ma rispecchiano entrambe un unico paradigma di sviluppo detto imperativo. Capire il paradigma imperativo significa comprendere quali sono le principali caratteristiche dei linguaggi di programmazione oggi più conosciuti.
La caratteristica principale di tutti i linguaggi imperativi è quella di richiedere che un programma sia costituito da un insieme di istruzioni, poste in sequenza o organizzate tramite determinate strutture di controllo. Queste istruzioni manipolano dati, di solito referenziati tramite variabili. Tutti questi linguaggi consentono di dichiarare variabili e funzioni, forniscono tipi di dato elementari e costrutti come array e strutture per creare tipi più complessi. Tutti, inoltre, forniscono strutture di controllo come il FOR per effettuare iterazioni, o l'IF per eseguire condizionatamente alcune istruzioni.
Le differenze, quindi, sono quasi esclusivamente di tipo sintattico, legate alle librerie o al diverso uso di strutture ed array. La logica con cui il problema è stato risolto (la struttura dell'algoritmo) rimane la stessa in tutti i linguaggi imperativi.

La programmazione logica utilizza un paradigma di sviluppo alternativo a quello adottato dalla maggior parte degli attuali linguaggi di programmazione. Invece di fornire una serie di istruzioni per risolvere un particolare problema, ci si limita a specificare gli input necessari attraverso la definizione dei Fatti noti e quale output, il Goal, debba essere ottenuto [5,8,11,12]. È l'interprete del linguaggio che si assume il compito di trovare il modo migliore per risolvere il problema (algoritmo). Non esistono FOR, non esistono IF, non è possibile "tradurre" un programma scritto con un linguaggio imperativo in uno logico, se non ripensando completamente il programma stesso.

La programmazione logica ha solide basi matematiche, fondate sulla logica simbolica.
Di fatto, ciò che si può scrivere con un programma logico corrisponde ad un sottoinsieme delle cosiddette clausole di Horn, che sono disgiunzioni di letterali in cui al massimo uno è positivo.
Queste, a loro volta, costituiscono un sottoinsieme delle forme clausali, che sono una forma semplificata del calcolo dei predicati, branca della logica matematica.

L'esempio seguente ha lo scopo di introdurre i concetti che stanno alla base della programmazione logica.

Supponiamo di avere un database con l'elenco delle persone che hanno lavorato presso un'azienda.
Per ogni persona è indicata la data di assunzione e la data di termine rapporto.
Il programma, date due persone, deve determinare se hanno lavorato contemporaneamente in azienda o in periodi separati.
Raffinando queste specifiche si ottiene:

X ha lavorato insieme a Y se :

- X ha lavorato in azienda dalla data iniziale I1 alla data finale F1;
- Y ha lavorato in azienda dalla data I2 alla data F2;
- il periodo I1-F1 si sovrappone al periodo I2-F2;

Il periodo I1-F1 si sovrappone al periodo I2-F2 se:

- I2 è maggiore o uguale a I1 ma minore o uguale a F1 (in altri termini se Y ha iniziato a lavorare dopo X, ma prima che X terminasse il rapporto con l'azienda);
- I1 è maggiore o uguale a I2 ma minore o uguale a F2 (nel caso contrario).

Immaginando di dover scrivere un programma che implementi le specifiche di cui sopra con un linguaggio imperativo, il primo passo da effettuare è quello di tradurre la descrizione in un algoritmo corrispondente.
In questo caso la traduzione risulta immediata: è sufficiente prelevare i valori delle date relative alle due persone, confrontarle e ritornare il valore ottenuto. Successivamente si potranno implementare delle ottimizzazioni, come ad esempio quella di evitare il secondo confronto nel caso in cui il primo abbia già dato esito positivo.

Per quanto riguarda la programmazione logica, in questo caso non deve essere svolto nessun tipo di traduzione. La specifica è già il programma, l'unico compito da assolvere è quello di trascrivere la specifica stessa usando una sintassi corretta. È sufficiente indicare il significato di "lavorare insieme" e di "sovrapporsi"

- insieme(X,Y) :-
  ha_lavorato(X,I1,D1), ha_lavorato(Y,I2,D2), si_sovrappone(I1,D1,I2,D2).

- si_sovrappone(I1,D1,I2,D2) :-
  I2 >= I1, I2 <= D1.

- si_sovrappone(I1,D1,I2,D2) :-
  I1 >= I2, I1 <= D2.

e lasciare all'interprete il compito di risolvere il problema. A questo punto potremo interrogare il sistema scrivendo

- ? insieme(rossi,verdi).

In caso positivo il sistema risponderà con: True.

Tradurre queste specifiche in codice, utilizzando un linguaggio tradizionale, non è stato uno sforzo concettuale così complesso. Immaginando, però, di voler estendere il programma per conoscere, data una persona, l'elenco di tutte le persone che hanno lavorato con la stessa, la traduzione non è immediata. Dovremo aggiungere una iterazione per confrontare le date relative alla prima persona con quelle di ognuna delle altre persone presenti nel database.
Questo tipo di estensione utilizzando un linguaggio logico non è necessaria in quanto è sufficiente interrogare l'interprete scrivendo:

- ? insieme(rossi,X).

Per ottenere la risposta: verdi; bianchi; ferrari.

La programmazione logica ha avuto negli Anni '80 una discreta diffusione, soprattutto legata al campo dell'intelligenza artificiale [9]. Una delle scommesse di quegli anni era il progetto giapponese denominato Fifth Generation, legato allo sviluppo di una nuova generazione di computer "intelligenti", basati sui linguaggi di programmazione logica.
Recentemente l'interesse verso il campo dell'intelligenza artificiale è diminuito e questo probabilmente perché le aspettative legate all'intelligenza artificiale e alla programmazione logica erano troppo elevate. I vantaggi di questo tipo di approccio sono però ancora oggi evidenti in determinati contesti applicativi:

- sistemi esperti;
- riconoscimento del linguaggio naturale;
- intelligenza artificiale.

In [10] è possibile consultare un elenco recente di applicazioni sviluppate mediante un programma logico: applicazioni per la gestione di aeroporti, sistemi informativi logistici, sistemi per la gestione di ospedali e molte altre cosiddette "storie di successo", che testimoniano la praticabilità di questo tipo di approccio alla programmazione.

Nel nostro contesto la programmazione logica è stata da subito preferita a quella imperativa per lo sviluppo dello strumento software. Vediamone le motivazioni.

Per rappresentare l'ipergrafo delle dipendenze e per ottenere l'insieme dei diritti finali di ogni utente a partire dai privilegi iniziali, si è scelto di adottare uno dei linguaggi possibili nel mondo della programmazione logica, in particolare la versione free di SWI-Prolog [1,2,3,7].

L'utilizzo di un linguaggio dichiarativo permette di sfruttare il meccanismo di backtracking come caratteristica nativa del linguaggio al fine di poter calcolare tutte le evoluzioni di ogni utente dell'infrastruttura.

Nel paragrafo successivo vengono definiti tutti i termini chiave del linguaggio adottato per lo sviluppo dello strumento; qui di seguito, invece, si riportano le applicazioni nel nostro contesto delle tre caratteristiche principali della programmazione logica: Fatti, Regole e Goal.

- I Fatti sono stati utilizzati per inserire tutte le informazioni note del sistema e per ottenere la rappresentazione dell'ipergrafo delle dipendenze e del grafo delle evoluzioni, come discusso in 3.1.1 e 3.2.1.

- Le Regole consentono di acquisire attraverso deduzioni logiche nuove informazioni dopo l'esecuzione degli attacchi da parte delle minacce.
  Situazione tipica dell'applicazione di una regola è quella dell'aggiunta dei diritti all'insieme di una minaccia grazie alle dipendenze tra gli attributi di sicurezza dei componenti o attraverso un attacco di una minaccia contro il sistema.

- Lo scopo di un programma logico è quello di interrogare il sistema per verificare se una determinata risposta (il Goal) è derivabile dai Fatti e dalle Regole del programma.
  Per esempio, nel nostro contesto, soddisfare un Goal significa verificare la terminazione con successo di una minaccia che mira ad acquisire un insieme di diritti goal (obiettivo).

Per quanto concerne le strutture dati, la scelta principale è stata quella dell'uso di un tipo di dato astratto, l'insieme [6], realizzato senza l'implementazione di liste, ma sfruttando sempre e solo la struttura dei termini definita attraverso Fatti e Regole.

La Regola usata per gestire il tipo di dati astratto insieme è:

- `elem(X,Y):- (condizioni).`

che significa "se valgono le condizioni specificate allora l'elemento X appartiene all'insieme Y".
Questo termine è stato pesantemente sfruttato con una duplice funzione: quella di esprimere un input del programma (Fatto) e quella di verificare l'appartenenza di determinati elementi ad un insieme (Regola). La caratteristica principale è quella di poter interrogare l'interprete in due modalità, una per testare se un particolare elemento appartiene all'insieme o meno, ottenendo dall'interprete una risposta del tipo YES/NO, e l'altra che utilizzi il meccanismo di backtracking per ottenere un elenco di tutti gli elementi che appartengono a quel particolare insieme.
Inoltre è possibile richiamare ed utilizzare le clausole definite nei diversi file: se il file A include il file B e B include C, allora A può usufruire delle clausole sia di B che di C.

I vantaggi principali della scelta della programmazione logica sono in termini di tempo di esecuzione e di memoria allocata per i moduli del progetto, ma riguardano anche la possibilità di aggiungere informazioni ad un insieme sfruttando il meccanismo di backtracking, la completa indipendenza da variabili statiche e/o strutture dati globali e la totale portabilità dei dati verso altri ambienti che utilizzino il paradigma della programmazione logica come linguaggio base.

Riassumendo, l'uniformità nella rappresentazione del sistema permette di effettuare deduzioni logiche, indipendentemente dal livello di astrazione, per acquisire conoscenza sulle minacce in termini di diritti e consente di capire quali di esse riescono ad ottenere il controllo del sistema.
La programmazione logica è d'ausilio in questa fase per rappresentare l'ipergrafo delle dipendenze, descritto in 1.3.1 e in 3.1.1. Lo sforzo iniziale di dover trascrivere tutte le informazioni che fungono da input per l'infrastruttura garantisce, tuttavia, di considerare tutti gli obiettivi che una minaccia è in grado di raggiungere, oppure, tutte le minacce che possono ottenere determinati obiettivi.
La programmazione logica, anche in questo caso, viene utilizzata per la rappresentazione del grafo delle evoluzioni, introdotto in 1.3.2 e descritto nel dettaglio in 3.1.2. Inoltre è il programma logico stesso che gestisce l'esplosione di stati, normalmente controllata dall'analista, dovuta al calcolo del prodotto cartesiano tra gli stati del grafo delle evoluzioni, fornendo un elenco di informazioni difficile da ottenere manualmente.

2.2 Prolog: caratteristiche principali

Il Prolog (PROgramming in LOGic) è un linguaggio di programmazione che adotta il paradigma logico.
Ideato da Robert Kowalski (aspetto teorico), Marten Van Emdem (dimostrazione sperimentale) ed implementato da Alain Colmerauer negli Anni '70, rappresenta il tentativo di costruire un linguaggio di programmazione che consenta l'espressione del problema in forma logica, invece della classica traduzione di un algoritmo di soluzione in forma di istruzioni da eseguire da parte della macchina. Viene impiegato in molti programmi di intelligenza artificiale, anche perché la sua sintassi e la semantica sono molto semplici e chiare.

Come la logica proposizionale, il Prolog si basa sul calcolo dei predicati del primo ordine.
Nella logica matematica il linguaggio del primo ordine serve per gestire meccanicamente enunciati e ragionamenti che coinvolgono i connettivi logici, le relazioni e i quantificatori universale $\forall$ ed esistenziale $\exists$.
L'espressione “del primo ordine” indica che c'è un insieme di riferimento e i quantificatori possono riguardare solo gli elementi di tale insieme e non i sottoinsiemi; ad esempio si può dire “per tutti gli elementi x dell'insieme vale P(x)” ma non “per tutti i sottoinsiemi A vale P(A)”;

Tuttavia la sintassi è limitata a formule dette clausole di Horn che sono disgiunzioni di letterali del primo ordine quantificate universalmente con al più un letterale positivo.
L'esecuzione di un programma Prolog equivale alla dimostrazione di un teorema mediante la regola di inferenza detta risoluzione (introdotta da Robinson nel 1965). Ricordiamo che per regola di inferenza nella logica matematica si intende una regola formale che stabilisce quando un enunciato formalizzato (una formula di un linguaggio proposizionale o del primo ordine) è conseguenza logica di un altro, soltanto sulla base della struttura sintattica degli enunciati.

Vengono riportate in modo sintetico le definizioni delle principali caratteristiche Prolog, ossia Regole, Fatti e Goal, nonché i concetti chiave del linguaggio: unificazione, ricorsione (in coda) e backtracking.

- *Clausola di Horn*

In logica, e in particolare nel calcolo proposizionale, una clausola di Horn è una disgiunzione di letterali in cui al massimo uno dei letterali è positivo.

Un esempio di clausola di Horn è il seguente: $\neg A \vee \neg B \vee C$.

Il numero dei letterali può essere arbitrario (anche zero); la condizione che al massimo uno sia positivo permette di scrivere la clausola sotto forma di implicazione.
Se il numero di letterali positivi è esattamente uno, la clausole di Horn vengono dette definite.
La premessa (corpo) di tali clausole è una congiunzione di letterali positivi e la sua conclusione (testa) è un singolo letterale positivo.

Partendo dall'esempio, applichiamo prima De Morgan: $\neg (A \wedge B) \vee C$;
dopodiché utilizziamo l'equivalenza logica: $\neg X \vee Y = X \Rightarrow Y$;
ricaviamo quindi: $(A \wedge B) \Rightarrow C$.
Nella progettazione dei database, formule di questo tipo sono chiamate vincoli di integrità.

Un programma Prolog è un insieme di clausole di Horn che rappresentano Fatti, Regole e Goal.

- *Regola Prolog*

- A :- B,C. $\equiv$ $(A \Leftarrow B \wedge C)$ $\equiv$ $(A \vee \neg B \vee \neg C)$.

Come si può notare dall'ultimo passaggio la regola Prolog non è altro che una clausola di Horn con un solo predicato affermato (si ricorda che le clausole di Horn sono letterali del primo ordine in OR tra loro con al massimo un letterale positivo).

- Fatto Prolog

- A. $\equiv$ (A <= tt) $\equiv$ (ff $\vee$ A) $\equiv$ A.

Il fatto Prolog anche in questo caso è una clausola di Horn con un solo predicato affermato.

- Goal Prolog

- :- B,C. $\equiv$ (ff <= B $\wedge$ C) $\equiv$ ($\neg$B $\vee$ $\neg$C $\vee$ $\neg$ff) $\equiv$ ($\neg$B $\vee$ $\neg$C).

Il goal Prolog è una clausola di Horn con tutti i predicati negati.
Definizione alternativa del Goal Prolog nel caso di predicato con variabile:

- :- A(x). $\equiv$ $\forall$ x ($\neg$A(x)).

- Unificazione

Dati due termini si dice che unificano se:

- sono identici;
- le variabili contenuti in entrambi i termini possono essere sostituite con costanti/variabili tali da rendere identici, dopo la sostituzione, i due termini.

A differenza del matching (in cui il termine è messo in corrispondenza con un pattern), l'unificazione è un processo bidirezionale: entrambi i termini possono contenere costanti e variabili.

Esempi:

- ? X = italia ha successo con X = italia;
- ? francia = italia fallisce;
- ? f(X,a) = f(b,Y) ha successo con X = b e Y = a;
- ? f(X,Y) = f(Y,c) ha successo con X = Y = c;
- ? f(X,a) = f(Y,b) fallisce;

- Ricorsione (in coda)

Gli algoritmi ricorsivi sono utilizzati in Prolog per eseguire compiti ripetitivi su di un insieme di input: l'algoritmo richiama se stesso generando un ciclo di chiamate che ha termine al verificarsi di una condizione particolare detta caso di base.
Si parla di ricorsione di coda (Tail Recursion) quando la chiamata ricorsiva è l'ultima istruzione eseguita nella funzione.
Inoltre, nel caso della programmazione logica, la ricorsione è efficiente quanto un ciclo iterativo, compensando così la assenza in Prolog e simili del concetto di iterazione; non esistono, infatti, costrutti come cicli FOR o WHILE pesantemente utilizzati nella programmazione imperativa.

- *Backtracking*

Questa tecnica è la caratteristica principale dei linguaggi di programmazione logica.
Il backtracking è un modo sistematico per individuare tutte le possibili soluzioni di un determinato problema. Per esempio tutte le possibili permutazioni di un insieme di oggetti, o tutti i possibili sottoinsiemi, oppure ancora tutti i cammini tra due nodi di un grafo, o tutti gli alberi di copertura di un grafo. La caratteristica comune di tutti questi problemi è quella di dover generare univocamente ogni possibile soluzione. Per evitare perdite o duplicazioni di dati è necessario definire un ordine sistematico di generazione tra le possibili soluzioni.
Il backtracking costruisce un albero (albero delle soluzioni) dove ciascun vertice interno è una soluzione parziale e c'è un arco che va dal vertice x al vertice y se il vertice y è stato creato estendendo la soluzione parziale del vertice x. Le foglie dell'albero sono le soluzioni. L'albero delle soluzioni offre un modo alternativo di pensare al backtracking in quanto il processo di costruzione delle soluzioni corrisponde alla visita in profondità dell'albero, come mostrato in Figura3.

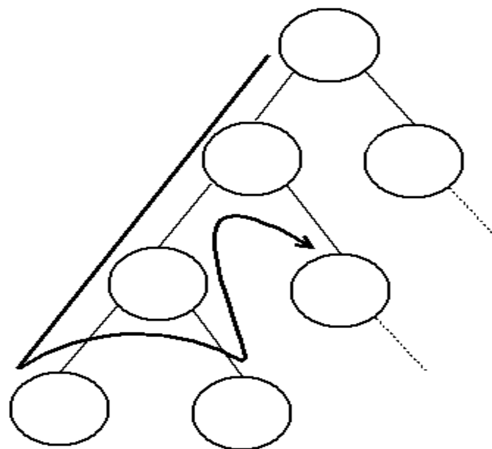

Figura3: Ricerca con backtracking

CAPITOLO 3 : SVILUPPO DEL SOFTWARE

3.1 Definizione dello strumento

Lo strumento sviluppato ha lo scopo di simulare scenari di attacco contro una infrastruttura informatica. Risultato della modellazione è la rappresentazione mediante clausole Prolog dei grafi delle dipendenze (ipergrafo) e delle evoluzioni, già discussi in 1.3.1 e 1.3.2.

Usando la programmazione logica, infatti, abbiamo sfruttato la libertà di poter definire termini e clausole che rappresentino semanticamente ciò che con la programmazione imperativa avremmo dovuto descrivere attraverso strutture dati di non facile gestione, come, per esempio, quella del grafo.

Una delle caratteristiche principali dello strumento è la possibilità di poter simulare molteplici infrastrutture, modificando il solo modulo contenente tutte le informazioni dell'infrastruttura che fungono da input per la simulazione. Rimangono quindi trasparenti all'utilizzatore dettagli tecnici e algoritmi alla base della modellazione.

Tutto ciò che l'amministratore di rete o l'analista dovrà compiere sarà una trascrizione formale delle conoscenze acquisite sulla infrastruttura studiata; in altri termini capire e descrivere le minacce del sistema, i loro privilegi iniziali, le risorse a disposizione, gli obiettivi che vorrebbero raggiungere ed i relativi diritti goal, i componenti del sistema, quelli vulnerabili, le dipendenze tra diritti, ma, soprattutto, tutti gli attacchi che si conoscono o si ipotizzano eseguibili da parte delle minacce.

Come da definizione in 1.2, gli attacchi sono le azioni che le minacce compiono per raggiungere determinati diritti, sfruttano vulnerabilità note dei componenti e consentono di generare una catena di attacchi fino all'eventuale conseguimento dell'obiettivo (insieme di diritti goal) della minaccia.

Una volta inseriti correttamente tutti gli input necessari si può interrogare l'interprete Prolog per risalire alle informazioni cercate.
In particolare si possono eseguire query a specifici database al fine di ottenere, per ogni minaccia, l'insieme dei diritti che è riuscita a raggiungere. Inoltre riusciremo ad ottenere sia la sequenza di attacchi che ogni utente dell'infrastruttura esegue per arrivare al proprio stato goal - le evoluzioni utente - sia ogni possibile configurazione del sistema in conseguenza di tutti gli attacchi eseguiti dalle minacce.

Uno stato dell'infrastruttura non è altro che un insieme di coppie <utente,diritto>, che rappresenta una configurazione del sistema in cui sono elencate tutte le minacce con i diritti acquisiti fino a quel momento. Le evoluzioni utente partono dallo stato iniziale, in cui le minacce possiedono i soli diritti legali (quelli derivanti dall'insieme dei diritti iniziali e dal calcolo delle dipendenze note nel sistema), fino allo stato finale, detto goalState, in cui la minaccia ha raggiunto uno dei suoi obiettivi.

Prendiamo ora in analisi la struttura e lo scopo dei moduli principali del progetto, discutendone gli algoritmi fondamentali. Presenteremo il modulo di inserimento dei dati (input del sistema), necessario per descrivere tutte le conoscenze acquisite/ipotizzate sull'infrastruttura e contenente tutti i Fatti Prolog necessari alla rappresentazione dell'ipergrafo delle dipendenze, il modulo per i risultati ottenuti dalle simulazioni (output) e infine il modulo che si occupa delle evoluzioni utente e della rappresentazione del grafo delle evoluzioni.

3.1.1 Input necessari e rappresentazione dell'ipergrafo delle dipendenze

Questo modulo contiene tutte le informazioni sul sistema prima degli attacchi delle minacce.
Esso raccoglie tutte e sole le caratteristiche del sistema, quelle dedotte dallo studio reale della infrastruttura, più quelle ipotizzate dall'analista, utili per condurre la simulazione degli scenari possibili. Funge da base di partenza per le successive modellazioni dell'ipergrafo delle dipendenze e del grafo delle evoluzioni ed è l'unico modulo dell'intero progetto in cui si inseriscono manualmente dei dati. Cambiando solo alcune clausole, infatti, sarà possibile studiare una nuova infrastruttura.

Il software sviluppato è stato strutturato in moduli (i vari moduli possono interagire grazie al comando Prolog include(nomeFile)). Uno di questi moduli definisce le nozioni insiemistiche, come ad esempio:

- `elem(X,unione(A,B)).` (il termine elem è già stato discusso in 2.1).
- `sottinsieme(A,B).`

Queste due clausole mostrano i due principali modi per interrogare l'interprete Prolog: è possibile, infatti, individuare determinati elementi all'interno di uno o più insiemi, come nel caso della prima clausola, oppure verificare particolari proprietà insiemistiche, ottenendo dall'interprete risposte del tipo YES/NO.

Per ogni infrastruttura studiata si descrivono gli utenti iniziali U (minacce), con i loro diritti iniziali contenuti nell'insieme dirIn(U), i relativi obiettivi G elencati nell'insieme goal(U,G) e le risorse usufruibili memorizzate in ris(U).

Si ipotizzano n componenti del sistema, collegati tra loro tramite dipendenze.
Il concetto di componente è stato espresso mediante la clausola

- `components(C).` dove per C si intende il componente;

mentre le dipendenze sono state espresse mediante

- `implies(xCn,xCn).` dove per x si intende uno dei tre possibili attributi di sicurezza sul componente (c/i/a) e per Cn il componente soggetto a dipendenza.

L'ordine della coppia di termini che viene passata alla clausola implies(X,Y) definisce la dipendenza tra due privilegi; per esempio, l'iperarco <iC2,cC4> mostrato in Figura1 è espresso attraverso la clausola

- `implies(iC2,cC4).`

ed esprime il concetto, già introdotto in 1.3.1, che possedere il permesso di scrittura (integrity i) sul componente C2 implica il raggiungimento del diritto di lettura (confidentiality c) sul componente C4 (in altri termini iC2 implica cC4, quindi cC4 dipende da iC2).
Come si può notare, la relazione di dipendenza non riguarda solo due attributi di sicurezza dello stesso componente o tra gli stessi attributi di due componenti (cC1 implica iC1, oppure cC1 implica cC2), ma la relazione può collegare tra loro attributi e componenti in diverse combinazioni.

Le dipendenze tra diritti sono sempre semplici (1:1) o multiple (N:1).
Per esempio, nel caso di relazioni 1:1 (diritto implica diritto), possiamo avere le seguenti combinazioni:

- 1:1 tra attributi diversi dello stesso componente (cC1 implica iC1);
- 1:1 tra attributi uguali di componenti diversi (cC1 implica cC2);
- 1:1 tra attributi diversi di componenti diversi (cC1 implica iC2);

Come detto in 1.3.1, a priori, non si dovrebbero porre limiti al numero dei diritti dal quale un diritto dipende; un limite dello strumento presentato è sicuramente quello di considerare dipendenze relativamente semplici composte da al più quattro diritti sorgenti.

È importante notare quanto sia potente il concetto di dipendenza fra diritti: le dipendenze possono consentire ad una minaccia, che dispone inizialmente di un limitato insieme di privilegi, di acquisire a catena molteplici diritti, come vedremo nella simulazione descritta in 4.2, o di poter eseguire una sequenza di attacchi necessaria per raggiungere un suo obiettivo.

Per quanto riguarda gli attacchi conosciamo le seguenti informazioni:

- risorse necessarie => ris(A);
- diritti necessari => dir(A);
- diritti obiettivo => scopo(A);

La clausola che informa sugli attacchi A eseguibili contro il sistema, su quale componente C e da parte di quale utente U è:

- `attOnBy(A,C,U).`

La clausola che si riferisce ai componenti del sistema vulnerabili è:

- `haVulne(C,X).`

Interrogando l'interprete Prolog con tale clausola si ottiene l'elenco di tutti quei componenti C che possiedono delle vulnerabilità. Queste ultime sono indicate con dei nomi simbolici (a,b,c,d,...) per poter astrarre dal significato reale. Un limite dello strumento è sicuramente quello di non approfondire il legame tra vulnerabilità ed attacchi, in quanto non è stato considerata la restrizione:

- l'attacco A1 è eseguibile se il componente C1 possiede la vulnerabilità V1.

bensì:

- l'attacco A1 è eseguibile se il componente C1 possiede qualche vulnerabilità.

Lo strumento controlla se la clausola haVulne(c1,X) unifica con qualche Fatto Prolog.
Perchè l'attacco a1 sia eseguibile è sufficiente che si verifichi una sola unificazione per quanto riguarda il componente c1.

Queste, in sintesi, tutte e sole le informazioni per il modulo di inserimento dati, a partire dal quale si potranno ottenere le rappresentazioni dei grafi delle dipendenze (ipergrafo) e delle evoluzioni:

1. utenti U;
2. componenti C;
3. vulnerabilità note dei componenti C;
4. diritti iniziali dell'utente U;

5. risorse usufruibili dall'utente U;
6. dipendenze note tra gli attributi dei componenti C;
7. obiettivi per ogni utente U;
8. diritti goal per ogni obiettivo di U;
9. attacchi A possibili su un determinato componente C;
10. scopo di ogni attacco A;
11. diritti necessari per ogni attacco A;
12. risorse necessarie per ogni attacco A.

Tutte le conoscenze esplicite del sistema sono state descritte mediante Fatti Prolog.
Per rappresentare l'ipergrafo delle dipendenze è necessario fornire un elenco di Fatti Prolog che descrivano nodi ed iperarchi attraverso le due clausole components(C) e implies(X,Y) descritte precedentemente. Interrogando l'interprete Prolog con la clausola relativa ai componenti si ottiene un unico passo di unificazione che restituisce la lista di tutti i nodi dell'ipergrafo che rappresentano dei componenti, mentre per quanto riguarda le dipendenze, e quindi gli iperarchi, l'interprete Prolog esegue più passi per unificare il termine implies(X,Y) con le variabili inserite nei Fatti Prolog che descrivono le dipendenze del sistema.

3.2 Output ottenuti

Questo, in sintesi, l'algoritmo alla base della simulazione del comportamento di ogni minaccia nel sistema:

1. per ogni minaccia M si inserisce un insieme di diritti iniziali;

2. si calcola la chiusura transitiva tra l'insieme dei diritti iniziali e le dipendenze note del sistema, ottenendo l'insieme dei diritti legali;

3. test del raggiungimento di un obiettivo di M:
   - se un obiettivo è stato raggiunto si segnala la terminazione con successo;
     si verifica se la minaccia M ha raggiunto altri obiettivi;
   - altrimenti (se non vengono raggiunti obiettivi) si considerano gli attacchi che la minaccia M esegue contro il sistema;

4. test sulla possibilità da parte della minaccia M di eseguire l'attacco A contro il sistema:
   - se M può eseguire l'attacco A allora M acquisisce i privilegi goal dell'attacco A più quelli ottenuti dal calcolo della chiusura transitiva tra il nuovo insieme di diritti e le dipendenze note del sistema;
     si controlla l’eventuale raggiungimento di un obiettivo da parte della minaccia M;
     si controlla se M può eseguire nuovi attacchi contro il sistema;
   - altrimenti (se M non può eseguire alcun attacco) si segnala il mancato raggiungimento dell'obiettivo da parte della minaccia M;
     si analizza la minaccia successiva.

Per ogni minaccia del sistema, a partire dagli input descritti in 3.1.1, viene calcolata la chiusura transitiva tra l'insieme dei diritti iniziali e le dipendenze note, controllando se i Fatti Prolog che descrivono le dipendenze permettono o meno l'espansione dell'insieme dei diritti della minaccia.
La chiusura transitiva verrà successivamente calcolata ogniqualvolta una minaccia esegue un nuovo

attacco: per ogni attacco la minaccia acquisisce tutti i relativi diritti goal, più quelli eventualmente deducibili dalle dipendenze note del sistema; a questo punto l'algoritmo verifica se la minaccia M ha raggiunto uno dei suoi obiettivi. Ciò è possibile grazie alla clausola

- `stopWithSuccess(U,X).`

che attraverso la clausola

- `sottinsieme(A,B).`

verifica se l'insieme dei diritti di ogni obiettivo è compreso in quello dei diritti finora raggiunti dalla minaccia, rispondendo semplicemente YES/NO. L'algoritmo procede verificando se la minaccia M è in grado di eseguire l'attacco A contro il sistema.

L'algoritmo è il seguente:

1. l'attacco A è eseguibile dalla minaccia M e sul componente C (A è un attacco noto già verificato contro il sistema oppure lo si ipotizza eseguibile).
   In questo caso l'interprete Prolog controllerà se le variabili A e M unificano con

- `attOnBy(A,C,M).`

2. il componente C, vittima dell'attacco, A deve possedere almeno una vulnerabilità.
   L'interprete controlla se esiste un fatto Prolog riguardante il componente C unificando con

- `haVulne(C,_).`

3. la minaccia M deve possedere tutti i diritti necessari per poter eseguire l'attacco A.
   L'interprete Prolog verifica la clausola

- `sottinsieme(dir(A),dirPreA(U)).` restituendo YES/NO;

4. la minaccia M deve possedere tutte le risorse necessarie per poter eseguire l'attacco A.

- `sottinsieme(ris(A),ris(U)).`

Le clausole che esprimono gli attacchi eseguiti dalle minacce sono:

- `attSucc1(M,A).` restituisce gli attacchi che hanno successo grazie ai soli diritti legali (privilegi iniziali più quelli ottenuti attraverso le dipendenze note del sistema);
- `attSucc2(M,A).` riconosce gli attacchi sequenziali, ossia quelli che, una volta eseguiti da parte della minaccia M, permettono l'esecuzione di nuovi attacchi;
- `attSuccTot(M,A).` è l'unione dei due tipi di attacchi precedentemente descritti.

In particolare gli insiemi utilizzati per descrivere i diritti acquisiti da ogni minaccia nelle diverse fasi della simulazione sono stati realizzati con il predicato elem (discusso in 2.1):

- dirPreA(U) rappresenta l'insieme dei diritti legali (vedi paragrafo 1.2) contenente tutti i diritti dell'utente prima dell'esecuzione di qualsiasi attacco;

- dirPost(U,A) contiene i diritti conseguenza dell'attacco A: l'unione tra l'insieme dei diritti legali della minaccia e quello dei diritti goal di A;

- dirEx(U,A) rappresenta l'insieme dei diritti estesi per ogni minaccia dopo l'esecuzione di tutti gli attacchi che è in grado di eseguire;

- dirEx2(U,A) contiene tutti i privilegi acquisiti dall'utente U fino all'esecuzione dell'attacco sequenziale A (compreso).

Tutti gli output della simulazione vengono salvati in opportuni database per poter essere successivamente recuperati ed analizzati; è possibile inoltre visualizzare tutti i risultati ottenuti grazie ad uno specifico modulo che restituisce le seguenti informazioni:

- INPUT:
  1. Utenti del sistema (minacce);
  2. Obiettivi di ogni utente;
  3. Diritti iniziali per ogni utente;
  4. Diritti goal per ogni obiettivo;
  5. Componenti del sistema;
  6. Componenti del sistema vulnerabili;
  7. Attacchi ipotizzati nel sistema;
  8. Scopo di ogni attacco (diritto/i goal).

- OUTPUT:
  1. Utenti che hanno raggiunto qualche obiettivo;
  2. Nuovi diritti (legali) acquisiti tramite dipendenze e prima di qualsiasi attacco;
  3. Attacchi eseguibili dalla minaccia U grazie ai soli diritti legali;
  4. Attacchi sequenziali per ogni utente U (oltre a quelli eseguibili con i diritti legali);
  5. Attacchi globali con successo nel sistema;
  6. Diritti ottenuti per ogni utente;
  7. Stati del grafo delle evoluzioni;
  8. GoalState del sistema;
  9. Evoluzioni del sistema;
  10. Stati generati dal prodotto cartesiano.

Viene inoltre fornito un menù per risalire alle principali informazioni ottenute dalla simulazione, come da Figura4:

SWI-Prolog -- c:/Documents and Settings/Mario/Desktop/TESI 29magg07/prolog/source/start.pl

File Edit Settings Run Debug Help

```
È possibile interrogare l interprete Prolog in due modalità:

- lasciando le variabili dei termini non istanziate (maiuscole)
per ottenere tutte le possibili deduzioni logiche;
- specificando le variabili del termine di cui si vuole dedurre informazione.

1. Per interrogare il database degli stati del sistema
risultanti da tutte le possibili evoluzioni:
elem((U,X),database(N)).

2. Per interrogare il database degli stati del sistema
ottenuti grazie al prodotto cartesiano tra i precedenti stati:
elem((U,X),pcDatabase(N)).

3. Per conoscere gli stati goal del sistema:
elem((U,N,A,G),goalState).

4. Per visualizzare le evoluzioni del sistema:
elem((U,A,N,G),evolution).

5. Per dedurre nodi e cammini del grafo delle evoluzioni
in seguito al calcolo del prodotto cartesiano:
elem(N,stateFrom(U,A,W,B)).

6. Per sapere se ci sono stati equivalenti nella computazione del grafo delle evoluzioni:
stateEquivalent(N,M).

7. Infine, per de-allocare tutti i database utilizzati:
svuota.


No
2 ?- █
```

Figura4: Menu interrogazione interprete

Le clausole

- `elem((U,X),database(N)).`
- `elem((U,X),pcDatabase(N)).`

consentono di interrogare i database per visualizzare tutte le coppie <utente,diritto> in un determinato stato. Entrambe le clausole fanno uso del meccanismo di backtracking per poter anche individuare se una determinata coppia, per esempio <u1,aC1>, è presente in qualche stato, restituendo tutti gli identificativi degli stati che contengono tale informazione.

Le clausole

- `elem((U,A,N,G),evolution).`
- `elem((U,N,A,G),goalState).`

restituiscono le informazioni rappresentate dal grafo delle evoluzioni (paragrafo 3.2.1).

Infine, per utilizzare lo strumento proposto e per poter visualizzare tutti i risultati della simulazione, è necessario eseguire il file start.pl (vedi allegato p.67) e fornire all'interprete Prolog il comando

- `start(U,X,A,B,C,N).` Le variabili inserite sono, rispettivamente, per gli utenti (U), i diritti (X), gli attacchi (A e B), i componenti (C) e gli stati del sistema (N) ottenuti nelle evoluzioni utente.

In particolare il meccanismo di backtracking che consente di richiamare tutte le clausole espresse nei moduli del progetto viene attivato premendo “;” oppure “N”.

3.2.1 Rappresentazione del grafo delle evoluzioni

Come discusso in 1.3.2 nel grafo delle evoluzioni vengono mostrati tutti i possibili cammini che portano le minacce al raggiungimento dei relativi stati goal, ovvero gli stati in cui sono contenuti tutti i diritti goal di un obiettivo della minaccia.
Il grafo delle evoluzioni è un DAG in cui i nodi rappresentano gli stati del sistema e gli archi gli attacchi che causano la transizione di stato.

Per rappresentare il grafo delle evoluzioni che mostra tutte le evoluzioni di tutte le minacce del sistema abbiamo tenuto conto dei seguenti vincoli:

- lo stato iniziale contiene tutte le coppie <utente,diritti_legali>;
- una minaccia può avere più obiettivi, ossia più insiemi di diritti da raggiungere;
- ogni minaccia provoca una transizione di stato se esegue un attacco che consente di acquisire nuovi diritti rispetto a quelli contenuti nello stato precedente;
- una evoluzione termina nel primo stato che contiene tutti i privilegi di un obiettivo della minaccia;
- nel grafo delle evoluzioni non possono comparire stati equivalenti (stati che contengono identiche coppie <utente,diritto>), dato che due stati equivalenti sono lo stesso stato.

A differenza dell'ipergrafo delle dipendenze, in cui sono stati utilizzati solamente Fatti Prolog per rappresentare nodi e archi, il grafo delle evoluzioni viene rappresentato mediante specifiche clausole che sfruttano il meccanismo di backtracking per astrarre nodi e archi di tale grafo.
Vediamo subito un esempio.

La clausola che restituisce tutte le possibili evoluzioni delle minacce è

- `elem((U,A,N,G),evolution).`

dove U sta per utente (o minaccia), A l'attacco eseguito dalla minaccia, N lo stato generato e G l'obiettivo della minaccia (o goal).

Questo esempio mostra il risultato della invocazione della clausola elem((U,A,N,G),evolution):

- U = u1
  A = null
  N = 1
  G = goal1(u1) ;

- U = u1
  A = a1
  N = 2
  G = goal1(u1) ;

- U = u1
  A = a2
  N = 4
  G = goal1(u1) ;

- U = u2
  A = null
  N = 1
  G = goal1(u2) ;
- U = u2
  A = a1
  N = 3
  G = goal1(u2) ;

- U = u2
  A = null
  N = 1
  G = goal2(u2) ;

- U = u2
  A = a1
  N = 3
  G = goal2(u2) ;

- U = u2
  A = a3
  N = 5
  G = goal2(u2) ;

- U = u2
  A = a4
  N = 6
  G = goal2(u2) ;

Lo stato finale dell'evoluzione è uno stato in cui la minaccia ha raggiunto uno dei suoi obiettivi (goalState) e si ottiene attraverso la clausola

- `elem((U,N,A,G),goalState).`

Per conoscere gli obiettivi e i relativi diritti goal di ogni minaccia è sufficiente interrogare l'interprete con

- `goal(U,G).` dove U è la minaccia e G gli obiettivi;
- `elem(X,g).` dove X sono i diritti dell'obiettivo g restituito dalla clausola precedente.

Nell'esempio sopra riportato, che mostrava il risultato della invocazione della clausola elem((U,A,N,G),evolution), si osservano tre evoluzioni, una per la minaccia u1 e due per u2.

Vediamole nel dettaglio.

1. u1 attraversa tre stati: parte da N = 1 stato iniziale, grazie all'attacco a1 raggiunge N = 2 e sfruttando l'attacco a2 transita nello stato N = 4, che è il suo goalState. Nello stato 4, quindi, la minaccia u1 ha raggiunto i diritti goal dell'obiettivo goal1(u1).

2. La prima evoluzione di u2 è composta da una sola transizione di stato: da N = 1, stato iniziale, a N = 3, goalState, grazie all'esecuzione dell'attacco a1. In N = 3, quindi, u2 ha raggiunto i diritti goal dell'obiettivo goal1(u2).

3. La seconda evoluzione per la minaccia u2 attraversa 4 stati: la transizione da N = 1 a N = 3 è causata dall'esecuzione dell'attacco a1, da N = 3 a N = 5 da a3, da N = 5 a N = 6 (goalState) da a4. In N = 6 la minaccia u2 ha raggiunto il secondo ed ultimo suo obiettivo: goal2(u2).

Il grafo delle evoluzioni, quindi, è formato da sei nodi (sei stati) e cinque archi, uno per attacco.
Le tre evoluzioni del sistema saranno, quindi, i cammini S1-S2-S4 per u1, S1-S3 e S1-S3-S5-S6 per quanto riguarda la minaccia u2.

Come anticipato in 1.3.2 la costruzione dei cammini di ogni minaccia (le evoluzioni) tiene conto del concetto di monotonicità. Infatti non compaiono stati che sono conseguenza di un attacco che non fa acquisire nuovi diritti alla minaccia che lo esegue, oppure uno stato contenente meno diritti dello stato sorgente.
Inoltre nel grafo delle evoluzioni non compaiono cicli tra i cammini del grafo e nemmeno coppie di stati equivalenti (due stati che contengono identiche coppie <utente,diritto>).

Infine, a partire dalle evoluzioni appena ricavate, è possibile calcolare il **prodotto cartesiano** tra gli stati generati dalle evoluzioni delle minacce. Ciò è utile per studiare le configurazioni del sistema dopo l'esecuzione di tutti gli attacchi delle minacce, fornendo un elenco di tutti i possibili stati del sistema (configurazioni).
Il prodotto cartesiano tra due insiemi A e B (A x B) si definisce come l'insieme di tutte le possibili coppie composte da un elemento di A e da un elemento di B.
Dato che uno stato del sistema nel nostro contesto non è altro che un insieme di coppie <minaccia,diritto>, il calcolo del prodotto cartesiano tra due stati del grafo S1 e S2 restituisce tutte le possibili coppie <minaccia,diritto_in_S1> e <minaccia,diritto_in_S2>, che equivale all'unione, senza ripetizioni, tra gli stati S1 e S2 (S1 U S2).
Per esempio, in una architettura con due utenti che eseguono attacchi, lo stato complessivo del sistema dipende dall'unione degli stati generati dalle due minacce. In altri termini chi si occupa della sicurezza del sistema è interessato ad avere una visione globale sulle azioni di tutti i singoli utenti. Un problema che si presenta in tutti i modelli che cercano di rappresentare tutte le conseguenze delle azioni delle minacce contro il sistema studiato è l'esplosione esponenziale degli stati in conseguenza del calcolo del prodotto cartesiano.
Una alternativa possibile potrebbe essere quella di analizzare un utente alla volta, il che semplificherebbe la modellazione, ma non fornirebbe all'analista la visione globale del sistema in conseguenza delle combinazioni di attacchi eseguiti dalle minacce.
La scelta adottata è stata quella di salvare in differenti database le informazioni delle evoluzioni degli utenti (stati del grafo delle evoluzioni) e quelle provenienti dall'unione degli stati precedentemente ottenuti. In questo modo si è in grado di reperire sia la rappresentazione del grafo delle evoluzioni, sia la simulazione di tutti gli stati possibili all'interno dell'infrastruttura.

CAPITOLO 4 : SIMULAZIONI EFFETTUATE

Questo capitolo presenta le tre simulazioni studiate: si inizia a considerare un contesto limitato, al fine di fornire una dimostrazione di base sul funzionamento dello strumento e sugli output restituiti, fino ad analizzare due infrastrutture di rete tipiche di un contesto aziendale di medio-piccole dimensioni. Nelle simulazioni verranno descritti componenti, dipendenze, vulnerabilità e attacchi delle infrastrutture studiate, nonché il comportamento delle singole minacce all'interno del sistema, gli insiemi dei privilegi raggiunti, ma soprattutto le evoluzioni generate: gli stati del grafo delle evoluzioni, le transizioni tra di essi (gli attacchi eseguiti dalle minacce), gli stati goal per ogni obiettivo, etc.
L'esecuzione degli attacchi da parte delle minacce contro il sistema permette l'acquisizione di determinati diritti. Il calcolo della chiusura transitiva tra l'insieme dei diritti raggiunto e le dipendenze note del sistema consente alle minacce di espandere il proprio insieme dei diritti iniziali fino all'eventuale raggiungimento dei relativi obiettivi, che sono degli insiemi di diritti goal.

4.1: Caso di studio 1: simulazione di uno scenario base

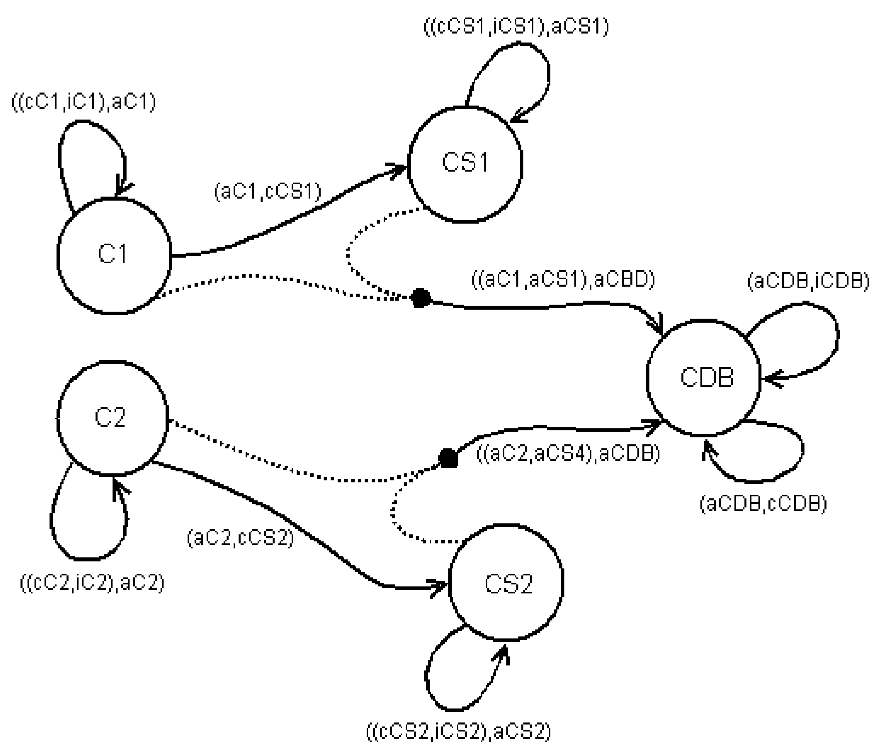

Figura5: Ipergrafo delle dipendenze

La prima prova effettuata analizza uno scenario limitato in cui l'architettura è composta da cinque componenti e gli utenti coinvolti sono le minacce u1 e u2. I componenti C1 e C2 sono due host di una LAN, CS1 e CS2 sono i due server che comunicano rispettivamente con C1 e C2, CDB è un database di cui le minacce vorrebbero acquisire il controllo.

Le dipendenze che si possono osservare in Figura5 denotano un particolare processo di acquisizione dei diritti: se una minaccia riesce ad ottenere il controllo del componente C1 (analoga situazione a partire dal componente C2) allora è abilitata a leggere informazioni sul server CS1; inoltre se acquisisce, grazie a due particolari dipendenze, il permesso di scrittura sullo stesso componente allora ottiene il controllo (disponibilità) sul server CS1 e sul database CDB. Vedremo in seguito come ogni minaccia riesce ad espandere il proprio insieme di privilegi grazie ad attacchi e dipendenze.

Interrogando l'interprete Prolog con le due clausole implies(X,Y) e components(X) si ottiene la rappresentazione dell'ipergrafo delle dipendenze (come da Figura5) che descrive la configurazione del sistema prima che le minacce eseguano gli attacchi. Si può notare che le dipendenze inserite sono di due tipi:

- semplici : diritto implica diritto (X = aC2; Y = cCS2);
- doppie : due diritti implicano un diritto (X = cCS2,iCS2; Y = aCS2).

C1, C2, CS1 e CS2 sono i quattro componenti vulnerabili del sistema contro cui verranno eseguiti i seguenti attacchi:

- [U = u1; A = a1; C = C1] con obiettivo il diritto iC1;
- [U = u2; A = a2; C = C2] con obiettivo il diritto iC2;
- [U = u1; A = a3; C = CS1] con obiettivo il diritto iCS1;
- [U = u2; A = a4; C = CS2] con obiettivo il diritto iCS2;

Le minacce u1 e u2 hanno le seguenti caratteristiche:

- partono entrambi dall'insieme dei diritti iniziali composto da (cC1,cC2);
- possiedono le stesse risorse, tali da consentire tutti gli attacchi ipotizzati nel sistema;
- non acquisiscono alcun nuovo diritto limitatamente alla prima chiusura transitiva, quindi l'insieme dei diritti iniziali coincide con quello dei diritti legali;
- hanno entrambi un solo obiettivo, denominato goal1(u1)/goal1(u2);
- l'obiettivo di entrambe le minacce è quello di acquisire il privilegio aCDB: ottenere il privilegio della disponibilità sul database.

Per raggiungere il relativo obiettivo, la minaccia u1 agisce sul componente C1 e, di conseguenza, sul server CS1, la minaccia u2 su C2 e, quindi, su CS2: la rappresentazione del grafo delle evoluzioni (Figura6) evidenzia il comportamento delle due minacce:

- a partire dallo stato iniziale S1 (contenente quattro coppie <utente,diritto> <u1,cC1>,<u1,cC2>,<u2,cC1>,<u2,cC2>) le due minacce u1 e u2 sono in grado di eseguire, grazie ai soli diritti legali, rispettivamente gli attacchi a1 e a2. Questa separazione dei cammini mette in risalto il concetto di evoluzione, ossia il percorso che porta una minaccia dallo stato iniziale, che contiene i soli diritti legali, allo stato goal, in cui sono contenuti i diritti goal di un obiettivo della minaccia (un solo obiettivo per minaccia, in questo caso);

- vengono generati gli stati S2 e S3 che contengono entrambe sette privilegi: cinque per la minaccia che causa la transizione di stato (S2 è conseguenza dell'attacco a1 da parte di u1; in S3 è u2 che esegue a2), dovuti all'attacco e al calcolo della seconda chiusura transitiva con le dipendenze note del sistema, più i due iniziali della rimanente minaccia; in S2 la minaccia u1 ha raggiunto il permesso di scrittura su C1 (iC1), grazie alle dipendenze note del sistema ottiene la disponibilità sullo stesso componente (aC1) e la confidenzialità sul server CS1 (cCS1); analogo comportamento per la minaccia u2 ma con i componenti C2 (iC2 e aC2)e CS2 (cCS2).

- vengono generati S4 e S5 frutto degli attacchi sequenziali a3 da parte di u1 e a4 per u2; infatti, solo i diritti acquisiti tramite i primi due attacchi a1 e a2 e le relative dipendenze consentono di effettuare gli attacchi sequenziali a3 e a4;

- per generare lo stato S4 la minaccia u1 esegue l'attacco a3 sul componente CS1. Scopo di a3 è il diritto iCS1 sullo stesso componente. Grazie alle dipendenze note del sistema u1 raggiunge la disponibilità sul server CS1 (aCS1) e, conseguentemente, il completo controllo del database con cui il server comunica (aCDB, cCDB, iCDB); analoga situazione per la minaccia u2, ma con l'attacco a4 sferrato su CS2 per acquisire così i privilegi iCS2, aCS2, aCDB, cCDB, iCDB.

- S4 e S5 sono riconosciuti come goalState, ossia gli stati che per primi contengono tutti i diritti goal di un obiettivo dell'utente (un solo obiettivo per minaccia in questa simulazione); le minacce non eseguono altri attacchi, quindi non vengono generati ulteriori cammini; tutte le evoluzioni possibili (due) sono state analizzate e memorizzate negli opportuni database: S1-S2-S4 per u1 e S1-S3-S5 per u2;

- entrambe gli utenti hanno raggiunto l'obiettivo del diritto aCDB che permette il pieno controllo del database;

- entrambe le minacce u1 e u2 passano dall'insieme dei diritti iniziali (contenuto nello stato S1) composto da soli due elementi a quello finale (gestito in S4 per u1 e S5 per u2) in cui, grazie ai due attacchi effettuati (a1-a3 per u1 e a2-a4 per u2) ed alle opportune dipendenze, i diritti acquisiti sono dieci;

- gli stati generati nel calcolo delle evoluzioni delle minacce sono cinque; tra di essi non esistono coppie di stati equivalenti (contenenti le stesse identiche coppie <utente,diritto>).

La simulazione appena descritta permette di capire inequivocabilmente il concetto di evoluzione utente: l'evoluzione di ogni minaccia non si arresta nel primo stato contenente qualche diritto goal di un obiettivo della minaccia, bensì in quel particolare stato (goalState) che per primo contiene tutti i privilegi di un obiettivo della minaccia.

La Figura6 descrive il grafo delle evoluzioni ottenuto in questa simulazione; gli stati doppiamente cerchiati rappresentano gli stati goal (goalState) delle due minacce.

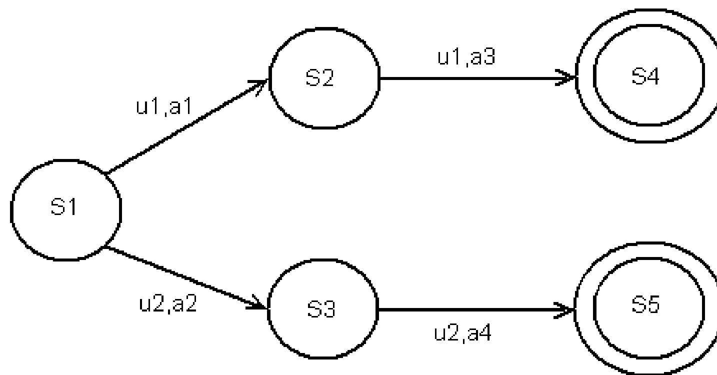

Figura6: Grafo delle evoluzioni

Al termine del calcolo delle evoluzioni si considerano le unioni tra gli stati ottenuti, ossia tutte le possibili combinazioni di stati, al fine di valutare il comportamento del sistema in seguito a scenari di attacco composti, come spiegato in 3.2.1.
Quello che si ottiene sono undici stati tra cui due coppie di stati equivalenti: <S4,S7> e <S5,S10>.
I primi stati di queste due coppie (S4 e S5) sono generati nel calcolo delle evoluzioni delle minacce e memorizzati in uno specifico database, database(N), mentre i secondi, risultato dell'unione tra gli stati del grafo delle evoluzioni, sono gestiti da pcDatabase(N). Questi ultimi stati sono ottenuti attraverso il seguente schema, che rappresenta tutte le possibili combinazioni, senza ripetizioni, tra gli stati del grafo delle evoluzioni (Figura7):

- S2 U S3 => S6; S2 U S4 => S7; S2 U S5 => S8;
- S3 U S4 => S9; S3 U S5 => S10;
- S4 U S5 => S11;

Gli stati generati dall'unione di due stati del grafo delle evoluzioni non compariranno mai nelle evoluzioni degli utenti, in quanto tutti i possibili scenari d'attacco sono già stati presi in considerazione, ma consentono di mantenere una traccia di tutte le possibili configurazioni del sistema in seguito ad attacchi eseguiti da minacce diverse e/o in sequenze diverse.
Per quanto riguarda le coppie di stati equivalenti, <S4,S7> e <S5,S10>, si nota subito che la sequenza di attacchi eseguita dalle minacce u1 e u2 per raggiungere gli stati S7 e S10 è la stessa per raggiungere gli stati S4 e S5; ne consegue che S7 e S10 sono gli stessi stati di S4 e S5, per cui non vengono rappresentati nel grafo delle evoluzioni.
I tagli evidenziati in Figura7 mostrano le evoluzioni utente da non considerare: i nodi finali, infatti, (potrebbero anche non rappresentare degli stati goal, ma non è il caso di questa simulazione) rappresentano quegli stati in cui la minaccia raggiunge i privilegi di un suo obiettivo eseguendo sequenze di attacchi già considerate negli stati precedentemente ottenuti (l'obiettivo di u2 raggiunto negli stati S8 e S11 tramite la sequenza a2-a4 è contenuto in S5). L'interprete Prolog deduce nodi e cammini di tale grafo attraverso la clausola elem(N,stateFrom(U,A,W,B)).

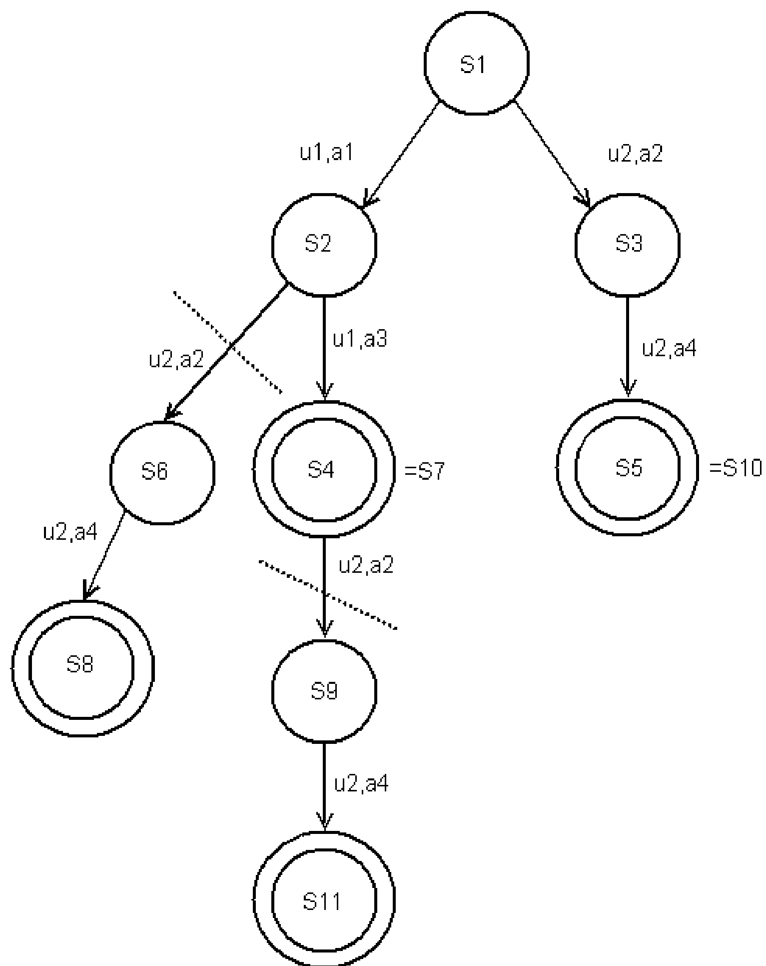

Figura7: Unione tra gli stati del grafo delle evoluzioni

4.2: Caso di studio 2: simulazione di uno scenario reale 1

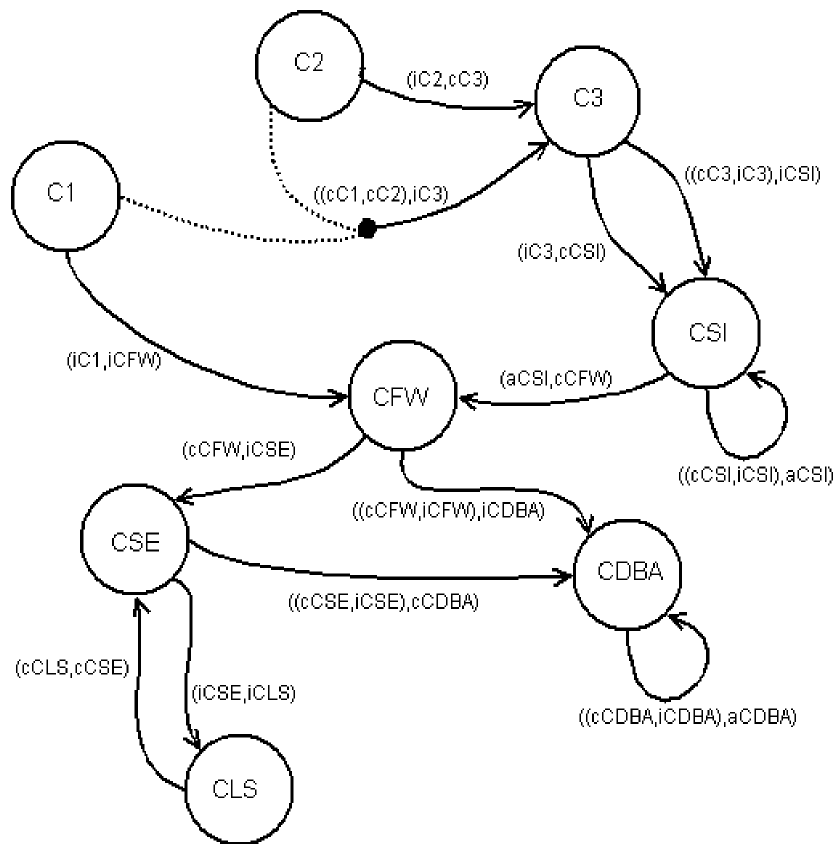

Figura8: Ipergrafo delle dipendenze

La seconda simulazione considera una infrastruttura informatica formata da otto componenti: tre nodi semplici (C1, C2 e C3), un server interno (CSI) in cui confluiscono le richieste dei tre client, un firewall (CFW) al centro dei collegamenti tra il server interno [29], il server esterno (CSE) per accedere alla rete esterna (internet) e il database aziendale (CDBA) contenente l'anagrafica dei dipendenti; il server esterno, inoltre, comunica con un log server (CLS) al fine di monitorare tutte le connessioni da parte degli utenti della rete aziendale verso internet.
Con questo esempio di infrastruttura si è cercato di simulare componenti, vulnerabilità, minacce ed attacchi a partire da un sistema reale, pur limitato nella sua complessità, ma che rappresenti un tipico contesto aziendale di dimensioni medio-piccole.
Inoltre le dipendenze tra diritti (quindi tra componenti) sono state pensate affinché le minacce acquisissero una serie di diritti a catena in modo da raggiungere i due componenti principali del sistema, il log server e il database. In questo modo si è voluto focalizzare l'attenzione sulla struttura del sistema, evidenziando così quelle vulnerabilità dei componenti e quelle dipendenze tra diritti che consentono alle minacce di eseguire gli attacchi necessari per acquisire privilegi sui componenti log server e database.

Questa la situazione iniziale:

- Le minacce che agiscono sul sistema sono tre: u1, u2 e u3.
- I componenti vulnerabili C1, CSI e CLS consentono alle minacce di eseguire gli attacchi elencati in seguito; come per la simulazione precedente si è scelto di lasciare dei nomi simbolici ai difetti dei componenti, ottenibili tramite la clausola haVulne(C,X).
  Per esempio l'attacco a2 viene eseguito contro il log server (componente CLS) con lo scopo di ottenere il permesso di lettura sullo stesso; la vulnerabilità in questo caso potrebbe nascere da una cattiva gestione delle password degli account administrator, o da una inadeguata protezione dei moduli che gestiscono le autenticazioni utente.
- Ogni minaccia ha un solo obiettivo, denominato goal1(U); l'obiettivo della minaccia u1, identico a quello di u3, è composto dai due privilegi iCLS e aCDBA, che sono i permessi di scrittura sul log server e della disponibilità sul database aziendale, mentre la minaccia u2 ha come obiettivo i permessi di scrittura su log server e database (iCLS e iCDBA) e della disponibilità sul firewall (aCFW).
- La minaccia u1 parte dall'insieme dei diritti iniziali formato dai soli (cC1,cC2), mentre sia per u2 che per u3 l'insieme è composto dai quattro privilegi (cC1,cC2,iC1,iC2).
- Le tre minacce possiedono le risorse necessarie per l'esecuzione degli attacchi ipotizzati.

Le dipendenze note del sistema, rappresentate dagli iperarchi dell'ipergrafo delle dipendenze, sono deducibili grazie alla clausola implies(X,Y); anche in questa simulazione si tratta di dipendenze semplici o doppie.

Gli attacchi ipotizzati nel sistema sono cinque:

- [U = u1; A = a1; C = CSI] con obiettivo il diritto iCSI;
- [U = u1 e U = u3; A = a2; C = CLS] con obiettivo il diritto cCLS;
- [U = u1; A = a3; C = C1] con obiettivo il diritto iC1;
- [U = u2; A = a4; C = CFW] con obiettivo il diritto aCFW;

Si considera ora il calcolo delle evoluzioni, studiando le fasi del sistema stato dopo stato, a partire da quello iniziale in cui i tre utenti sono in possesso dei diritti legali, fino all'eventuale stato goal.

- in S1 notiamo una situazione particolare, in quanto le due minacce u2 e u3, che partivano dall'insieme dei diritti iniziali (cC1,cC2,iC1,iC2), sfruttando le dipendenze note del sistema, acquisiscono quattordici diritti senza eseguire alcun attacco. In particolare la minaccia u2 acquisisce due dei tre diritti goal del suo obiettivo: i permessi di scrittura sul log server e sul database (iCLS e iCDBA); la minaccia u3, invece, acquisisce uno dei due diritti goal, il diritto iCLS.

A partire da S1 le tre minacce del sistema eseguono gli attacchi possibili grazie ai soli diritti legali:

- in S2 sono contenute le conseguenze di a1 da parte di u1;
- in S3 la minaccia u2 esegue l'attacco a4;
- in S4 è l'utente u3 che esegue l'attacco a2;

Da notare due aspetti in questa fase iniziale di attacchi:

- u2 esegue a4 e raggiunge il suo obiettivo nello stato S3. Lo stato S3 risulta il goalState per la minaccia u2 e sarà quindi il nodo finale della evoluzione di u2. I diritti acquisiti sono quindici, tra cui i tre diritti goal iCLS, iCDBA (scrittura su log server e database) e aCFW (disponibilità sul firewall).

- u3 esegue a2 e raggiunge il suo obiettivo nello stato S4 caratterizzato dai permessi di scrittura sul log server (raggiunto senza alcun attacco ma grazie alle sole dipendenze del sistema) e della disponibilità sul database; u2 ottiene diciotto diritti finali. S4 risulta goalState per u3 e sarà quindi il nodo finale dell'evoluzione della minaccia u3.

A questo punto le minacce eseguono gli attacchi sequenziali, come evidenzia la Figura9:

- in S5 sono contenuti i diritti conseguenza dell'attacco a2 da parte di u1; non tutti i diritti goal dell'obiettivo della minaccia u1 sono contenuti in questo stato;

- in S6 la minaccia u1 esegue l'attacco a3 e raggiunge il proprio obiettivo. Lo stato S6 è quindi il nodo finale della evoluzione della minaccia u1 (goalState).

Per capire il significato di goalState di una minaccia analizziamo il caso dell'utente u1:
grazie all'attacco a1 raggiunge nello stato S2 uno dei due privilegi cercati (iCLS) e in S6, grazie all'attacco a3, il secondo privilegio (aCDBA) dell'obiettivo denominato goal1(u1). Lo stato S6 è quindi l'unico goalState per la minaccia u1.

Questi i risultati ottenuti dalle tre minacce nella simulazione studiata:

- tutti e tre gli utenti raggiungono il relativo obiettivo;

- u1 esegue l'attacco a1 e gli attacchi sequenziali a2 e a3; in altri termini a3 è possibile solo se ha avuto successo a2, che, a sua volta, si può eseguire solo se la minaccia ha eseguito l'attacco a1; la minaccia u1 espande il proprio insieme di privilegi dai due iniziali ai sedici finali raggiunti nello stato goal S6; l'evoluzione di u1 risulta quindi S1-S2-S5-S6.

- u2 raggiunge il suo obiettivo grazie all'attacco a4 nello stato S3. La transizione di stato è quindi S1-S3, seconda evoluzione del sistema. S3 contiene quindici diritti acquisiti dalla minaccia u2 a partire dai quattro diritti iniziali e dai quattordici diritti legali contenuti in S1.

- u3 raggiunge l'unico suo obiettivo, goal1(u3), grazie ad un solo attacco, a2, e con una unica transizione di stato, S1-S4, terza ed ultima evoluzione del sistema; passa quindi dai quattro privilegi iniziali ai quattordici ottenuti grazie alle sole dipendenze del sistema, per arrivare ai diciotto finali raggiunti in S4, dopo l'esecuzione dell'attacco a2.

- le evoluzioni utente generano sei stati complessivi nel sistema (memorizzati in database(N)), tra cui non esistono coppie di stati equivalenti.

- l'unione tra gli stati ottenuti nel calcolo delle evoluzioni utente (gestita nel database pcDatabase(N)) genera gli stati S10 e S16 equivalenti a S6 e lo stato S9 che è equivalente a S5. S9, S10 e S16 non compaiono, quindi, nel grafo di Figura10 che rappresenta il calcolo dell'unione tra gli stati del grafo delle evoluzioni.

La Figura9 descrive il grafo delle evoluzioni ottenuto in questa seconda simulazione.

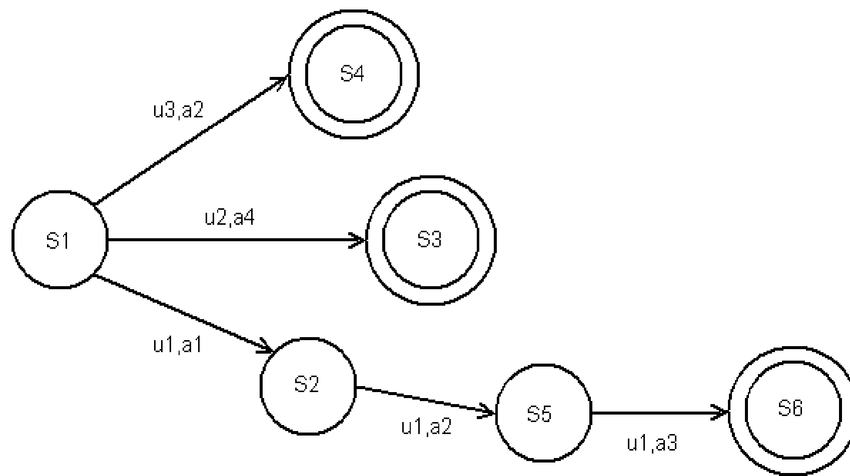

Figura9: Grafo delle evoluzioni

La Figura10, invece, rappresenta il risultato dell'unione tra i sei stati del grafo delle evoluzioni. Stesso schema della simulazione precedente per ottenere gli stati rappresentati in Figura10:

- S2 U S3 => S7; S2 U S4 => S8; S2 U S5 => S9; S2 U S6 => S10;
- S3 U S4 => S11; S3 U S5 => S12; S3 U S6 => S13;
- S4 U S5 => S14; S4 U S6 => S15;
- S5 U S6 => S16.

Gli stati equivalenti S9, S10 e S16 derivano da cammini ottenuti grazie alla stessa sequenza di attacchi a partire dallo stato iniziale S1.

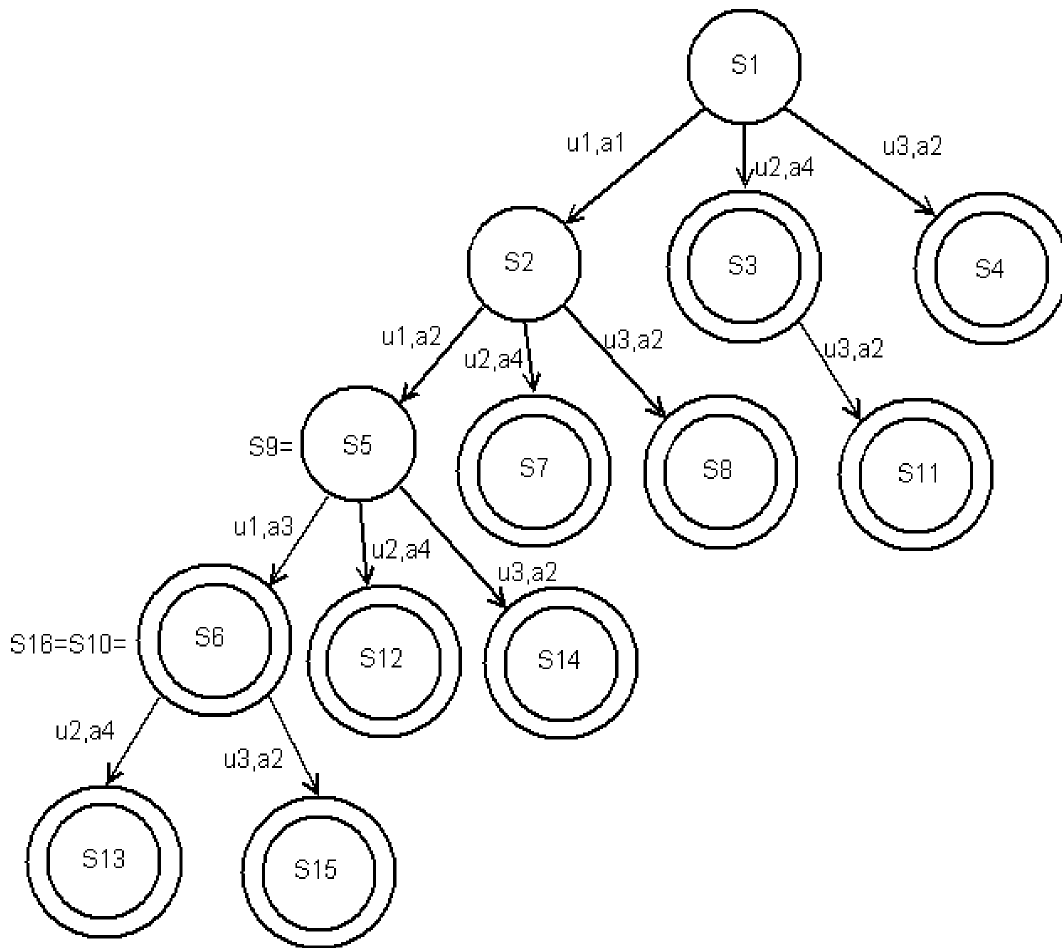

Figura10: Unione tra gli stati del grafo delle evoluzioni

4.3 Caso di studio 3: simulazione di un scenario reale 2

La terza ed ultima simulazione considera un altro esempio di infrastruttura informatica tipica di una rete aziendale medio-piccola.

L'ipergrafo delle dipendenze in Figura11 evidenzia tutte le dipendenze del sistema.

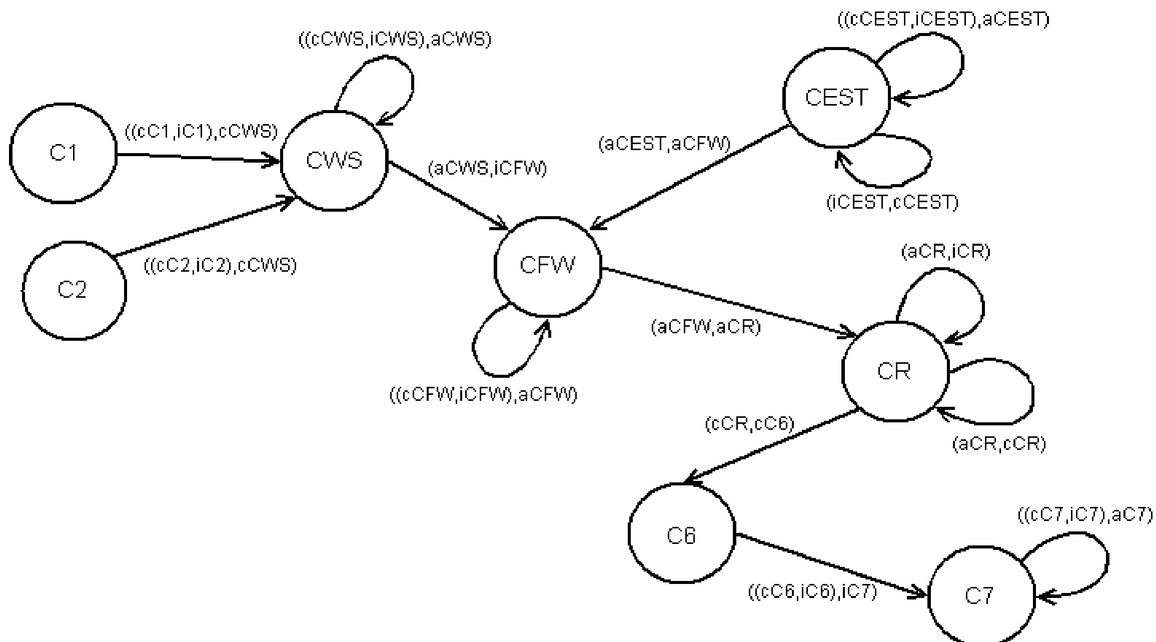

Figura11: Ipergrafo delle dipendenze

L'infrastruttura è composta da due client (C1 e C2) che comunicano con il web server (CWS), due host (C6 e C7), un router (CR), da cui passano tutte le richieste dei due host, un firewall (CFW) che separa web server e router ed un nodo esterno (CEST) di cui il firewall si fida.

Gli iperarchi del grafo i cui nodi sorgente sono anche nodi destinazione descrivono dipendenze che riguardano gli attributi di sicurezza dello stesso componente (vedi paragrafo 3.1.1).
Per esempio la dipendenza ((cC1,iC1),cCWS) significa che, se si possiedono i diritti in lettura e scrittura sul componente C1, allora si è in grado di leggere informazioni dal componente web server.

Gli utenti del sistema sono u1 e u2; a differenza delle precedenti simulazioni gli obiettivi delle minacce sono più di uno:

- u1 : goal1(u1) = (cCFW,aC7);
  admin = (aCWS,aCFW,aCEST,aCR);

- u2 : goal1(u2) = (iCFW);
  goal2(u2) = (aCEST,aC7);
  admin = (aCWS,aCFW,aCEST,aCR);

In particolare l'obiettivo admin (di entrambe le minacce u1 e u2) è un insieme contenente i diritti della disponibilità sui quattro componenti principali dell'infrastruttura. Le minacce che riescono a controllare i componenti web server CWS, firewall CFW, nodo esterno CEST e router CR sono quelle altamente pericolose perché in grado di controllare l'intero sistema.

I componenti del sistema vulnerabili sono CWS, CFW, C6 e CEST.
Gli attacchi ipotizzati nel sistema sono otto, quattro da parte della minaccia u1 e, specularmente, quattro dall'utente u2. Come vedremo non tutti gli attacchi elencati di seguito saranno sfruttabili dalle minacce, questo per l'assenza di eventuali diritti o risorse.

- [U = u1/u2; A = a1; C = CWS] con obiettivo il diritto iCWS;
- [U = u1/u2; A = a2; C = CFW] con obiettivo il diritto cCFW;
- [U = u1/u2; A = a3; C = CEST] con obiettivo il diritto iCEST;
- [U = u1/u2; A = a4; C = C6] con obiettivo il diritto iC6;

Analizziamo ora il comportamento del sistema in conseguenza degli attacchi eseguiti dalle minacce u1 e u2; si ottiene la rappresentazione del grafo delle evoluzioni mostrato in Figura12.

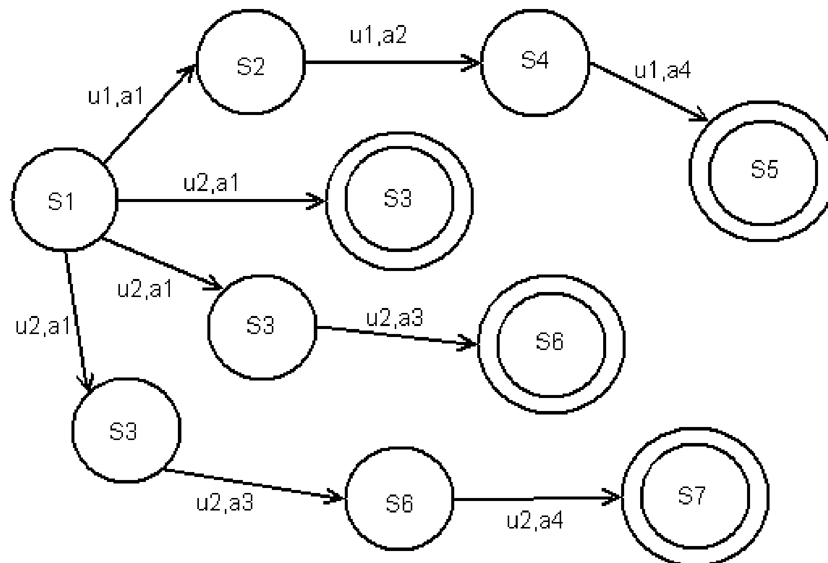

Figura12: Grafo delle evoluzioni

- Nello stato iniziale S1 entrambe le minacce possiedono quattro diritti, i tre iniziali (cC1,iC1,cC7) per u1 e (cC2,iC2,cC7) per u2, più il diritto cCWS, acquisito tramite dipendenze, che permette loro di raggiungere il web server in sola lettura e che consente alle due minacce di eseguire l'attacco a1.

- Grazie ai diritti legali entrambe le minacce u1 e u2 possono eseguire l'attacco a1 e raggiungere rispettivamente gli stati S2 e S3.
  In S2 la minaccia u1 espande il proprio insieme di privilegi da quattro a dodici, tra i quali il diritto della disponibilità sui componenti router, web server e firewall, ossia tre dei quattro privilegi necessari per raggiungere l'obiettivo admin.
  In S3 l'insieme dei diritti della minaccia u2 si espande da quattro a sette; in S3 viene raggiunto il primo degli obiettivi di u2, goal1(u2), caratterizzato dal permesso di scrittura sul firewall (iCFW). S3 è quindi uno dei goalState per la minaccia u2 e comparirà come nodo finale di un cammino nel grafo delle evoluzioni.

- S4 e S5 sono due stati generati dalla sequenza di attacchi a2 e a4 da parte della minaccia u1 (l'attacco a3 non è invece eseguibile per l'assenza del diritto cC2). L'insieme dei privilegi di u1 contiene in S5 sedici diritti. Lo stato S5 è l'unico goalState della minaccia u1: tutti i possibili attacchi sono stati eseguiti e l'unico obiettivo raggiunto risulta essere goal1(u1), composto dai due diritti cCFW e aC7.
  Il secondo obiettivo della minaccia u1 è il già discusso admin, che però non viene raggiunto per l'assenza del diritto aCEST.

- S6 è lo stato generato dalla minaccia u2 grazie all'esecuzione dell'attacco a3 (l'attacco a2 non risulta usufruibile per l'assenza del diritto cC1). In S6 viene raggiunto un altro obiettivo dell'utente u2, il goal admin caratterizzato dalla disponibilità dei quattro componenti CWS, CFW, CEST e CR.

- S7 è l'ultimo stato generato nel calcolo delle evoluzioni utente. Contiene le conseguenze dell'attacco a4 da parte della minaccia u2, tra cui i privilegi aC7 e aCEST utili al raggiungimento del terzo ed ultimo obiettivo denominato goal2(u2).
  Tutti i possibili attacchi per la minaccia u2 sono stati eseguiti; l'insieme finale dei privilegi raggiunti è composto da diciannove elementi.

Riassumendo, nel calcolo delle evoluzioni di ogni minaccia si ottengono sette stati.
Le evoluzioni ottenute sono quattro: S1-S2-S4-S5 per u1, S1-S3, S1-S3-S6, S1-S3-S6-S7 per u2.
Lo stato S5, goalState per la minaccia u1, contiene l'obiettivo goal1(u1); S3, S6 e S7 sono i tre goalState per u2 in cui vengono raggiunti rispettivamente gli obiettivi goal1(u2), admin e goal2(u2).
Il particolare obiettivo admin discusso ad inizio paragrafo viene raggiunto solo dalla minaccia u2.

L'unione tra tutti gli stati del grafo delle evoluzioni genera ventidue stati, ottenuti seguendo lo stesso schema delle precedenti simulazioni:

- S2 U S3 => S8; ... ; S2 U S7 => S12;
- S3 U S4 => S13; ... ; S3 U S7 => S16;
- S4 U S5 => S17; ... ; S4 U S7 => S19;
- S5 U S6 => S20; S5 U S7 => S21;
- S6 U S7 => S22.

Gli stati da considerare in realtà sono solo sedici in quanto sei tra questi ventidue sono equivalenti

ad alcuni precedentemente ottenuti nel calcolo delle evoluzioni: S9 è equivalente a S4, S10 e S17 sono equivalenti a S5, S15 a S6, S16 e S22 a S7. Tutte le coppie <utente,diritto> degli stati risultanti dall'unione tra quelli ottenuti nel calcolo delle evoluzioni utente vengono memorizzate nel database pcDatabase(N).
Interrogando l'interprete Prolog con la clausola elem(N,stateFrom(U,A,W,B)) è possibile capire da quali minacce e da quali attacchi sono generati i cammini per arrivare allo stato N.

Con questa simulazione abbiamo voluto studiare una tipica infrastruttura informatica in cui le azioni delle minacce sono finalizzate ad acquisire il controllo completo del sistema.
L'infrastruttura descritta rappresenta una valida base di partenza per condurre nuove simulazioni al variare di minime configurazioni del sistema in termini di dipendenze tra diritti, oppure aggiungendo ipotesi di attacco finora non considerate. Tipici scenari potrebbero essere quelli in cui l'attacco parte dal web server, passa al firewall e raggiunge il nodo interno, vero obiettivo della minaccia. Oppure quello di un nodo esterno di cui il firewall si fida: attaccando il nodo esterno si riesce a bypassare il firewall ed attaccare i componenti della rete interna.

CAPITOLO 5 : CONCLUSIONI E SVILUPPI FUTURI

L'analisi del rischio (Risk Analysis) è utile per studiare minacce e vulnerabilità di una infrastruttura informatica e per capire come investire un insieme limitato di risorse a difesa delle aree più a rischio.
Lo studio effettuato in questa tesi ha portato alla realizzazione di uno strumento per poter simulare strategie alternative di attacchi contro una infrastruttura informatica formata da componenti che condividono informazioni e/o risorse. Lo studio integra il processo di Risk Analysis facendo in modo che la probabilità di un attacco con successo nel sistema e l'impatto che ne può derivare (di conseguenza il rapporto che esiste tra le vulnerabilità di un componente e gli obiettivi di una minaccia) non siano più una deduzione arbitraria e soggettiva dell'analista, ma possano essere calcolati a partire da un insieme di informazioni ottenute attraverso il processo di simulazione.
La simulazione condotta in questo studio consiste nella modellazione di due grafi: quello delle **dipendenze** (ipergrafo) e quello delle **evoluzioni**.
Le dipendenze note del sistema consentono alle minacce di espandere il proprio insieme dei diritti iniziali e di raggiungere il relativo obiettivo. Componenti e dipendenze delle infrastrutture studiate sono rappresentate attraverso l'**ipergrafo delle dipendenze**. Ogni utente del sistema (o minaccia) possiede dei diritti iniziali e degli obiettivi da conseguire. Per raggiungere gli obiettivi le minacce eseguono degli attacchi contro il sistema; per ogni attacco è necessario verificare che le sue precondizioni siano rispettate: che esistano delle vulnerabilità nel componente vittima dell'attacco e che risorse e diritti necessari per poter eseguire l'attacco siano in possesso della minaccia. La sequenza di attacchi che consente alla minaccia di raggiungere un obiettivo, ossia un insieme di diritti goal, viene detta evoluzione. Le evoluzioni delle minacce sono rappresentate mediante il **grafo delle evoluzioni**.
Per rappresentare l'ipergrafo delle dipendenze e il grafo delle evoluzioni e per condurre la simulazione degli attacchi delle minacce contro il sistema abbiamo adottato un linguaggio logico.
"La disponibilità di strumenti automatici è importante non solo per ridurre il tempo di implementazione dell'assessment, ma soprattutto per garantire che nessuna evoluzione possa essere tralasciata. Tutto ciò risulta fondamentale per infrastrutture di larga scala, dato che uno dei passi più complessi della stesura dell'assessment è appunto il calcolo di tutte le evoluzioni" [29].
La scelta dello strumento di programmazione è ricaduta sul principale linguaggio della programmazione logica, il **Prolog**, e questo per due ragioni: perché viene offerto il meccanismo di backtracking come "native feature" e perché è possibile gestire tutte le caratteristiche del sistema che fungono da input allo strumento in maniera semplice e funzionale. In particolare l'utilizzo dei Fatti Prolog ha permesso di elencare le seguenti informazioni (ricavate o da uno studio preliminare del sistema, cioè prima che le minacce eseguano i relativi attacchi, o dalle ipotesi dell'analista): utenti (o minacce) e componenti del sistema, vulnerabilità note dei componenti, diritti/risorse iniziali delle minacce, dipendenze note tra gli attributi dei componenti, obiettivi per ogni utente, diritti goal per ogni obiettivo, attacchi delle minacce contro un determinato componente, scopo di ogni attacco, diritti/risorse necessari/e per ogni attacco. Inoltre viene sfruttato il meccanismo di backtracking per esplorare completamente lo spazio delle soluzioni, ossia per ottenere le evoluzioni causate da ogni singolo utente, per esplorare tutti i cammini possibili del grafo delle evoluzioni e per dedurre tutti i privilegi di una minaccia. L'insieme dei diritti che una minaccia riesce a raggiungere si ottiene grazie al calcolo della chiusura transitiva tra l'insieme dei diritti di una minaccia e le dipendenze note del sistema, ripetendo questa operazione dopo l'esecuzione di ogni attacco da parte delle minacce.
Il programma è formato da un insieme di clausole che sono applicate alle strutture dati che rappresentano l'ipergrafo delle dipendenze e i diritti degli utenti, lasciando allo strumento il compito

di dedurre in modo automatico, ripetibile e verificabile il comportamento del sistema in conseguenza di vulnerabilità, dipendenze, minacce e attacchi. In altri termini il programma implementa un motore di inferenza indipendente dall'infrastruttura che funge invece da input. Anche se il programma deve essere aggiornato ad ogni modifica dell'ipergrafo delle dipendenze, un notevole vantaggio di questa scelta implementativa è il tempo di esecuzione del programma per ottenere tutte le deduzioni logiche.

Lo strumento realizzato in questa tesi può essere utilizzato per un'estesa sperimentazione con riferimento a casi reali. Approfondimenti potrebbero riguardare la definizione delle vulnerabilità dei componenti in gioco, descritte in questa tesi astraendo dal significato reale. Un limite dello strumento sviluppato, accennato in 3.1.1, è sicuramente quello di non approfondire il legame tra vulnerabilità ed attacchi: allo stato attuale, infatti, una delle pre-condizioni per eseguire un attacco contro il sistema è che il componente su cui viene eseguito l'attacco possieda almeno una vulnerabilità, non specificando quale vulnerabilità sia necessaria per quel determinato attacco.
Un altro approfondimento potrebbe riguardare la definizione di dipendenza tra diritti senza porre alcun limite ai diritti sorgenti della dipendenza, come accennato in 1.3.1. Inoltre si potrebbe considerare il caso di una minaccia che fornisce/sottrae privilegi alle altre minacce del sistema, oppure il caso descritto in 1.3.2 dell'inserimento di eventuali contromisure che riportano la minaccia ad uno stato precedente.

APPENDICE

PSC - Prolog Source Code

La versione utilizzata dell'interprete SWI-Prolog è la Multi-threaded 5.6.22.
Il codice dello strumento è aggiornato a 02/06/2007.

insiemistiche.pl

```
%----------------------------------------------------
%% CLAUSOLE UTILIZZATE PER LE RELAZIONI INSIEMISTICHE|
%----------------------------------------------------

%% CLAUSOLE CHE OTTENGONO DALL'INTERPRETE SI/NO
%% --------------------------------------------
   nonSottinsieme(A,B) :-
      elem(X,A), not(elem(X,B)).

   sottinsieme(A,B) :-
      not(nonSottinsieme(A,B)).

   vuoto(A) :-
      not(elem(_,A)).

   diversi(A,B) :-
      elem(X,A), not(elem(X,B)).
   diversi(A,B) :-
      elem(X,B), not(elem(X,A)).

   uguali(A,B) :-
      not(diversi(A,B)).

   intersecati(A,B) :-
      elem(X,A), elem(X,B).

   disgiunti(A,B) :-
      not(intersecati(A,B)).


%% CLAUSOLE PER INDIVIDUARE PARTICOLARI ELEMENTI DELL'INSIEME
%% ----------------------------------------------------------
   elem(X,unione(A,B)) :-
      elem(X,A).
   elem(X,unione(A,B)) :-
      elem(X,B), not(elem(X,A)).

   elem(X,intersezione(A,B)) :-
      elem(X,A), elem(X,B).

   elem(X,differenza(A,B)) :-
      elem(X,A), not(elem(X,B)).
   elem(X,differenza(A,B)) :-
      elem(X,B), not(elem(X,A)).
```

```
%-------------------------------------------------------
%% INPUT PER LA MODELLAZIONE DI UN PIANO DI ATTACCO |
%-------------------------------------------------------

%% input per le clausole insiemistiche
:- include(insiemistiche).

%% COMPONENTI DEL SISTEMA
%% ----------------------
components([c1,c2,cS1,cS2,cDB]).

%% VULNERABILITA' DEI COMPONENTI
%% -----------------------------
haVulne(c1,(a,b)).
haVulne(c2,(c,d)).
haVulne(cS1,(e,f)).
haVulne(cS2,(g,h)).

%% DIPENDENZE TRA DIRITTI
%% ----------------------

   %% DIPENDENZE SEMPLICI
%% ----------------------
implies(aC2,cCS2).
implies(aC1,cCS1).
implies(aCDB,cCDB).
implies(aCDB,iCDB).

   %% DIPENDENZE "DOPPIE"
%% ----------------------
implies((cCS2,iCS2),aCS2).
implies((cCS1,iCS1),aCS1).
implies((aC2,aCS2),aCDB).
implies((aC1,aCS1),aCDB).
implies((cC1,iC1),aC1).
implies((cC2,iC2),aC2).

   %% DIRITTI INIZIALI DEI VARI UTENTI
%% ----------------------------------
elem(cC1,dirIn(u1)).
elem(cC2,dirIn(u1)).
userPossiedeDir(u1,dirIn(u1)).
elem(cC1,dirIn(u2)).
elem(cC2,dirIn(u2)).
userPossiedeDir(u2,dirIn(u2)).

   %% DIRITTI GOAL DEI VARI UTENTI
%% ------------------------------
elem(aCDB,goal1(u1)).
goal(u1,goal1(u1)).
elem(aCDB,goal1(u2)).
goal(u2,goal1(u2)).

   %% RISORSE A DISPOSIZIONE DEI VARI UTENTI
%% --------------------------------------
elem(a,ris(u1)).
elem(b,ris(u1)).
userPossiedeRis(u1,ris(u1)).
elem(a,ris(u2)).
```

```
elem(b,ris(u2)).
userPossiedeRis(u2,ris(u2)).

   %% DIRITTI NECESSARI PER I VARI ATTACCHI
%% ---------------------------------------
elem(cC1,dir(a1)).
attRichiedeDir(a1,dir(a1)).
elem(cC2,dir(a2)).
attRichiedeDir(a2,dir(a2)).
elem(aC1,dir(a3)).
attRichiedeDir(a3,dir(a3)).
elem(aC2,dir(a4)).
attRichiedeDir(a4,dir(a4)).

   %% RISORSE NECESSARIE PER I VARI ATTACCHI
%% ----------------------------------------
elem(a,ris(a1)).
elem(b,ris(a1)).
attRichiedeRis(a1,ris(a1)).
elem(a,ris(a2)).
elem(b,ris(a2)).
attRichiedeRis(a2,ris(a2)).
elem(a,ris(a3)).
elem(b,ris(a3)).
attRichiedeRis(a3,ris(a3)).
elem(a,ris(a4)).
elem(b,ris(a4)).
attRichiedeRis(a4,ris(a4)).

   %% DIRITTO GOAL PER OGNI ATTACCO
%% -------------------------------
scopo(a1,iC1).
scopo(a2,iC2).
scopo(a3,iCS1).
scopo(a4,iCS2).

   %% ATTACCHI-COMPONENTI-UTENTI
%%----------------------------
%% L'attacco a1 è diretto sul componente c1 ed eseguito da u1, se può.
attOnBy(a1,c1,u1).
attOnBy(a2,c2,u2).
attOnBy(a3,cS1,u1).
attOnBy(a4,cS2,u2).
```

---

input2.pl

```
%-----------------------------------------------------
%% INPUT PER LA MODELLAZIONE DI UN PIANO DI ATTACCO |
%-----------------------------------------------------

%% input per le clausole insiemistiche
:- include(insiemistiche).

%% COMPONENTI DEL SISTEMA
components([c1,c2,c3,cSI,cLS,cFW,cSE,cDBA]).

%% VULNERABILITA' DEI COMPONENTI
haVulne(c1,(j)).
haVulne(cSI,(x,y)).
haVulne(cLS,(z,w)).
haVulne(cFW,(k)).

%% DIPENDENZE TRA DIRITTI
%% ----------------------
   %% DIPENDENZE SEMPLICI
%% ----------------------
implies(iC2,cC3).
implies(cCLS,cCSE).
implies(iCSE,iCLS).
implies(aCSI,cCFW).
implies(cCFW,iCSE).
implies(iC1,iCFW).
implies(iC3,cCSI).
   %% DIPENDENZE "DOPPIE"
%% ----------------------
implies((cC1,cC2),iC3).
implies((cC3,iC3),iCSI).
implies((cCSI,iCSI),aCSI).
implies((cCFW,iCFW),iCDBA).
implies((cCSE,iCSE),cCDBA).
implies((cCDBA,iCDBA),aCDBA).

   %% DIRITTI INIZIALI DEI VARI UTENTI
%% ----------------------------------
elem(cC1,dirIn(u1)).
elem(cC2,dirIn(u1)).
userPossiedeDir(u1,dirIn(u1)).
elem(cC1,dirIn(u2)).
elem(cC2,dirIn(u2)).
elem(iC1,dirIn(u2)).
elem(iC2,dirIn(u2)).
userPossiedeDir(u2,dirIn(u2)).
elem(cC1,dirIn(u3)).
elem(cC2,dirIn(u3)).
elem(iC1,dirIn(u3)).
elem(iC2,dirIn(u3)).
userPossiedeDir(u3,dirIn(u3)).

   %% DIRITTI GOAL DEI VARI UTENTI
%% ------------------------------
elem(iCLS,goal1(u1)).
elem(aCDBA,goal1(u1)).
goal(u1,goal1(u1)).
elem(iCLS,goal1(u2)).
elem(iCDBA,goal1(u2)).
elem(aCFW,goal1(u2)).
```

```
goal(u2,goal1(u2)).
elem(iCLS,goal1(u3)).
elem(aCDBA,goal1(u3)).
goal(u3,goal1(u3)).

   %% RISORSE A DISPOSIZIONE DEI VARI UTENTI
%% ------------------------------------------
elem(a,ris(u1)).
elem(b,ris(u1)).
elem(c,ris(u1)).
userPossiedeRis(u1,ris(u1)).
%%vuoto(u2).
elem(d,ris(u2)).
elem(e,ris(u2)).
userPossiedeRis(u2,ris(u2)).
elem(a,ris(u3)).
elem(c,ris(u3)).
userPossiedeRis(u3,ris(u3)).

   %% DIRITTI NECESSARI PER I VARI ATTACCHI
%% -----------------------------------------
elem(cCSI,dir(a1)).
attRichiedeDir(a1,dir(a1)).
elem(iCLS,dir(a2)).
attRichiedeDir(a2,dir(a2)).
elem(cCDBA,dir(a3)).
attRichiedeDir(a3,dir(a3)).
elem(iCLS,dir(a4)).
elem(iCDBA,dir(a4)).
attRichiedeDir(a4,dir(a4)).

   %% RISORSE NECESSARIE PER I VARI ATTACCHI
%% ------------------------------------------
elem(a,ris(a1)).
elem(b,ris(a1)).
attRichiedeRis(a1,ris(a1)).
elem(a,ris(a2)).
elem(c,ris(a2)).
attRichiedeRis(a2,ris(a2)).
elem(a,ris(a3)).
elem(b,ris(a3)).
elem(c,ris(a3)).
attRichiedeRis(a3,ris(a3)).
elem(d,ris(a4)).
elem(e,ris(a4)).
attRichiedeRis(a4,ris(a4)).

   %% DIRITTO GOAL PER OGNI ATTACCO
%% --------------------------------
scopo(a1,iCSI).
scopo(a2,cCLS).
scopo(a3,iC1).
scopo(a4,aCFW).

   %% ATTACCHI-COMPONENTI-UTENTI
%% -----------------------------
attOnBy(a1,cSI,u1).
attOnBy(a2,cLS,u1).
attOnBy(a2,cLS,u3).
attOnBy(a3,c1,u1).
attOnBy(a4,cFW,u2).
```

input3.pl

```
%-----------------------------------------------------
%% INPUT PER LA MODELLAZIONE DI UN PIANO DI ATTACCO |
%-----------------------------------------------------

%% input per le clausole insiemistiche
:- include(insiemistiche).

%% COMPONENTI DEL SISTEMA
%% ----------------------
components([cWS(webServer),cFW(firewall),cEST(nodoEsterno),cR(router),c1,c2,c6,c
7]).

%% VULNERABILITA' DEI COMPONENTI
%% -----------------------------
haVulne(cWS,(a)).
haVulne(cFW,(b)).
haVulne(c6,(c)).
haVulne(cEST,(d)).

%% DIPENDENZE ESPLICITE TRA COMPONENTI
%% -----------------------------------

   %% DIPENDENZE SEMPLICI
%% ----------------------
implies(aCWS,iCFW).
implies(aCEST,aCFW).
implies(aCFW,aCR).
implies(aCR,iCR).
implies(aCR,cCR).
implies(iCEST,cCEST).
implies(cCR,cC6).

   %% DIPENDENZE "MULTIPLE"
%% ----------------------
%%implies((aCWS,aCFW,aCEST,aCR),admin).
implies((cC1,iC1),cCWS).
implies((cC2,iC2),cCWS).
implies((cCWS,iCWS),aCWS).
implies((cCFW,iCFW),aCFW).
implies((cC6,iC6),iC7).
implies((cC7,iC7),aC7).
implies((cCEST,iCEST),aCEST).

   %% DIRITTI INIZIALI DEI VARI UTENTI
%% ----------------------------------
elem(cC1,dirIn(u1)).
elem(iC1,dirIn(u1)).
elem(cC7,dirIn(u1)).
userPossiedeDir(u1,dirIn(u1)).

elem(cC2,dirIn(u2)).
elem(iC2,dirIn(u2)).
elem(cC7,dirIn(u2)).
userPossiedeDir(u2,dirIn(u2)).

   %% OBIETTIVI (INSIEMI DI DIRITTI GOAL) DEI VARI UTENTI
%% ----------------------------------------------------
elem(cCFW,goal1(u1)).
elem(aC7,goal1(u1)).
```

```
goal(u1,goal1(u1)).

elem(aCWS,admin).
elem(aCFW,admin).
elem(aCEST,admin).
elem(aCR,admin).
goal(u1,admin).

elem(iCFW,goal1(u2)).
goal(u2,goal1(u2)).

elem(aCEST,goal2(u2)).
elem(aC7,goal2(u2)).
goal(u2,goal2(u2)).

elem(aCWS,admin).
elem(aCFW,admin).
elem(aCEST,admin).
elem(aCR,admin).
goal(u2,admin).

   %% RISORSE A DISPOSIZIONE DEI VARI UTENTI
%% ----------------------------------------
elem(a,ris(u1)).
elem(b,ris(u1)).
userPossiedeRis(u1,ris(u1)).

elem(a,ris(u2)).
elem(b,ris(u2)).
userPossiedeRis(u2,ris(u2)).

   %% DIRITTI NECESSARI PER I VARI ATTACCHI
%% ---------------------------------------
elem(cCWS,dir(a1)).
attRichiedeDir(a1,dir(a1)).

elem(cC1,dir(a2)).
elem(iCFW,dir(a2)).
attRichiedeDir(a2,dir(a2)).

elem(cC2,dir(a3)).
elem(iCFW,dir(a3)).
attRichiedeDir(a3,dir(a3)).

elem(cC6,dir(a4)).
attRichiedeDir(a4,dir(a4)).

   %% RISORSE NECESSARIE PER I VARI ATTACCHI
%% ----------------------------------------
elem(a,ris(a1)).
attRichiedeRis(a1,ris(a1)).

elem(b,ris(a2)).
attRichiedeRis(a2,ris(a2)).

elem(a,ris(a3)).
elem(b,ris(a3)).
attRichiedeRis(a3,ris(a3)).

elem(a,ris(a4)).
elem(b,ris(a4)).
attRichiedeRis(a4,ris(a4)).
```

```
   %% DIRITTO GOAL PER OGNI ATTACCO
%% --------------------------------
scopo(a1,iCWS).
scopo(a2,cCFW).
scopo(a3,iCEST).
scopo(a4,iC6).

   %% ATTACCHI-COMPONENTI-UTENTI
%%------------------------------
attOnBy(a1,cWS,u1).
attOnBy(a1,cWS,u2).
attOnBy(a2,cFW,u1).
attOnBy(a2,cFW,u2).
attOnBy(a3,cEST,u1).
attOnBy(a3,cEST,u2).
attOnBy(a4,c6,u1).
attOnBy(a4,c6,u2).
```

```
%---------------------------------------------------
%% REGOLE DI DEDUZIONE LOGICA PER LA RAPPRESENTAZIONE |
%%         DELL'IPERGRAFO DELLE DIPENDENZE            |
%---------------------------------------------------

%% input per l'ipergrafo delle dipendenze
:- include(input3).

c :- consult(dipendenze).

%% attacchi eseguibili da parte di U grazie ai soli diritti legali
attSucc1(U,A) :-
   user(U),
   attOnBy(A,C,U),
   haVulne(C,_),
   sottinsieme(ris(A),ris(U)),
   sottinsieme(dir(A),dirPreA(U)).


%% attacchi sequenziali per ogni utente U (solo quelli sequenziali)
attSucc2(U,B) :-
   user(U),
   attSucc1(U,A),
   (
      attOnBy(B,C,U),
      A\=B, haVulne(C,_),
      sottinsieme(ris(B),ris(U)),
      sottinsieme(dir(B),dirEx(U,_))
   ).


%% tutti gli attacchi con successo nel sistema, ordinati per successione
%% temporale
attSuccTot(U,A) :-
   user(U),
   attOnBy(A,C,U),
   haVulne(C,_),
   sottinsieme(ris(A),ris(U)),
   sottinsieme(dir(A),dirEx(U,_)).


%% diritti di U prima di qualsiasi attacco
elem(X,dirPreA(U)):-
   user(U),
   (
      (elem(X,dirIn(U)));
      (implies(Y,X), elem(Y,dirPreA(U)));
      (implies((Y,Z),X), elem(Y,dirPreA(U)), elem(Z,dirPreA(U)));
      (implies((Y,Z,W),X), elem(Y,dirPreA(U)), elem(Z,dirPreA(U)),
elem(W,dirPreA(U)));
      (implies((Y,Z,W,J),X), elem(Y,dirPreA(U)), elem(Z,dirPreA(U)),
elem(W,dirPreA(U)), elem(J,dirPreA(U)))
   ).


%% diritti di U dopo l'attacco A (iniziali, dipendenze, attacco, dipendenze)
elem(X,dirPost(U,A)) :-
   not(attOnBy(A,_,U)) -> fail;
   (elem(X,dirPreA(U)));
```

```
   (implies(Y,X), elem(Y,dirPost(U,A)), not(elem(X,dirPreA(U))));
   (implies((Y,Z),X), elem(Y,dirPost(U,A)), elem(Z,dirPost(U,A)),
not(elem(X,dirPreA(U))));
   (implies((Y,Z),X), elem(Y,dirPost(U,B)), elem(Z,dirPost(U,A)),
not(elem(X,dirPreA(U))), B\=A);
   (scopo(A,X), attSuccTot(U,A), not(elem(X,dirPreA(U))));
   (implies((Y,Z,W,J),X), elem(Y,dirPost(U,_)), elem(Z,dirPost(U,_)),
elem(W,dirPost(U,_)), elem(J,dirPost(U,_))).


%% diritti estesi di U (dopo ogni singolo attacco A da parte di U)
elem(X,dirEx(U,A)):-
   not(attSucc1(U,A)) *->
      elem(X,dirPreA(U));
      elem(X,dirPost(U,A)).


%% diritti estesi di U (dopo tutti gli attacchi da parte di U fino ad A)
elem(X,dirEx2(U,A)):-
   elem(X,dirPost(U,A));
   (once(attSuccTot(U,B)), A=B) *->
      fail;
      (
         attSuccTot(U,A),
         (
            attSuccTot(U,B) *->
            (
               A\=B -> (elem(X,dirPost(U,B)), not(elem(X,dirPost(U,A))));
                        ( elem(X,dirPost(U,B)),not(elem(X,dirPost(U,A)));
                          implies((Y,Z),X), elem(Y,dirPost(U,A)),
elem(Z,dirPost(U,B)),
                          not(elem(X,dirPreA(U))), not(elem(X,dirPost(U,A))),
not(elem(X,dirPost(U,B)))
                        );
               A=B  -> (!,fail)
            );
            fail
         )
      ).


%% utenti del sistema
user(U) :-
   userPossiedeDir(U,dirIn(U)).


%% utenti del sistema che raggiungono il proprio goal
stopWithSuccess(U,X):-
   user(U),
   goal(U,X),
   sottinsieme(X,dirEx(U,A)).
```

```
%-------------------------------------------------------
%% REGOLE DI DEDUZIONE LOGICA PER LA RAPPRESENTAZIONE |
%%          DEL GRAFO DELLE EVOLUZIONI                |
%-------------------------------------------------------


%% input per il grafo delle evoluzioni
:- include(dipendenze).

c :- consult(evoluzioni).

%% attSuccDif(U,A) :- user(U), A=a5.


%% 1: Stato Iniziale S(1): tutti i diritti degli utenti prima di qualsiasi
%% attacco
elem((U,X), s(N)) :-
   get_val_contatore(N),
   assert(db3(utente,stato,attacco,obiettivo)),
   assert(db4(N)),
   write('\nStart Transizioni di Stato'),
   write('\n**************************'),
   user(U), (A = null),
   (
      assert(db2(U,N,A)),
      elem(X,dirPreA(U)), assert(db(U,X,N))
      %%(once(attOnBy(_,_,U)), elem(X,dirPreA(U)));
      %%(not(attOnBy(_,_,U)), elem(X,dirEx(U,_)))
   );
   (user(U), goal(U,G), sottinsieme(G,dirPreA(U))) ->
      (A = null), get_val_contatore(N), assert(db3(U,N,A,G)), writeln('\n***
           GoalState per:');
   (user(U), goal(U,G), not(attOnBy(_,_,U)), elem(X,dirEx(U,_)),
sottinsieme(G,dirEx(U,_))) ->
      (A = null), get_val_contatore(N), assert(db3(U,N,A,G)), writeln('\n***
GoalState per:').


%% 2: Stati S(i), con 1<i<N, dove N sono i passi di backtracking:
%% un nuovo stato per ogni attacco eseguibile con i soli diritti legali.
%% N.B. funzione monotona.
elem((U,X), s(N)) :-
   attSucc1(R,A),
   (
      write('\n*******************'),
      write('\nTransizione di Stato'),
      write('\n*******************'),
      incrementa_contatore,
      get_val_contatore(N),
      assert(db4(N)),
      user(U),
      (
         (U=R, not(db3(R,_,A,_)), assert(db2(U,N,A)));
         (U=R, elem(X,dirEx(R,A)), assert(db(U,X,N)));
         (U=S, S\=R, elem(X,dirPreA(S)), assert(db(U,X,N)))
      );
      (U=R, goal(R,G), sottinsieme(G,dirEx(R,A)), not(db3(R,_,_,G))) ->
         get_val_contatore(N), assert(db3(R,N,A,G)), writeln('\n*** GoalState
per:')
```

```
   ).


%% 3: N passi di backtracking per ogni attacco sequenziale da parte di U.
%% negli attacchi sequenziali la funzione monotona si verifica automaticamante
%% per gli utenti che eseguono tali attacchi, mentre per gli altri utenti viene
%% forzata manualmente.
elem((U,X), s(N)) :-
   user(R), once(attSuccTot(R,A1)),
   attSuccTot(R,A2),
   A1\=A2,
   (
      write('\n*******************'),
      write('\nTransizione di Stato'),
      write('\n*******************'),
      incrementa_contatore,
      get_val_contatore(N),
      assert(db4(N)),
      user(U),
      (
         (U=R, assert(db2(U,N,A2)), elem(X, unione(dirPost(R,A1),
dirEx2(R,A2))), assert(db(U,X,N)));
         (U=S, S\=R, elem(X,dirPreA(S)), assert(db(U,X,N)))
      );
      (U=R, goal(R,G), not(db3(R,_,_,G)), sottinsieme(G,
unione(dirPost(R,A1),dirEx2(R,A2)))) ->
         get_val_contatore(N), assert(db3(R,N,A2,G)), writeln('\n*** GoalState
per:')
   ).


elem((U,X),s2(N)) :-
   db4(F), db4(G), F>=2, F<G,
   (
      write('\n*******************'),
      write('\nTransizione di Stato'),
      write('\n*******************'),
      incrementa_contatore,
      get_val_contatore(N),
      assert(db6(N)),
      assert(state(N,F,G)), %%EDITED HERE
      elem((U,X),unione(database(F),database(G))),
      assert(db5(U,X,N))
   ).


memorizzaDB(U,X,N):-
   assert(state(n,f,g)),
   inizializza_contatore,
   elem((U,X),s(N));
   writeln('\n***************'),
   writeln('Stop Transizioni'),
   writeln('***************'),
   writeln('\nNumero totale degli Stati del sistema:'),
   cont(N);
   writeln('\n*** Prodotto Cartesiano tra gli Stati del Sistema:'),
   elem((U,X),s2(N));
   writeln('\n***************'),
   writeln('Stop Transizioni'),
   writeln('***************'),
   writeln('\nNumero di Stati dopo il Prodotto Cartesiano:'),
   cont(N).
```

```
elem((U,N,A,G),goalState) :- db3(U,N,A,G).

elem((U,A,N,G),evolution) :-
   writeln('\nPer ogni utente U, lo stato finale della evoluzione è il suo
goalState;'),
   writeln('\nX = utente,attacco,stato,obiettivo;'),
   goal(U,G),
   (
      writeln('\n---'),
      db2(U,N,A), not(db3(U,N,A,G)), db4(N), not(stateEquivalent(_,N)),
db3(U,M,_,G), N=<M;
      %%db2(U,N), stateEquivalent(N,M), N>=M, db4(N);
      db3(U,N,A,G), db4(N), not(stateEquivalent(_,N))
   ).


elem((U,A,N),evol) :-
      elem((U,A,N,_),evolution).

elem(N,numState) :-
      db4(N).

elem(N,pcNumState) :-
      db6(N).

elem(N,stateFrom(U,A,W,B)) :-
      db2(U,F,A), db2(W,G,B), state(N,F,G).

elem((U,X),database(N)) :-
      db(U,X,N).

elem((U,X),pcDatabase(N)) :-
      db5(U,X,N).

stateEquivalent(N,M) :-
   db4(N),
   (
      db4(M),
      N<M,
      uguali(database(N),database(M))
   );
   db6(N),
   (
      db6(M),
      N<M,
      uguali(pcDatabase(N),pcDatabase(M))
   );
   db4(N), db6(M), uguali(database(N),pcDatabase(M)).

svuota :-
   retractall(cont(N)),
   retractall(db(U,X,N)),
   retractall(db2(U,N,A)),
   retractall(db3(U,N,A,G)),
   retractall(db4(N)),
   retractall(db5(U,N)),
   retractall(db6(N)),
   retractall(state(N,F,G)),
   writeln('\nTutte le strutture dati utilizzate sono state deallocate.').
```

```
inizializza_contatore :-
   assert(cont(1)).

incrementa_contatore :-
   retract(cont(N)),
   N1 is N + 1,
   assert(cont(N1)).

get_val_contatore(N) :-
   cont(N).
```

---

start.pl

```
%-------------------------------------------------------
%% CLAUSOLA start(U,X,A,B,C,N) PER LA VISUALIZZAZIONE |
%%          DEGLI OUTPUT DELLA MODELLAZIONE            |
%-------------------------------------------------------


%% input per la visualizzazione dei risultati della modellazione
:- include(evoluzioni).

start :- writeln('\nPer avviare il processo di Backtracking digitare:'),
         writeln('"start(U,X,A,B,C,N)"').

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln(''),
   writeln(''),
   writeln('****************************************************'),
   writeln('* OUTPUT DELLA MODELLAZIONE DI UN PIANO DI ATTACCO *'),
   writeln('*    REALIZZATO TRAMITE GRAFO DELLE DIPENDENZE     *'),
   writeln('*            E GRAFO DELLE EVOLUZIONI              *'),
   writeln('****************************************************'),
   writeln(''),
   writeln(''),
   writeln('PREMERE N PER FORZARE IL MECCANISMO DI BACKTRACKING '),
   writeln(''),
   writeln(''),
   writeln('*****************************'),
   writeln('Utenti del sistema (minacce):'),
   writeln('*****************************'),
   user(U).

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln('*************************'),
   writeln('Obiettivi di ogni utente:'),
   writeln('*************************'),
   goal(U,X).

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln('*********************************'),
   writeln('Diritti goal per ogni obiettivo:'),
   writeln('*********************************'),
   goal(_,A),
   elem(X,A).

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln('*********************************************'),
   writeln('Utenti che hanno raggiunto qualche obiettivo:'),
   writeln('*********************************************'),
   stopWithSuccess(U,X).

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln('**********************************'),
   writeln('Diritti iniziali per ogni utente:'),
   writeln('**********************************'),
   elem(X,dirIn(U)).
```

```
start(U,X,A,B,C,N) :-
   writeln(''),
   writeln('***************************************************'),
   writeln('Nuovi diritti (legali) acquisiti tramite dipendenze'),
   writeln('e prima di qualsiasi attacco <TC(G)>              :'),
   writeln('***************************************************'),
   (user(U), elem(X,differenza(dirIn(U), dirPreA(U))));
   (uguali(dirIn(U),dirPreA(U)), writeln('\nNessun diritto proveniente da
dipendenze.')),
   (user(U), goal(U,G), sottinsieme(G,dirPreA(U))) ->
      (
         writeln('\nUtente U che ha già raggiunto il proprio Goal'),
         writeln('attraverso diritti legali e dipendenze:')
      ).

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln('***********************'),
   writeln('Componenti del sistema:'),
   writeln('***********************'),
   components(X).

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln('***********************************'),
   writeln('Componenti del sistema vulnerabili:'),
   writeln('***********************************'),
   haVulne(C,_).

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln('****************************'),
   writeln('Dipendenze note del sistema:'),
   writeln('****************************'),
   implies(X,N).

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln('*********************************'),
   writeln('Attacchi ipotizzati nel sistema:'),
   writeln('*********************************'),
   attOnBy(A,C,U).

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln('***************************************'),
   writeln('Scopo di ogni attacco (diritto/i goal):'),
   writeln('***************************************'),
   scopo(X,A).

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln('********************************************************************'
),
   writeln('Attacchi eseguibili dalla minaccia U grazie ai soli diritti
legali:'),
   writeln('********************************************************************'
),
   attSucc1(U,A).

start(U,X,A,B,C,N) :-
```

```
   writeln(''),
   writeln('******************************************************'),
   writeln('Attacchi sequenziali per ogni utente U                '),
   writeln('(oltre a quelli eseguibili con i soli diritti legali):'),
   writeln('******************************************************'),
   attSucc2(U,B).

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln('******************************************'),
   writeln('Attacchi globali con successo nel sistema:'),
   writeln('******************************************'),
   attSuccTot(U,A).

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln('*********************************'),
   writeln('Diritti ottenuti per ogni utente:'),
   writeln('*********************************'),
   (user(U),elem(X,dirEx(U,_)));
   writeln(''),
   goal(U,N),
   stopWithSuccess(U,N) *->
      (writeln(''), writeln('Utente U che ha raggiunto un proprio obiettivo
N:'));
      writeln('Nessun utente ha raggiunto obiettivi.').

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln(''),
   writeln('*********************************'),
   writeln('Stati del Grafo delle Evoluzioni:'),
   writeln('*********************************'),
   memorizzaDB(U,X,N).

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln('**********************'),
   writeln('GoalState del sistema:'),
   writeln('**********************'),
   elem((U,N,A,C),goalState).

start(U,X,A,B,C,N) :-
   writeln(''),
   writeln('***********************'),
   writeln('Evoluzioni del sistema:'),
   writeln('***********************'),
   elem((U,A,N,C),evolution);
   writeln('---'),
   writeln(''),
   writeln(''),
   writeln('È possibile interrogare l interprete Prolog in due modalità:'),
   writeln(''),
   writeln('- lasciando le variabili dei termini non istanziate (maiuscole)'),
   writeln('per ottenere tutte le possibili deduzioni logiche;'),
   writeln('- specificando le variabili del termine di cui si vuole dedurre
informazione.'),
   writeln(''),
   writeln('1. Per interrogare il database degli stati del sistema'),
   writeln('risultanti da tutte le possibili evoluzioni:'),
   writeln('elem((U,X),database(N)).'),
   writeln(''),
```

```
   writeln('2. Per interrogare il database degli stati del sistema'),
   writeln('ottenuti grazie al prodotto cartesiano tra i precedenti stati:'),
   writeln('elem((U,X),pcDatabase(N)).'),
   writeln(''),
   writeln('3. Per conoscere gli stati goal del sistema:'),
   writeln('elem((U,N,A,G),goalState).'),
   writeln(''),
   writeln('4. Per visualizzare le evoluzioni del sistema:'),
   writeln('elem((U,A,N,G),evolution).'),
   writeln(''),
   writeln('5. Per dedurre nodi e cammini del grafo delle evoluzioni'),
   writeln('in seguito al calcolo del prodotto cartesiano:'),
   writeln('elem(N,stateFrom(U,A,W,B)).'),
   writeln(''),
   writeln('6. Per sapere se ci sono stati equivalenti nella computazione del
grafo delle evoluzioni:'),
   writeln('stateEquivalent(N,M).'),
   writeln(''),
   writeln('7. Infine, per de-allocare tutti i database utilizzati:'),
   writeln('svuota.'),
   writeln(''),
   writeln(''),
   writeln(''),
   writeln('~~~ End of Program ~~~'),
   writeln(''),
   writeln(''),
   writeln(''),
   writeln('####################################################################'),
   writeln('# Source Code by Mario Chilosi <mario.chilosi@gmail.com>          #'),
   writeln('# Last Update on 02/06/2007, h 11.33 am                            #'),
   writeln('# SWI-Prolog (Multi-threaded, Version 5.6.22)                      #'),
   writeln('# Copyright (c) 1990-2006 University of Amsterdam                  #'),
   writeln('# SWI-Prolog comes with *ABSOLUTELY NO WARRANTY*                   #'),
   writeln('# This is Free Software,                                           #'),
   writeln('# and you are welcome to redistribute it under conditions          #'),
   writeln('# Please visit http://www.swi-prolog.org for details               #'),
   writeln('####################################################################'),
   fail.
```

BIBLIOGRAFIA

1. **SWI-Prolog's Home**
   www.swi-prolog.org/

2. **SWI-Prolog downloads**
   http://www.swi-prolog.org/download.html

3. **SWI-Prolog 5.6.32 Reference Manual**
   http://gollem.science.uva.nl/SWI-Prolog/Manual/

4. **Prolog Tutorial © J.R.Fisher**
   http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html

5. **Esempi di programmazione Prolog**
   http://www.lix.polytechnique.fr/~catuscia/didattica/progr_Prolog.html

6. **Insiemi e Prolog**
   http://xoomer.alice.it/roberto-ricci/articoli/loginfo/insiemi.htm

7. **Guida SWI-Prolog**
   www.cs.unibo.it/~riccucci/Teaching%20Material/Scienze%20della%20Formazione/Guida%20SWI-Prolog1.pdf

8. **Programmazione logica e Prolog**
   http://www-lia.deis.unibo.it/Books/libro_pl/

9. **Intelligenza Artificiale I**
   http://www.disi.unige.it/person/MascardiV/Didattica/materialeMaurizio

10. **Visual Prolog**
    http://www.visual-prolog.com/

11. **SEM242 - Prolog and Logic Programming**
    http://www.cs.bham.ac.uk/~pjh/prolog_course/sem242.html

12. **Logic Programming**
    http://vl.fmnet.info/logic-prog/

13. **CMU Prolog Repository**
    http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/lang/prolog/0.html

14. **Fault Tree Analysis – FTA**
    http://www.fault-tree.com/

15. **Information Security - Holger Schlingloff**
    www2.informatik.hu-berlin.de/~hs/Lehre/2001-WS_ITSec/L2.pdf

16. **Risk Analysis, Risk Assessment, Risk Management**
http://www.nr.no/~abie/RiskAnalysis.htm

17. **Sicurezza Reti: L'analisi del rischio**
http://www.shinynews.it/ebusiness/1105-qualita-servizio-3.shtml

18. **Introduction to Risk Analysis**
http://www.security-risk-analysis.com/introduction.htm

19. **CIP - Critical Infrastructure Protection**
http://www.nr.no/~abie/CIP.htm

20. **The BS7799 Security Standard**
http://www.riskserver.co.uk/bs7799/

21. **FMEA - Failure Mode and Effects Analysis**
http://www.fmeainfocentre.com/gyilyui

22. **ASSET Automated Security Self-Evaluation Tool**
http://csrc.nist.gov/asset/

23. **Centro Serra – Secur-Net**
http://www.serra.unipi.it/sicurezza/base2006-1.pdf

24. F. Baiardi, C.Telmon
Average Impact of Attacks on Billing Infrastructures
MMM-ACNS 2005, Mathematical Models and Methods for Advance Computer Network Security, St. Petersburg, September 2005

25. F. Baiardi
Vulnerabilities, Attacks and Equilibrium in Information Infrastructures
Proceedings of the ENEA International Workshop on Complex Networks and Infrastructure Protection, Roma, March 2006

26. F. Baiardi, L. Ricci, P. Mori, A. Vaccareli
Short Paper: Policy Driven Virtual Machine Monitor for Protected Grids
15th IEEE International Symposium on High Performance Distributed Computing, Paris, France, June 2006

27. F. Baiardi
Allocating Resources to the Search for Vulnerabilities In Information Infrastructures
Proceedings of NATO Research Workshop on Computational Models of Risk to Infrastructure, 9-13 May 2006

28. F. Baiardi, F. Martinelli, L. Ricci, C. Telmon
Constrained Automata: a Formal Tool for ICT Risk Assessment
Proceedings of NATO Avanced Research Workshop on Information Security and Assurance, June 2005

29. F. Baiardi, S. Suin, C. Telmon
Assessing the Risk of an Information Infrastructure through Security Dependencies
First international workshop on critical information infrastructure security, Samo, Sept 2006

30. F. Baiardi, S. Suin, C. Telmon
A Mathematical Framework to Assess the Security of an Information Infrastructure
First italian workshop on privacy and security, Roma, June 2006

31. S. Noel, S. Jajodia, B. O'Berry, M. Jacobs
Efficient Minimum-Cost Network Hardening Via Exploit Dependency Graphs
Center for Secure Information Systems, George Mason University, Fairfax VA 22030, U.S.A. Proceedings of 19th Annual Computer Security Applications Conference, Las Vegas, Nevada, December 2003

32. S. Noel, E. Robertson, S. Jajodia
Correlating Intrusion Events and Building Attack Scenarios Through Attack Graph Distances
Center for Secure Information Systems, George Mason University, Fairfax VA 22030, U.S.A. Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04) - Volume 00, p. 350 - 359, 2004, ISBN ~ ISSN:1063-9527 , 0-7695-2252-1, IEEE Computer Society Washington, DC, USA

33. S. Noel, S. Jajodia
Managing Attack Graph Complexity Through Visual Hierarchical Aggregation
Center for Secure Information Systems, George Mason University, Fairfax, VA 22030, U.S.A. Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security, Washington DC, USA. p. 109 - 118, 2004, ISBN:1-58113-974-8

34. S. Noel, M. Jacobs, P. Kalapa, S. Jajodia
Multiple Coordinated Views for Network Attack Graphs
Proceedings of the IEEE Workshops on Visualization for Computer Security, Minneapolis, Minnesota, October 2005, p. 12, 2005, ISBN:0-7803-9477-1

35. S. Noel, B. O'Berry, C. Hutchinson, S. Jajodia, L. Keuthan, and A. Nguyen
Combinatorial Analysis of Network Security
Proceedings of SPIE -- Volume 4738, Wavelet and Independent Component Analysis Applications IX, Harold H. Szu, James R. Buss, Editors, March 2002, p. 140-149

36. L. Wang, C. Yao, A. Singhal, S. Jajodia
Interactive Analysis of Attack Graphs Using Relational Queries
Proceedings of the 20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security, Springer Lecture Notes in Computer Science, Vol. 4127, Ernesto Damiani and Peng Liu, editors, Sophia Antipolis, France, July 31 - August 2, 2006, p. 119-132

37. L. Wang, A. Liu, S. Jajodia
Using Attack Graphs for Correlating, Hypothesizing, and Predicting Network Intrusion Alerts
Computer Communications 29 (2006) 2917–2933, Center for Secure Information Systems, George Mason University, Fairfax, VA 22030-4444, USA. Available online at www.sciencedirect.com from 25 April 2006

38. P. Ammann, D. Wijesekera, S. Kaushik
Scalable, Graph-Based Network Vulnerability Analysis
ISE Department, Center for Secure Information Systems, George Mason University, Fairfax, VA 22030, U.S.A. In Proceedings CCS 2002: 9th ACM Conference on Computer and Communications Security, Washington, DC, November 2002. p. 217-224

39. Oleg M. Sheyner
Scenario Graphs and Attack Graphs
CMU-CS-04-122, April 14, 2004, School of Computer Science, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA

40. Christopher C. Yang, Xiaodong Shi
Discovering Event Evolution Graphs from Newswires
Department of System Engineering and Engineering Management, The Chinese University of Hong Kong. In Proceedings of the 15th International Conference on World Wide Web (Edinburgh, Scotland, May 23 - 26, 2006). WWW '06. ACM Press, New York, NY, 945-946

41. Christopher J. Alberts, Sandra G. Behrens, Richard D. Pethia, William R. Wilson
Operationally Critical Threat, Asset, and Vulnerability Evaluation (OCTAVE) Framework, Version 1.0
Networked Systems Survivability Program, Technical Report CMU/SEI-99-TR-017, ESC-TR-99-017, June 1999

42. B. D. Jenkins
Risk Analysis Helps Establish a Good Security Posture; Risk Management Keeps it that Way
1998, Countermeasures, Inc
http://www.nr.no/~abie/RA_by_Jenkins.pdf

43. Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein and Charles E. Youman
Role-Based Access Control Models
IEEE Computer, Volume 29, Number 2, February 1996, p. 38-47.
Revised October 26, 1995
http://csrc.nist.gov/rbac/sandhu96.pdf

44. M. Koch, L.V. Mancini and F. Parisi
A Graph-Based Formalism for RBAC
ACM Transactions on Information and System Security (TISSEC), Volume 5 , Issue 3 August 2002, p. 332 – 365, 2002, ISSN:1094-9224

45. P. Rotondo
Network-Based Vulnerability Assessment
Pier Luigi Rotondo IT Specialist, IBM Tivoli Rome Laboratory
http://cesare.dsi.uniroma1.it/Sicurezza/doc/vulnerability.pdf

46. D. Balzarotti, M. Monga, S. Sicari
Assessing the Risk of Using Vulnerable Components
Quality of protection: security measurements and metrics, Advances in Information Security 23 Springer, New York, 2006

47. R. Lutz
Adapting Safety Requirements Analysis to Intrusion Detection
Jet Propulsion Laboratory, and Department of Computer Science, Iowa State University
http://www.sreis.org/old/2001/papers/sreis013.pdf

48. G. Helmer, J. Wong, M. Slagell, V. Honavar, L. Miller, R. Lutz
A Software Fault Tree Approach to Requirements Analysis of an Intrusion Detection System
Department of Computer Science, 226 Atanasoff Hall, Iowa State University, Ames, Iowa 50011
http://www.sreis.org/old/2001/papers/sreis005.pdf

49. K. Daley, R. Larson, J. Dawkins
A Structural Framework for Modeling Multi-Stage Network Attacks
ICPPW, Proceedings of the 2002 International Conference on Parallel Processing Workshops, p. 5, 2002, ISBN:0-7695-1680-7, IEEE Computer Society Washington, DC, USA

50. R. Lippmann, K. Ingols
An Annotated Review of Past Papers on Attack Graphs
ESC-TR-2005-054, Project Report IA-1, 31 March 2005
MIT Lincoln Laboratory, 244 Wood Street, Lexington, Massachusetts
http://www.ll.mit.edu/IST/pubs/0502_Lippmann.pdf

51. R. Lippmann, K. Ingols, C. Scott, K. Piwowarski, K. Kratkiewicz, M. Artz, R. Cunningham
Validating and Restoring Defense in Depth Using Attack Graphs
Military Communications Conference, 2006. MILCOM 2006, October 2006, p. 1-10, Washington, DC, ISBN: 1-4244-0618-8, INSPEC Accession Number: 9418886
MIT Lincoln Laboratory, 244 Wood Street, Lexington, Massachusetts

52. C. Phillips, L. Swiler
A Graph-Based System for Network-Vulnerability Analysis
Sandia National Laboratories, MS 0746, Albuquerque, NM
Proceedings of the 1998 workshop on New security paradigms, Charlottesville, Virginia, United States, p. 71-79, 1998, ISBN:1-58113-168-2