Capitolo 9

Appendice A: Widsard language, linguaggio ad espressioni regolari per eventi

Introduzione

Widsard è un sistema di intrusion detection in grado di monitorare un processo critico e di riconoscere un suo eventuale comportamento anomalo. Il programma si basa su un meccanismo di intercettazione delle chiamate di sistema, fornito per scopi di debug dal kernel 2.2.x di Linux, e grazie ad esso può cercare una corrispondenza tra la sequenza di chiamate di sistema eseguita dal processo monitorato e un insieme di pattern presenti nelle specifiche relative a quel processo.

Questa appendice descrive il linguaggio adottato per creare le specifiche che descrivono il comportamento "normale o anomalo" del processo da monitorare. Si tratta di un linguaggio per espressioni regolari minimale, cui è aggiunta la possibilità di utilizzare variabili ed eseguire istruzioni generiche.

"Widsard language" prende spunto in larga parte dal linguaggio descritto in [US99-02], ma utilizza Linux come sistema operativo di riferimento.

Elementi lessicali

"Widsard language" permette di utilizzare molte delle caratteristiche di un linguaggio di programmazione generale, anche se è destinato ad un diverso utilizzo. In determinati contesti, è infatti possibile far uso di espressioni relazionali, assegnamenti a variabili, chiamate a funzioni. In questo paragrafo sono descritti gli elementi lessicali che il linguaggio ha in comune con gli altri linguaggi di programmazione.

Commenti

I commenti sono stringhe arbitrarie precedute dal simbolo cancelletto '#':

```
#Questo è un commento
```

Parole chiave

Le parole chiave del linguaggio sono:

```
intro exit begin int char string
```

Inoltre sono da considerarsi come parole chiave anche i nomi delle chiamate al sistema operativo, seguite dalla parentesi tonda aperta e la parentesi tonda chiusa, ad esempio:

```
open() read() getsockopt() ...
```

Una lista completa delle chiamate di sistema relative al sistema operativo Linux si trova nell'allegato 1.

Identificatori

Un identificatore è una sequenza di caratteri di cui il primo è una lettera minuscola oppure un *underscore*, mentre i successivi possono essere cifre, lettere minuscole, maiuscole oppure *underscore*. Un identificatore, a seconda del contesto, può essere interpretato come un tipo di dato oppure come una variabile posizionale.

Alcuni esempi di identificatori validi sono:

```
flag123
clientIP
_system
```

Costanti

Il linguaggio descritto manipola alcuni tipi di valori fondamentali, e per ciascuno di essi è possibile esprimere una costante. La convenzione utilizzata è la stessa del linguaggio C:

- un valore di tipo carattere è esprimibile ponendo il carattere tra apici singoli
- un valore di tipo stringa è esprimibile ponendo una sequenza di caratteri tra apici doppi

tipo di dato	esempio di costante
carattere	'c'
stringa	"una stringa"

Le costanti intere si possono esprimere in modo diverso a seconda della base considerata:

- in base decimale le costanti sono esprimibili tramite una normale sequenza di cifre
- in base ottale occorre far precedere la sequenza di cifre da 0
- in base esadecimale è necessario far precedere la sequenza di cifre da 0x oppure 0X

base	esempio
decimale	379
ottale	0240
esadecimale	0x68FA

Insiemi omogenei

É possibile rappresentare degli insiemi di elementi dello stesso tipo. La sintassi è la seguente:

```
{ value_one, value_two, ..., value_n }
```

I valori contenuti possono essere interi, caratteri o stringhe. Alcuni esempi di insiemi omogenei sono:

```
{1, 2, 3, 890, 900}
{'a', 'c', '3', '0'}
{"cgi", "cgi-bin", "cgi"}
```

è possibile anche specificare un insieme omogeneo vuoto:

```
{ }
```

Variabili

Nel linguaggio possiamo distinguere

1. variabili globali
2. variabili locali
3. variabili posizionali

Le variabili dei primi due tipi sono indicate attraverso identificatori particolari, perché preceduti dal carattere *'$'*. Ciascuna di queste variabili ha un tipo che deve essere dichiarato la prima volta che la variabile compare nelle specifiche. I tipi supportati sono quelli fondamentali, descritti nel paragrafo 9.

Esistono due tipi di variabili, le variabili scalari e le variabili insieme. Queste ultime si indicano con la sintassi

```
$var_name{}
```

e contengono insiemi omogenei di valori.

- Esempi di variabili scalari sono:

  ```
  $cgipath    $loggedUser   $found
  ```

- Esempi di variabili insieme sono:

  ```
  $fds{}   $shells{}   $invalid_dirs{}
  ```

Le variabili posizionali sono indicate attraverso normali identificatori ed il loro utilizzo è trattato al paragrafo 9. Alcuni esempi di variabili posizionali sono:

```
fd    return_value   flag    file    buffer
```

Tipi di dati

"Widsard language" definisce alcuni tipi di dati fondamentali, comuni alla maggior parte dei linguaggi di programmazione, ed in particolare al C. Per esprimere l'insieme di definizione dei tipi di dato utilizziamo le costanti definite nel file header limits.h della libreria standard del C:

tipo di dato	parola riservata	insieme di definizione
intero	int	[INT_MIN-INT_MAX]
carattere	char	[CHAR_MIN-CHAR_MAX]
stringa	string	[CHAR_MIN-CHAR_MAX]*

Oltre ai tipi di dato fondamentali, sono definiti i tipi degli argomenti delle chiamate di sistema del sistema operativo. Per i sistemi Linux/i386 vi è un elenco nell'allegato 2.

L'uso dei tipi permette al programma di effettuare alcuni controlli sui valori e sulle variabili presenti nelle specifiche relative al processo da monitorare, cosí da evitare irrimediabili interruzioni a causa di errori dell'utente, durante il tracciamento.

Operatori di confronto

In "Widsard language" sono definiti i seguenti operatori di confronto:

```
==    !=    >=    <=    <    >
```

La sintassi è la stessa del C. La semantica per i valori scalari interi è quella usuale. Per i caratteri vengono considerati i corrispondenti valori ASCII.

Sulle stringhe attualmente sono definiti i soli operatori di uguaglianza:

```
==    !=
```

Gli operatori di uguaglianza assumono una diverso significato quando uno degli operandi è un insieme omogeneo o una variabile insieme. Gli altri operatori non sono definiti in questo caso. Sia a uno scalare e I un insieme omogeneo, allora si ha:

espressione	semantica
$a == I$	$a \in I$
$a != I$	$a \notin I$

- Di seguito sono mostrati alcuni esempi di espressioni relazionali:

  ```
  $cgipath == "/home/httpd/cgi-bin"
  ```

  dove *$cgipath* è una variabile di tipo stringa.

  ```
  fd != {0, 1, 2}
  ```

dove *fd* è una variabile posizionale. Questa espressione assume valore diverso da zero - *true* - se il valore associato a *fd* non è contenuto nell'insieme *{0, 1, 2}*.

```
fd == $fds{}
```

dove *fd* è una variabile posizionale e *$fds{}* una variabile di tipo insieme. L'espressione assume valore diverso da zero, se il valore associato a *fd* è contenuto nell'insieme omogeneo referenziato da *$fds{}*.

Assegnamenti

La semantica di un assegnamento dipende dal tipo di variabile che compare a sinistra:

- se la variabile è di tipo scalare, l'assegnamento ha l'usuale significato
- sia $v{} una variabile di tipo insieme, a uno scalare e I un insieme omogeneo, tutti dichiarati con lo stesso tipo di dati, allora

assegnamento	semantica
$v{} = a	all'insieme cui $v{} si riferisce e' aggiunto l'elemento a
$v{} = I	a $v{} e' assegnato l'insieme I

Esempi:

- assegnamenti a variabili scalari:

```
$name = "user"
$flag = 4
```

- assegnamenti a variabili insieme:

```
$invalid_dirs{} = {"/etc", "/sbin", "/bin", "usr"}
$fds{} = {0, 1, 2}
```

- inserimento di valori scalari in insiemi referenziati da variabili insieme:

```
$fds{} = 2
$stringset{} = "a string"
```

- inserimento del valore associato alla variabile posizionale *fd* nell'insieme referenziato dalla variabile *$fds{}*:

```
$fds{} = fd
```

Funzioni

"Widsard language" permette di specificare chiamate a funzioni di libreria o a funzioni definite dall'utente. La chiamata a funzione ha la stessa sintassi del C:

```
function_name(arg1, arg2, ..., argN)
```

Attualmente il massimo numero di argomenti che può avere una funzione è 10. Un argomento può essere:

- una costante
- una variabile
- un identificatore (variabile posizionale in questo contesto, vedi il paragrafo 9)
- una funzione

I tipi degli argomenti e del valore restituito dalle funzioni sono quelli descritti nel paragrafo 9.

Le funzioni sono valutate ogni qual volta è necessario durante il monitoraggio del processo.

Widsard fornisce un insieme di funzioni predefinite, per il quale si rimanda all'appendice D. Di seguito sono mostrati alcuni esempi di chiamate a funzioni predefinite:

```
find(bufferpwd,$stringapwd,"")
log(0,$ipaddr)
kill_actual()
find_begin(pWD,"GET :/cgi-bin/:/etc/passwd",":")
log(find(bufferpwd,$stringapwd,""), $ipaddr)
```

Il file delle specifiche

Per monitorare un processo, Widsard richiede un file che contenga

- indicazioni su quali risorse utilizzare
- le regole da seguire

Il formato di questo file deve essere:

```
dichiarazioni
%%
regole
```

Le dichiarazioni

La sezione delle dichiarazioni del file delle specifiche può contenere:

- dichiarazioni di variabili globali e relativa inizializzazione
- i prototipi delle funzioni non predefinite utilizzate all'interno del file
- dichiarazioni di macro

Variabili globali

Le variabili che sono definite nella sezione delle dichiarazioni sono visibili in tutte le regole della successiva sezione del file delle specifiche. Una dichiarazione di variabile sintatticamente appare cosí:

```
type var = value
```

dove

- "type" è uno dei tipi di base (intero, carattere o stringa)
- "var" è una variabile scalare o una variabile insieme
- "value" è uno scalare se "var" è una variabile scalare, un insieme omogeneo se "var" è una variabile insieme. Il tipo di value deve concordare con "type".

Alcuni esempi di dichiarazioni di variabili globali sono:

```
int $execve_max_arg = 100
int $execve_max_path = 100
string $metachar = ";:<:>:*:|:!:#:(:):[:]:{:}"
string $hex_metachar = "%3B:%3C:%3E:%2A:%7C:%3F:%26:%24"
int $fds{} = { }
string $shells{} = {"/bin/sh", "/bin/bash",
                    "/bin/ksh", "/bin/zsh",
                    "/bin/csh", "/bin/tcsh" }
string $invalid_dirs{} = {"/etc", "/sbin",
                          "/usr", "/bin"  }
```

Prototipi di funzioni

Se nel file delle specifiche si intendono utilizzare funzioni diverse da quelle predefinite, è necessario che l'utente ne specifichi il prototipo e la libreria condivisa dove trovarne la definizione. La sintassi è la seguente:

```
include "shared_lib" type function_name(types)
```

La "shared_lib" è una libreria condivisa, generalmente un file con estensione ".so". Widsard prima di iniziare il monitoraggio caricherà la libreria e acquisirà l'*handler* alla funzione indicata nel prototipo. Non tutte le funzioni sono supportate da Widsard: vi sono infatti delle limitazioni sui tipi degli argomenti. L'elenco dei tipi di argomenti supportati per le funzioni è indicato nell'allegato 2.

Un esempio di prototipo di funzione non predefinita è il seguente:

```
include "/lib/libc.so.6" char_ptr realpath(char_ptr, char_ptr)
```

Macro

"Widsard" fornisce la possibilità di indicare con un nome arbitrario una espressione regolare per eventi. Le espressioni regolari per eventi saranno descritte nei prossimi paragrafi. La sintassi per la definizione di una macro è la seguente:

```
macro_name() := { regular_expression_for_events }
```

Un esempio di macro è:

```
openfd() := { open() || dup() || dup2() }
```

Le regole

La sezione delle regole del file delle specifiche deve essere preceduta dal simbolo '%%'. Ciascuna regola consiste in un'espressione regolare, per la quale il programma Widsard cerca una corrispondenza nella sequenza di eventi prodotta dal processo monitorato.

L'evento semplice

Il linguaggio ad espressioni regolari adottato ha come elemento fondamentale la specifica di un evento semplice. Tale evento corrisponde all'intercettazione di una chiamata di sistema eseguita dal processo monitorato, cui l'utente può far seguire delle istruzioni prima che il controllo ritorni ad esso. Pertanto

```
event <statements> .
```

è la forma della regola piú semplice che può apparire nelle specifiche relative ad un processo. Un evento è indicato attraverso il nome di una chiamata di sistema ed eventualmente delle condizioni sui parametri e sul valore di ritorno:

```
syscall()
syscall(params) | (conditions)
syscall(params) = ret_val | (conditions)                (1)
```

Se le condizioni non sono verificate, l'evento intercettato non viene preso in considerazione, cioè non si stabilisce la corrispondenza con il modello indicato dalla regola - *pattern*. Se invece l'evento corrisponde al pattern sono eseguite le istruzioni indicate dopo.

Le chiamate di sistema a cui si può far riferimento sono tutte quelle supportate dal sistema operativo utilizzato. Il nome di ciascuna di essa seguito dalle parentesi '(' ')' costituisce una parola riservata. L'elenco dei nomi delle chiamate di sistema che è possibile specificare sui sistemi Linux/i386 è nell'allegato 1.

Il contesto

L'intercettazione di una chiamata di sistema avviene in due momenti distinti:

- all'invocazione della chiamata di sistema, subito dopo che il controllo si è trasferito al kernel,

- all'uscita dalla chiamata di sistema, prima che il controllo torni al processo monitorato.

É importante distinguere queste due fasi, in quanto i parametri su cui è possibile eseguire dei controlli in uscita ad una chiamata di sistema, generalmente, non sono significativi in entrata, oppure hanno un significato diverso. Se l'evento e le condizioni su di esso sono specificate come in (1), Widsard interpreta che le condizioni debbano essere valutate in entrata alla chiamata di sistema. In alternativa si può indicare esplicitamente il contesto in cui valutare le condizioni:

```
syscall(params) = ret_val | (intro: conditions)
syscall(params) = ret_val | (exit: conditions)
syscall(params) = ret_val | (intro: conditions  exit: conditions)
```

Altrettanta attenzione richiede il contesto in cui eseguire le istruzioni in seguito al riconoscimento di un evento. In tal caso, il contesto deve essere sempre espresso:

```
<intro: statements>
<exit: statements>
<intro: statements  exit: statements>
```

La regola piú complessa, costituita da un singolo evento, che è possibile scrivere, è questa:

```
syscall(params) = ret_val | (intro: conditions  exit: conditions)
                            <intro: statements  exit: statements>
```

In questo caso, Widsard, al verificarsi della chiamata di sistema "syscall", valuta le condizioni in entrata e, se sono verificate, esegue le istruzioni relative. In uscita alla stessa chiamata, valuta le condizioni in uscita e, se sono verificate, esegue le istruzioni corrispondenti. Se uno dei controlli effettuati ha esito negativo, non si stabilisce la corrispondenza tra l'evento e il pattern.

Variabili posizionali

Un pattern costituito da

```
syscall()
```

corrisponde alla chiamata di sistema indicata, ma in tal modo non è possibile specificare alcun controllo o alcuna elaborazione degli argomenti o del valore di ritorno. Per far riferimento agli argomenti e al valore di ritorno, si possono utilizzare degli identificatori di nome arbitrario, qui indicati come variabili posizionali, che Widsard interpreterà come *handler* ai valori corrispondenti. Questo esempio:

```
open(filename, flag) = fd
```

utilizza tre variabili posizionali:

- "*filename*" è un riferimento al valore del primo parametro della open
- "*flag*" è un riferimento al valore del secondo parametro

- "*fd*" è un riferimento al valore di ritorno

Notare che i parametri della chiamata di sistema devono essere separati da virgole e che la variabile corrispondente al valore di ritorno di una chiamata di sistema deve essere esterna alle parentesi '(' ')' e preceduta dal simbolo '='.

Se non è necessario specificare delle variabili posizionali per alcuni parametri della chiamata di sistema, esse possono essere sostituite dal carattere *underscore* '_'. Ad esempio:

```
open(_, flag) = filedesc
```

Infine è anche possibile non specificare alcuna variabile posizionale per gli argomenti, ma solo quella per il valore di ritorno:

```
open() = fd
```

Le condizioni

Per descrivere piú in dettaglio un evento, si possono specificare una o piú condizioni:

```
(intro: condition1, condition2, ..., conditionN)
(exit: condition1, condition2, ..., conditionM)
(intro: condition1, ..., conditionN  exit: condition1, ..., conditionM)
```

L'insieme di condizioni relative ad una chiamata di sistema deve essere racchiuso tra parentesi tonde. Le condizioni sono separate da virgole. Se l'evento intercettato non verifica anche una sola di esse, non si stabilisce una corrispondenza con il pattern.

Una condizione può essere:

- un'espressione relazionale
- una funzione che restituisce un valore intero: se la valutazione di tale funzione produce un valore diverso da zero, si considera verificata la condizione.

Le espressioni relazionali sono descritte nel paragrafo sugli operatori di confronto 9.

Le etichette segnaposto, le variabili e le costanti possono essere utilizzate come operandi nelle espressioni relazionali, oppure come parametri delle funzioni.

Un altro modo per esprimere una condizione sugli argomenti di una chiamata di sistema è sostituire la variabile posizionale con una costante o con una variabile d'altro tipo. Ad esempio:

```
write($filedesc, _, _)
```

è equivalente a

```
write(fd, _, _) | (intro: fd == $filedesc)
```

Utilizzando questa scorciatoia, Widsard tenta di determinare automaticamente il contesto in cui effettuare il controllo. Se non ci riesce, costringerà l'utente a scrivere la condizione in modo esplicito.

Le istruzioni

Dopo aver stabilito una corrispondenza tra l'evento intercettato e quello specificato in una regola, Widsard tenta di eseguire le istruzioni indicate tra parentesi angolate:

```
<intro: statement1, statement2, ..., statementN>
<exit: statement1, statement2, ..., statementM>
<intro: statement1, ..., statementN  exit: statement1, ..., statementM>
```

Nel caso delle istruzioni, occorre sempre specificare il contesto in cui eseguirle: in entrata o in uscita alla chiamata di sistema.

Una istruzione può essere:

- un assegnamento a variabile
- una funzione

Se la funzione restituisce un valore, esso non viene considerato.

Variabili locali ad una regola

Le istruzioni da eseguire dopo un evento possono anche far riferimento a variabili non dichiarate precedentemente, inizializzandole con la stessa sintassi utilizzata per le variabili globali:

```
type var = value
```

La visibilità di queste variabili è limitata alla regola in cui sono dichiarate.

Esempi di pattern semplici

- Esempio di specifica di una chiamata di sistema con una condizione in entrata. La condizione è costituita da una espressione relazionale, di cui il primo operando è una variabile posizionale:

  ```
  write(fd, _, _) | (fd > 2)
  ```

  Questa specifica è equivalente a

  ```
  write(fd, _, _) | (intro: fd > 2)
  ```

- Esempio di specifica di una chiamata di sistema con una condizione in uscita. La condizione è costituita da una chiamata a funzione con variabili posizionali e variabili di stato come parametri:

  ```
  read(_,buffer, _) | (exit: find_begin(buffer, $patt, $sep))
  ```

- Esempio di chiamata di sistema con condizioni specificate sia in entrata, sia in uscita:

  ```
  write(fd, _, _) = ret | (intro: fd > 2, exit: ret > 0)
  ```

- Esempio di chiamata di sistema con una istruzione costituita da un assegnamento ad una variabile locale:

```
fork() = ret <exit: int $childpid = ret>
```

- Esempio di chiamata di sistema con una istruzione specificata in entrata. L'istruzione è una chiamata a funzione predefinita:

```
execve("/bin/sh") <intro: kill_actual()>
```

- Esempio di specifica con due condizioni e due istruzioni in entrata:

```
open(file, flag) | (file == "/etc/passwd", flag == O_WRITE)
                   <intro: log(0,"ATTACCO"), freeze()>
```

- Esempio di chiamata di sistema con una istruzione in entrata ed una condizione in uscita:

```
chdir(path) | (exit: path == "..") <intro: $changepwd = 1>
```

  In questo esempio l'istruzione viene eseguita in entrata, prima che sia valutata la condizione.

Eventi speciali

Due eventi semplici particolari sono:

```
begin()
@
```

Il primo evento è fittizio e sta ad indicare l'inizio del processo. Esso può essere utilizzato all'inizio di una regola, e non altrove, per indicare a Widsard di cercare una corrispondenza con quel pattern solo all'inizio della sequenza di eventi prodotta dal processo monitorato, non a partire da un evento qualsiasi di tale sequenza.

Il secondo evento corrisponde ad una qualunque chiamata di sistema valida.

Sequenze di eventi

Ciascuna regola può essere costituita da un evento semplice o da una successione di eventi. Widsard cercherà nella storia di eventi intercettati una corrispondenza con la sequenza specificata, eseguendo ad ogni passo le istruzioni indicate di seguito ad ogni elemento della sequenza. L'operatore di sequenzializzazione è indicato dal simbolo ';'. Il segnale di fine sequenza è il simbolo '.'. Dunque la forma base di una regola è una tupla di eventi semplici:

```
event1 ; event2 ; ... ; eventN .
```

Piú genericamente, l'elemento di una sequenza può essere:

- un evento semplice
- una combinazione di eventi

Combinazioni di eventi

Una combinazione di eventi è una espressione regolare costituita da eventi composti tra loro per mezzo di alcuni operatori. Questi sono definiti nella seguente tabella:

operazione	nome	semantica	associativita'
event	-	una occorrenza di *event*	-
!*event*	negazione	occorrenza di un evento diverso da *event*	destra
*event**	ripetizione	n occorrenze di *event* con $n \geq 0$	sinistra
event1 || *event2*	alternanza	occorrenza di *event1* oppure di *event2*	sinistra
event1 ; *event2*	sequenzializzazione	occorrenza di *event1* seguito da *event2*	sinistra
{ *event* }	parentesi	occorrenza di *event*	-

Gli operatori sono elencati nella tabella in ordine di precedenza decrescente.

Non tutte le combinazioni sono possibili. Nei prossimi paragrafi sono elencate le caratteristiche di ciascun operatore.

Negazione

É possibile negare un evento semplice, oppure un insieme di eventi alternativi tra loro. Le limitazioni sull'utilizzo di questo operatore sono:

- non è possibile negare una sequenza di eventi, una ripetizione di eventi o una combinazione di essi
- non è permessa la doppia negazione di un evento
- è considerato un errore negare l'evento qualunque "@" (equivarrebbe all'evento vuoto).

É errato specificare delle istruzioni "in" un evento negato, ad esempio

```
! open(filename,_) | (filename == "database") <exit: print_syscall()>
```

durante il parsing produrrà un errore, poiché le istruzioni indicate tra parentesi angolate hanno la precedenza sull'operatore di negazione. La versione corretta è

```
{ ! open(filename,_) | (filename == "database") } <exit: print_syscall()>
```

Tale pattern corrisponde ad una chiamata di sistema che non sia una "open" con il primo parametro uguale alla stringa "database": se la verifica ha esito positivo, allora viene eseguita l'istruzione e sarà stampata la chiamata di sistema realmente intercettata. Altrimenti, si è verificata esattamente una "open" sul file "database" per cui non si è stabilita la corrispondenza con il pattern e l'istruzione di stampa non è eseguita.

É comunque da notare che, nelle istruzioni che seguono una negazione di eventi, non è possibile utilizzare le variabili posizionali.

Ripetizione

É possibile indicare zero, una o piú occorrenze di una qualsiasi combinazione di eventi utilizzando l'operatore "*". Per specificare almeno una occorrenza di un evento occorre indicarlo esplicitamente:

```
event ; event *
```

Questo operatore non può comparire all'interno di una combinazione di eventi negata.

Alternanza

Questo è un operatore binario, i cui operandi possono essere eventi semplici o combinazioni di eventi.

L'utilizzo combinato della negazione con l'operatore di *alternanza* permette di esprimere facilmente insiemi arbitrari di chiamate di sistema, che sono considerate contestualmente equivalenti. Per esempio:

```
open() || dup() || dup2()
```

mostra l'insieme delle chiamate di sistema che possono allocare descrittori di file, mentre

```
! { open() || dup() || dup2() }
```

ne è il complementare. Attenzione che questi a sua volta non è equivalente a

```
!open() || !dup() || !dup2()
```

che corrisponde invece all'intero insieme delle chiamate di sistema.

Sequenzializzazione

Possono essere messi in sequenza eventi semplici o combinazioni di eventi. Questo operatore però non può apparire all'interno di una combinazione di eventi negata.

Da notare che il *sequencing* ha priorità minore rispetto all'*alternanza* per cui

```
event1 ; event2 || event3 ; event4
```

va interpretato nel modo seguente:

```
event1 ; { event2 || event3 } ; event4
```

Parentesi

Le parentesi rendono esplicito l'ordine con cui combinare gli eventi, ma in alcuni dei casi che fanno uso della negazione sono indispensabili. Ne sono già stati dati degli esempi in questo stesso paragrafo.

Esempi di combinazioni di eventi

- Esempio di *negazione* di un evento semplice :

```
! close(fd) | (fd == $fds{})
```

- Esempio di *alternanza* tra eventi semplici:

```
open(file, _) = fd | (file == $files{}) <exit: $fds{} = fd>
|| dup(fd1) = fd2 | (fd1 == $fds{}) <exit: $fds{} = fd>
|| dup2(fd1, fd2) | (fd1 == $fds{}) <intro: $fds{} = fd>
```

- Esempio di *sequenzializzazione* di eventi semplici:

```
fork() = ret | (exit: ret == 0) ; execve()
```

- Esempio di *ripetizione* di un evento semplice:

```
setreuid(ruid, euid) <exit: $userid = euid> *
```

- Esempio di regola contenente l'evento di inizio sequenza:

```
begin()
; {! getpeername()} *
; open(file, _) | (file == "/etc/passwd")
                  <exit: kill_actual()> .
```

- Esempio di regola contenente l'evento generico:

```
execve (c,_,_) | (find(c,$pathcgi,"") )
;@*
; open (pathf,_) | (find(pathf,"..",":"))
                   <intro: log("ATTACCO") > .
```