Sapienza Università di Roma

Facoltà di Ingegneria

Corso di Laurea Specialistica in

Ingegneria Informatica

Metodi Formali nell'Ingegneria del Software

Verifica formale del TCP e studio di possibili attacchi usando NuSMV

Prof.
Toni Mancini

Studente
Cristiano Sticca

Anno Accademico 2006/2007

Abstract

Con questo lavoro si vuole costruire un modello della macchina a stati del *transmission control protocol* (TCP) [1] e verificare tale modello usando il model checker[1] NuSMV [2]. In seguito si vuole verificare la possibilità di attacchi di tipo denial-of-service[2] (DoS), come quello meglio conosciuto con il nome TCP SYN flooding [7], e poi infine cercare di trovare un nuovo attacco a tale protocollo.

[1] il model checking è un metodo per verificare algoritmicamente i sistemi formali. Viene realizzato mediante la verifica del modello, spesso derivato dal modello hardware o software, soddisfacendo una specifica formale. La specifica è spesso scritta come formule logiche temporali.

[2] in questo tipo di attacco si cerca di portare il funzionamento di un sistema informatico che fornisce un servizio, ad esempio un sito web, al limite delle prestazioni, lavorando su uno dei parametri d'ingresso, fino a renderlo non più in grado di erogare il servizio.

Indice

Capitolo 1

Introduzione

In questa sezione viene presentato il protocollo TCP e viene fatta una breve introduzione sul *model checker* NuSMV [2].
Infine introduciamo i concetti base per un attacco di tipo *denial-of-service* [4].

1.1 Una breve introduzione al TCP

TCP è un importantissimo e ben conosciuto protocollo di rete. Benché TCP è sempre menzionato come parte dello strato di rete Internet TCP/IP [1] , esso è indipendente ed è un protocollo che può essere adattato in qualsiasi altro sistema di *delivery*.

TCP è un protocollo di alto livello molto complesso. Esso specifica il formato dei dati e gli *acknowledgments*[1] che due computer si scambiano per mantenere un trasferimento affidabile, come anche le procedure che i terminali usano per assicurare che i dati arrivino correttamente. Inoltre il trasferimento affidabile è assicurato anche dal fatto che i dati che i computer trasmettono non sono duplicati né ci sono perdite di questi ultimi. TCP assicura l'affidabilità usando dei *positive acknowledgments* con ritrasmissione. Questa tecnica richiede che un client per comunicare con una sorgente debba inviare indietro non appena ricevuto dei dati un messaggio di *acknowledge* (ACK) . Colui che invia tiene traccia di tutti i pacchetti inviati e aspetta un messaggio di ACK prima di inviare il prossimo pacchetto. Il mittente quindi fa partire un timer quando invia un pacchetto e ritrasmette quest ultimo qualora il timer scatti prima che arrivi un messaggio di ACK.

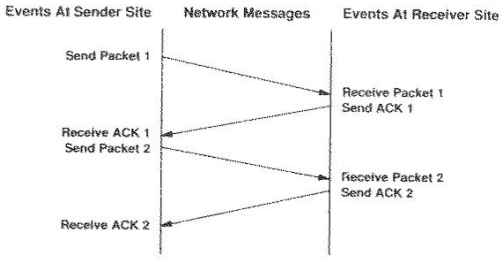

Fig. 1.1: Semplice funzionamento di uno scambio di messaggio con *positive acknowledgment* [1]

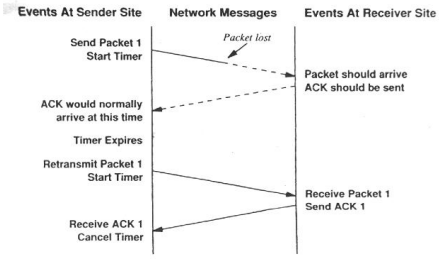

Fig. 1.2: Funzionamento di uno scambio di messaggi con *timeout* e *ritrasmissione* [1]

[1] tipo di segnale trasmesso dal ricevente al mittente per segnalare la corretta ricezione di un pacchetto dati.

Una cosa fondamentale nell'illustrare il TCP è che ogni pacchetto di dati inviato porta con sé un numero di sequenza (*sequence number*). Il meccanismo di *acknowledgement* impiegato è cumulativo, nel senso che un acknowledgment del sequence number X indica che tutti i pacchetti il cui numero di sequenza è minore di X sono stati ricevuti. Questo meccanismo permette molto facilmente di controllare se ci sono dati duplicati. Per stabilire una connessione, TCP usa una "stretta di mano a tre vie" (*three-way handshake*). Ciò viene mostrato nella figura qui di seguito.

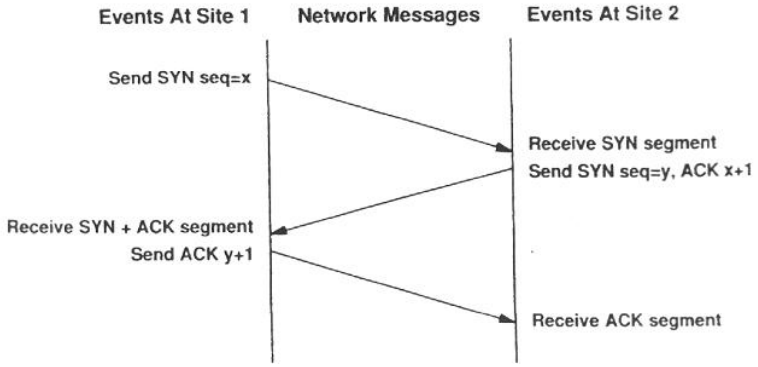

Fig. 1.3: Funzionamento del *3-way handshake* [1]

1.2 TCP diagramma stati e transizioni

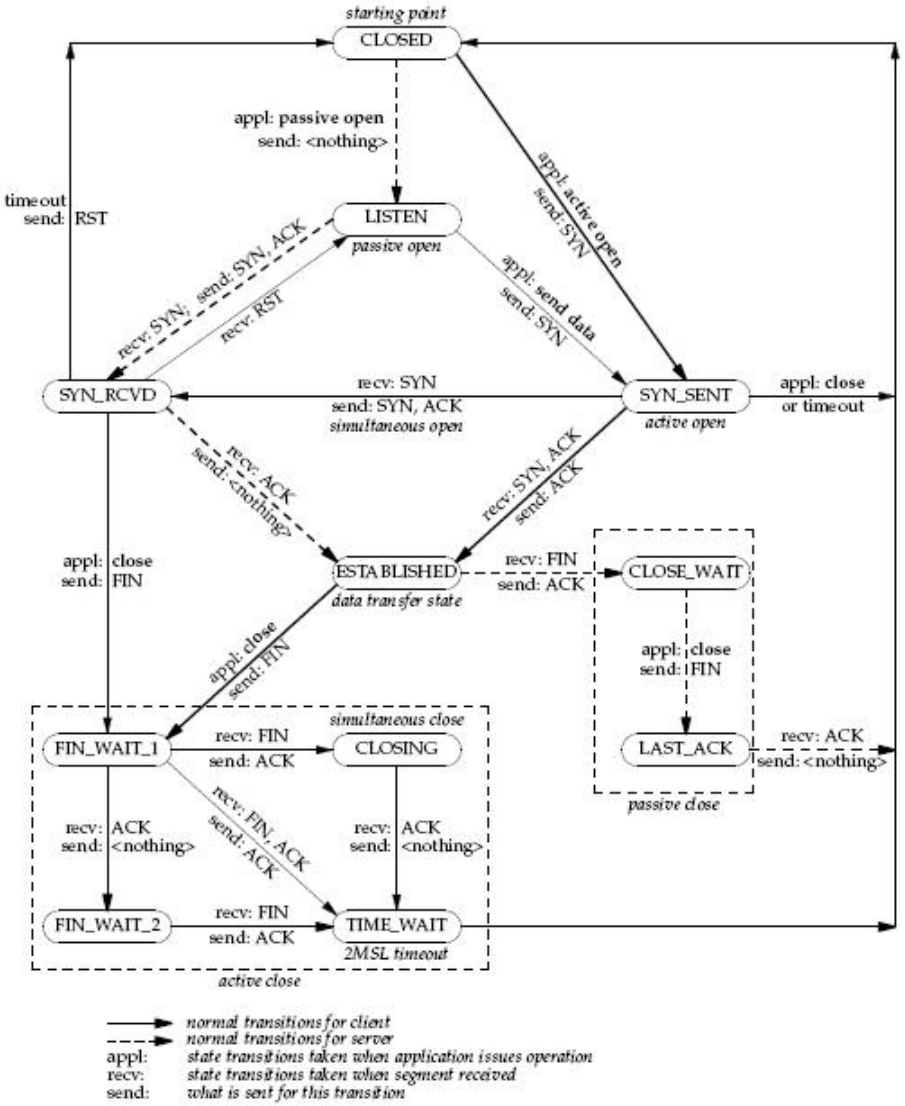

Fig. 1.4: Diagramma degli stati e delle transizioni del TCP [1]

1.2.1 Commenti al diagramma

CLOSED

- stato iniziale (protocollo non attivo)
- per uscire da questo stato si deve effettuare un operazione di *open* (passiva o attiva)
  - l'open passiva non manda nulla e passa allo stato LISTEN
  - l'open attiva spedisce un messaggio SYN e passa allo stato SYN-SENT

LISTEN

- in questo stato il protocollo è attivo ed è in ascolto su una porta
- quando riceve un SYN risponde con un SYN+ACK e passa allo stato SYN-RECEIVED
- se l'applicazione chiede di inviare dati manda un SYN e passa allo stato SYN-SENT

SYN-SENT

- stato in cui si è mandato un SYN e si attende l'ACK corrispondente
  - raggiunto da CLOSED con una open attiva o da LISTEN dopo un'operazione di SEND
- attende la risposta SYN per un certo tempo
  - se riceve un SYN con ACK passa allo stato ESTABLISHED e manda a sua volta un ACK
  - se riceve un SYN senza ACK (open simultanea) manda un SYN+ACK e passa allo stato SYN-RECEIVED
  - se non riceve risposta effettua una *close* o una *reset*

SYN-RECEIVED

- stato in cui si è ricevuto un SYN
  - se lo rifiuta, ritorna allo stato LISTEN con *reset*
  - se accetta, passa allo stato ESTABLISHED e manda l'ACK

ESTABLISHED

- stato in cui è stata stabilita le connessione ed è possibile iniziare il trasferimento dei dati
  - è stata completata la *3-way handshake*
- se l'applicazione decide di chiudere la connessione manda un messaggio di FIN e passa allo stato FIN-WAIT-1 (close attiva)
- se riceve un messaggio FIN risponde con un ACK e passa allo stato CLOSE-WAIT (close passiva)

CLOSE-WAIT

- stato in cui si è ricevuto un messaggio di FIN e si attende che l'applicazione chiuda la connessione
- quando l'applicazione decide di chiudere la connessione manda un messaggio di FIN e passa allo stato LAST-ACK

LAST-ACK

- stato in cui si è ricevuto il FIN dall'altro *end-point* e si è risposto con un FIN
  - il protocollo attende l'ACK al suo FIN
- quando riceve l'ACK risponde con l'ultimo ACK e chiude la connessione

FIN-WAIT-1

- stato in cui si è inviato un messaggio FIN e si attende che l'altro end-point chiuda la connessione
- se riceve un FIN+ACK manda l'ACK e passa allo stato TIME-WAIT
- se riceve solo un FIN (close simultanea) manda l'ACK e passa allo stato CLOSING
- se riceve un ACK passa allo stato FIN-WAIT-2

CLOSING

- stato in cui entrambi gli end-point hanno mandato un FIN contemporaneamente
- manda l'ACK e passa allo stato TIME-WAIT

FIN-WAIT-2

- stato in cui si è inviato un messaggio FIN per il quale è stato ricevuto l'ACK e si attende il FIN dell'altro end-point (half-close)
- quando riceve un FIN manda l'ACK e passa allo stato TIME-WAIT

TIME-WAIT

- attende un tempo pari a 2*MSL (Maximum Segment Life[2]) prima di chiudere la connessione per attendere eventuali richieste di ritrasmissione dell'ultimo ACK
  - la durata dipende dall'implementazione
- per tutto questo intervallo di tempo la porta dell'end-point non è utilizzabile

Una connessione TCP progredisce da uno stato all'altro in risposta ad eventi. Gli eventi sono: ***usercalls***, per esempio **OPEN, SEND, RECEIVE, CLOSE, ABORT, and STATUS**; ***i segmenti in arrivo*** in particolare quelli che contengono i **SYN, ACK, RST, FIN** flags; e ***timeouts***.

[2]il massimo limite di tempo che un segmento può rimanere in internet

1.3 NuSMV

NuSMV [2] è un model checker simbolico sviluppato dall'Università di Carnegie Mellon (CMU) e dall' Istituto per la Ricerca Scientifica e Tecnologica (IRST). Scopo principale del progetto NuSMV è stato quello di sviluppare un'architettura aperta per il model checking, che può essere usato con affidabilità per testare e verificare progetti industriali, come cuore di strumenti per la verifica, e per essere applicato ad altre aree di ricerca.

Le principali caratteristiche di NuSMV sono le seguenti:

- Funzionalità. NuSMV permette la rappresentazione di sistemi a stati finiti sia sincroni che asincroni, e l'analisi delle specifiche espresse in Computation Tree Logic (CTL) [3] e Linear Temporal Logic (LTL) [6].
- Achitettura. I differenti componenti e le funzionalità di NuSMV sono stati isolati all'interno di moduli. Sono state fornite interfacce per la comunicazione tra moduli e tutto ciò ha come effetto la riduzione degli sforzi per poter estendere e modificare NuSMV.
- Qualità dell'implementazione. NuSMV è scritto in ANSI C ed è commentato in maniera molto efficiente in modo da poter essere appreso facilmente da chiunque.

1.4 Attacchi per il TCP

Sebbene TCP è stato usato come protocollo affidabile per molto tempo esso presenta molti problemi di sicurezza come ad esempio l'IP *spoofing*[3] e gli attacchi Denial of Service. La maggior parte di questi attacchi è basata su un difetto particolare TCP. *"If available, the easiest mechanism to abuse is IP source routing. Assume that the target host uses the reverse of the source route provided in a TCP open request for return traffic...The attacker can then pick any IP source address desired, including that of a trusted machine on the target's local network".*

Nel progetto assumiamo che colui che compie un attacco può facilmente falsificare l'indirizzo IP.

1.5 Struttura della tesina

Capitolo 2 In questa sezione descriviamo il modello della macchina a stati del TCP che si vuole costruire e le astrazioni che assumiamo.

Capitolo 3 In questo capitolo verifichiamo una serie di proprietà per provare la consistenza del nostro modello.

Capitolo 4 Qui usiamo il modello costruito per trovare possibili attacchi del TCP.

Capitolo 5 Si conclude la relazione mostrando i problemi affrontati durante il lavoro e il resoconto di tutto ciò che è stato fatto.

Capitolo 6 Viene presentato un possibile sviluppo futuro.

[3] in una rete di computer, con il termine di IP spoofing si indica una tecnica tramite la quale si crea un pacchetto IP nel quale viene falsificato l'indirizzo IP del mittente.

Capitolo 2

Progetto e Implementazione

In questa parte introduciamo il modello che costruiremo; gli *input*, gli *output* e le transizioni tra gli stati della connessione.

Il modello ha un solo processo. Poiché l'obiettivo primario è di trovare possibili modi per attaccare TCP, abbiamo bisogno di conoscere come inviare i giusti segmenti per cambiare lo stato della connessione. Quindi ci concentriamo nella modellazione di come arrivano i segmenti, *usercalls* e *timeouts*, e come questi ultimi cambiano lo stato della connessione, cioè modelliamo gli *input* e le transizioni tra gli stati tralasciando gli *output* fatta eccezione per il *reset*.

```
event:  USERCALL, SEGMENT, TIMEOUT;
```

L'attività del TCP può essere caratterizzata come quella di rispondere ad eventi. Gli eventi che possono occorrere possono essere divisi in tre categorie: *usercalls, arriving segments, e timeouts*. Quindi definiamo tre tipi di *inputs*: *Usercalls, Segments, Timeouts*. La variabile *event* serve per indicare quale evento accade.
Qui di seguito definiamo le usercalls nel nostro modello.

```
   usercalls:  OPEN-P, OPEN-A, SEND, RECEIVE, CLOSE, ABORT;
active_flag:  Boolean;
```

Le usercalls includono *OPEN, SEND, RECEIVE, CLOSE, ABORT* e *STATUS*, Nel nostro modello eliminiamo la usercall STATUS poichè non cambia lo stato della connessione TCP. Ci sono due tipi di OPEN: una è

ATTIVA, l'altra è PASSIVA, cioè apre un *socket*[1] in ascolto. Utilizziamo inoltre una variabile booleana, *active_flag*, per indicare se l'ultima OPEN è attiva. La ragione per cui abbiamo bisogno di questa variabile è perché quando lo stato è SYN-RECEIVED e un segmento contenente un *reset control bit* arriva lo stato potrebbe cambiare in accordo alla precedente OPEN. Se la connessione è stata iniziata con una OPEN passiva (cioè proveniente dallo stato LISTEN), allora TCP deve ritornare allo stato LISTEN. Se invece la connessione è stata iniziata con una OPEN attiva (proveniente cioè dallo stato SYN-SENT) allora la connessione viene rifiutata. TCP dovrebbe entrare nello stato CLOSED, eliminare il TCB (*transfer control block* e ritornare.
Di seguito è mostrato il codice:

```
state = SYN-RECEIVED :
                ...
      event = SEGMENT:
              case
    !seq_ok :  SYN-RECEIVED;
rst_flag & !active_flag :  LISTEN;
 rst_flag & active_flag :  CLOSED;
                ...
              esac;
```

Mostriamo adesso la definizione delle variabili associate ai segmenti:

seq_ok: boolean;	Se il sequence number è accettabile
ack_ok: boolean;	Se l'ACK number è accettabile
prc_flag: LOW, EQUAL, HIGH;	SEG.PRG
urg_flag: boolean;	URG control bit
ack_flag: boolean;	ACK control bit
psh_flag: boolean;	PSH control bit
rst_flag: boolean;	RST control bit
syn_flag: boolean;	SYN control bit
fin_flag: boolean;	FIN control bit

[1] è il punto in cui il codice applicativo di un processo accede al canale di comunicazione per mezzo di una porta, ottenendo una comunicazione tra processi che lavorano su due macchine fisicamente separate.

Bit (left to right)	Meaning if bit set to 1
URG	Urgent pointer field is valid
ACK	Acknowledgement field is valid
PSH	This segment requests a push
RST	Reset the connection
SYN	Synchronize sequence numbers
FIN	Sender has reached end of its byte stream

Fig. 2.1: Significato dei flag

La variabile timeout è definita qui di seguito:

```
timeout:  USER-TIMEOUT, RETRANSMISSION-TIMEOUT, TIMEWAIT-TIMEOUT;
```

La variabile state è illustrata di seguito;

```
state:  LISTEN, SYN-SENT, SYN-RECEIVED, ESTABLISHED, FIN -WAIT -
1, FIN -WAIT -2, CLOSE -WAIT, CLOSING, LAST -ACK, TIME -WAIT, CLOSED;
```

E' raccomandato nel modellare il TCP di seguire la legge di Postel [5], vale a dire il principio di robustezza: *"be conservative in what you do, be liberal in what you accept from others"*. Quindi tutti gli input che includono *event, usercall, timeout, seq_ok, ack_ok, prc_flag, urg_flag, ack_flag, psh_flag, rst_flag, syn_flag, e fin_flag* vengono settati in maniera non deterministica. Per esempio si supponga che arrivi un segmento. Il SYN control bit è settato come segue:

```
next(syn_flag) := {0,1};
```

Ma il flag *active_flag* non può essere settato nondeterministicamente perché è legato alla usercall OPEN.
Quindi viene così definito:

```
next(active_flag) := case
      event = USERCALL & usercall = OPEN-A: 1;
      event = USERCALL & usercall= OPEN-P: 0;
      1:  active_flag;
 esac;
```

Le transizioni tra gli stati sono più complicate. Noi trattiamo i tre eventi in maniera separata e processiamo uno alla volta ognuno di questi tre eventi. Per ogni stato processiamo il segmento validando tutti i *flag* del segmento. Per esempio se lo stato è SYN-SENT e un segmento arriva il processo è il seguente:

1. Controlla l'ACK bit. Se questo è settato ma non accettabile il segmento viene scartato.
2. Se il RST bit è settato e l'ACK è accettabile il prossimo stato è CLOSED.
3. Se il RST bit è settato e l'ACK non è accettabile il segmento è scartato.
4. Se il blocco di sicurezza nel segmento non corrisponde con quello del TCB il prossimo stato rimane SYN-SENT.
5. Se il SYN bit è settato e l'ACK è accettabile il prossimo stato è ESTABLISHED.
6. Se il SYN bit è settato e l'ACK non è accettabile il prossimo stato è SYN-RECEIVED.
7. In tutti gli altri casi lo stato rimane SYN-SENT.

Di seguito viene mostrato il codice:

```
next(state) := case
          ...
         state = SYN-SENT:
            case
            ...
           event = SEGMENT:
             case
                ack_flag & !ack_ok:  SYN-SENT;
                rst_flag & ack_flag & ack_ok:  CLOSED;
                rst_flag & !ack_ok:  SYN-SENT;
                !(prc_flag = EQUAL) : SYN-SENT;
                syn_flag & ack_ok:  ESTABLISHED;
                syn_flag & !ack_ok:  SYN-RECEIVED;
                1:  SYN-SENT;
             esac;
           esac;
         esac;
```

Il processamento delle *usercalls* è relativamente semplice. Per ogni *usercall* assegniamo un valore della variabile *state* in accordo allo stato corrente della connessione.

Il processamento dei *timeout* è altrettanto semplice. Trattiamo solo il caso di USER TIMEOUT. Quando questo tipo di *timeout* si verifica lo stato della connessione passa a CLOSED.

La ragione per cui inseriamo nel nostro modello il *reset* è perché "il reset è inviato" significa che il segmento che arriva è errato oppure che la connessione sarà chiusa. Ciò è utile quando vogliamo determinare che tipo di segmento dovremmo inviare per ottenere una giusta risposta. Analizziamo tre casi:

1. Se la connessione non esiste (CLOSED) allora il *reset* è inviato in risposta a qualsiasi segmento entrante eccetto un altro *reset*.

   ```
   out_rst := case
   ...
   state = CLOSED & !rst_flag:  1;
   ...
   ```

2. Se la connessione è in un altro stato non sicronizzato (LISTEN, SYN-SENT, SYN-RECEIVED) e il segmento che arriva valida qualcosa che ancora non è stato inviato (porta un ACK non accettabile) oppure il blocco di sicurezza è incorretto allora un *reset* è inviato.

   ```
   out_rst := case
   ...
   (state = LISTEN | state = SYN-SENT | state = SYN-RECEIVED)
   & ((ack_flag & !ack_ok) | !(prc_flag = EQUAL)) : 1;
   ...
   ```

3. Se la connessione è in uno stato sincronizzato (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT), per ogni segmento non accettabile (sequence number errato oppure ACK non accettabile) viene inviato un *reset*. Questo viene inviato anche se il campo di sicurezza/precedenza risulta essere errato.

   ```
   out_rst := case
   ...
   (state = ESTABLISHED | state = FIN-WAIT-1 | state = FIN-WAIT-
   2 | state = CLOSE-WAIT | state = CLOSING | state = LAST-ACK
   | state = TIME-WAIT) & (!seq_ok | (ack_flag & !ack_ok) | !(prc_flag=EQUAL))
   : 1;
   ```

4. Se inoltre c'è una *usercall* di tipo ABORT e la connessione si trova nello stato SYN-RECEIVED o ESTABLISHED o FIN-WAIT-1 o FIN-WAIT-2 o CLOSE-WAIT allora un *reset* viene inviato.

```
out_rst := case
...
usercall = ABORT & (state = SYN-RECEIVED | state = ESTABLISHED
| state = FIN-WAIT-1 | state = FIN-WAIT-2 | state = CLOSE-
WAIT): 1;
...
```

Capitolo 3

Verifica consistenza e attacchi

In questa sessione verifichiamo dapprima la consistenza del nostro modello, e poi cerchiamo di trovare possibili attacchi al TCP.

3.1 Consistenza

Il nostro obiettivo finale è quello di trovare possibili modi di attaccare TCP quindi tutte le verifiche devono essere fatte su un modello consistente.

3.2 Transizioni tra gli stati della macchina a stati TCP

Ci sono 22 transizioni nella macchina a stati. Le transizioni verificate sono state la maggioranza, mentre quelle che per cui non è stato possibile effettuare una verifica formale sono state 6.
Passiamo in esame dapprima quelle verificabili.
Queste sono:

- LISTEN –> SYN-RECEIVED
- LISTEN –> SYN-SENT
- SYN-SENT –> ESTABLISHED
- SYN-RECEIVED –> ESTABLISHED

La ragione per cui ho preso in considerazione queste transizioni è dovuta al fatto che ci sono solo due stati (SYN-SENT, SYN-RECEIVED) che portano allo stato ESTABLISHED (stato in cui la connessione tra due *end-point* è stabilita). Quindi questi due stati sono obiettivi di un possibile attacco. Inoltre è stato preso in considerazione lo stato LISTEN in quanto è il primo stato che si incontra nello stabilire una connessione.
Analizziamo le transizioni una a una.
Per ogni transizione verifichiamo dapprima l'esistenza del lato sinistro di quest'ultima.

- **LISTEN⇒SYN-RECEIVED**

  ```
  EF (state = LISTEN & event = SEGMENT & !rst_flag & !ack_flag
  & syn_flag & (prc_flag = EQUAL))
  ```

  ```
  AG ((state = LISTEN & event = SEGMENT & !rst_flag & !ack_flag
  & syn_flag & (prc_flag = EQUAL)) → AX (state = SYN-RECEIVED))
  ```

  Questa proprietà certifica che se il TCP è nello stato LISTEN e un segmento che contiene il SYN control bit e non contiene ACK e RST bit ed ha un corretto livello di precedenza, allora il prossimo stato è SYN-RECEIVED.

- **LISTEN⇒SYN-SENT**

  ```
  EF (state = LISTEN & event = USERCALL & usercall = SEND)
  ```

  ```
  AG ((state = LISTEN & event = USERCALL & usercall = SEND) →
  AX(state= SYN-SENT))
  ```

  La proprietà sopra citata illustra che se lo stato è LISTEN ed il prossimo evento è una usercall del tipo SEND il prossimo stato deve essere SYN-SENT.

- **SYN-SENT⇒ESTABLISHED**

```
EF (state = SYN-SENT & event = SEGMENT & ack_flag & ack_ok
& !rst_flag & (prc_flag = EQUAL) & syn_flag)
```

```
AG((state = SYN-SENT & event = SEGMENT & ack_flag & ack_ok
& !rst_flag & (prc_flag = EQUAL) & syn_flag) → AX(state = ES-
TABLISHED))
```

Se lo stato è SYN-SENT e si hanno le seguenti condizioni: se il segmento contiene un acknowledgment corretto ed un ACK control bit, se non contiene un RST bit, se ha un livello di precedenza esatto ed infine se contiene il SYN flag allora il prossimo stato deve essere ESTABLISHED.

Analizziamo ora il perché di alcune transizioni non verificabili.
Le seguenti transizioni non sono state verificate:

FIN-WAIT-1 → CLOSING
FIN-WAIT-1 → FIN-WAIT-2
FIN-WAIT-1 → TIME-WAIT
CLOSING → TIME-WAIT
LAST-ACK → CLOSED
ESTABLISHED → CLOSE-WAIT

Il motivo della non verificabilità è il seguente.
Esse non sono verificabili in quanto per essere verificabili c'è necessità di mantenere l'informazione se il segmento con l'ACK è in risposta al messaggio di FIN inviato precedentemente.
Ma nel nostro modello non mantenendo informazioni circa quello che viene inviato non è possibile sapere se l'ACK è in risposta o meno al FIN.

3.3 Altri comportamenti del TCP

In questa sessione verifichiamo delle proprietà collegate all'invio di *reset*, utili nel determinare possibili segmenti errati.

- *Se non esiste alcuna connessione allora un reset è inviato*

  EF(state = CLOSED & ack_flag & !rst_flag & event = SEGMENT)

  AG((state = CLOSED & ack_flag & !rst_flag & event = SEGMENT) → (out_rst))

- *Se la connessione è in uno stato non sincronizzato (LISTEN, SYN-SENT, SYN-RECEIVED), e il segmento che arriva valida qualcosa che ancora non è stato inviato (ACK non accettabile) oppure se il segmento ha un livello di sicurezza incorretto allora viene inviato un reset.*

  EF(state = SYN-RECEIVED & seq_ok & !rst_flag & !syn_flag & ack_flag & !ack_ok & event = SEGMENT)

  AG((state = SYN-RECEIVED & seq_ok & !rst_flag & !syn_flag & ack_flag & !ack_ok & event = SEGMENT) → (AX(state = SYN-RECEIVED) & AF(out_rst)))

- *Se la connessione è in uno stato sincronizzato (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT), per qualsiasi segmento non accettabile (fuori dalla finestra temporale o con un numero di acknowledge errato) la connessione rimane nel medesimo stato. Se il segmento ha un livello di sicurezza incorretto allora viene inviato un reset.*

  EF(state = ESTABLISHED & !rst_flag & !syn_flag & ack_flag & (!ack_ok | !seq_ok) & event = SEGMENT)

  AG((state = ESTABLISHED & !rst_flag & !syn_flag & ack_flag & (!ack_ok | !seq_ok) & event = SEGMENT) → (AX(state = ESTABLISHED) & AF(out_rst)))

Capitolo 4

Attacchi

Come menzionato nella sessione 1.4, un tipo di possibile attacco del TCP è quello "denial-of-service" (DoS), come ad esempio il TCP SYN flooding [7].

Molti attacchi DoS non sfruttano un *bug* del software ma piuttosto un difetto nella particolare implementazione del protocollo.

Dapprima utilizzeremo il nostro modello per confermare l'attacco TCP SYN flooding e infine cercheremo di trovare un nuovo modo di attaccare TCP.

È noto che ogni volta che ad un client arriva su una porta valida (una porta sulla quale il server TCP è in ascolto) un segmento contenente un SYN, un TCB (*transfer control block*) deve essere allocato. Se non ci fossero limiti sul numero di richieste di connessioni concorrenti, un host potrebbe facilmente esaurire tutta la memoria soltanto per processare le connessioni TCP. Per fortuna TCP ha un limite superiore sul numero massimo di connessioni concorrenti, chiamato *backlog*. Ed esso è pari alla lunghezza della coda in cui le connessioni in ingresso e quelle incomplete vengono mantenute. Se il limite di backlog viene raggiunto tutte le nuove connessioni in ingresso non vengono considerate.

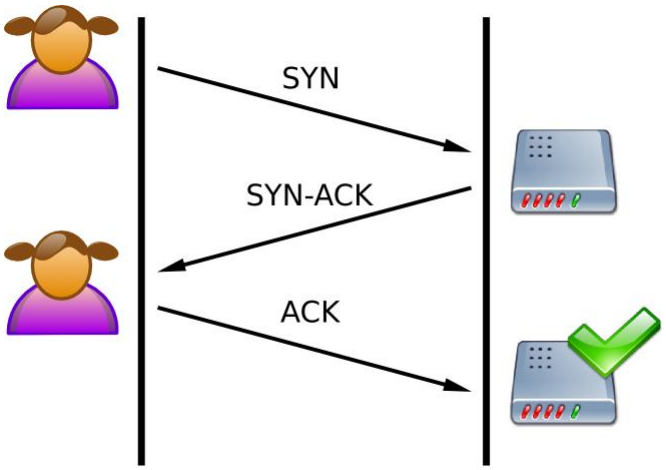

Fig. 4.1: Una connessione normale tra un utente ed un server. Il meccanismo del *"3way-handshake"* è realizzato correttamente

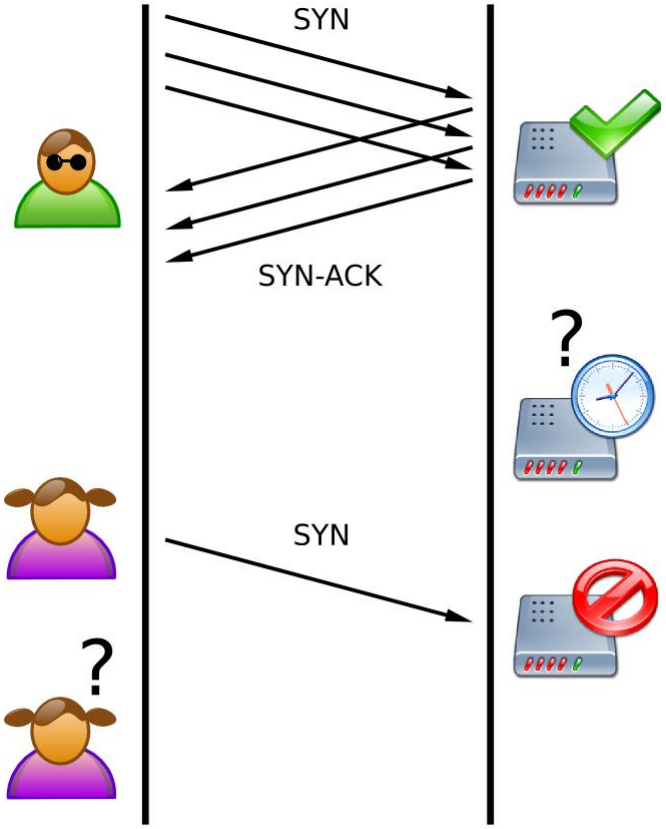

Fig. 4.2: SYN-flooding. Colui che attacca invia numerosi pacchetti SYN ma non invia al server l'ACK. Le connessioni, quindi, sono aperte per metà ed occupano risorse al server. Un utente cerca di connettersi ma il server rifiuta di aprire la connessione ed è quindi sotto attacco

Innanzitutto utilizzeremo NuSMV per produrre un controesempio che ci mostra l'esistenza di un possibile attacco di tipo SYN flooding e in seguito costruiremo un segmento per poi verificarlo sempre con NuSMV.

Poiché gli attacchi di tipo SYN flooding si hanno quando lo stato è in SYN-RECEIVED, si vuole verificare che se lo stato è SYN-RECEIVED allora prima o poi lo stato diverrà ESTABLISHED.

AG (state = SYN-RECEIVED → AF state = ESTABLISHED)

Inoltre poichè quando è in corso un tale attacco non ci sono nè segmenti in arrivo nè usercalls aggiungiamo un proprietà di fairness[1]:

FAIRNESS
state = SYN-RECEIVED →
!(event = USERCALL | event = SEGMENT)

NuSMV produce un controesempio. Il controesempio, illustrato nella figura seguente, assicura che se lo USER TIMEOUT avviene infinitamente spesso quando lo stato è SYN-RECEIVED allora lo stato non sarà mai ESTABLISHED, e ciò mostra la possibilità di un attacco di tipo SYN flooding.

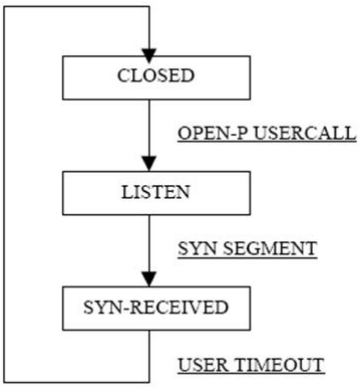

Fig. 4.3: Controesempio prodotto da NuSMV

Vedremo ora di costruire un segmento per questo attacco.
Prima di tutto, cerchiamo di costruire un SYN segment che può essere accettato da un host in ascolto.

Ci sono delle parti molto importanti nell'header di un segmento TCP: indirizzo IP, numero di sequenza, bit di controllo.

[1] un vincolo di fairness, in NuSMV, restringe l'attenzione del *model checker* solo a quei cammini lungo i quali una formula è vera infinitamente spesso

In accordo alla specifica del TCP sappiamo che l'host dall'altro lato non verifica il numero di sequenza di un segmento SYN. Quindi possiamo utilizzare qualsiasi tipo di sequence number. Potremmo anche utilizzare l'indirizzo IP reale. Ma non è una buona idea. Il modo migliore è quello di falsificare l'indirizzo IP in modo tale che nessuno possa capire da dove provenga il segmento.

Costruiamo adesso il segmento. Verifichiamo dapprima che sia possibile cambiare lo stato da LISTEN a SYN-RECEIVED quando arriva un segmento SYN.
Ecco la proprietà:

EF(event = SEGMENT & state = LISTEN & syn_flag → AX(state = SYN-RECEIVED))

La proprietà è verificata essere vera. Ora vogliamo verificare se questo attacco possa avvenire sempre.

Se cambiamo semplicemente "EF" con "AG" notiamo tramite il contro esempio che NuSMV ci mostra che la proprietà non è sempre vera.
La ragione a ciò è il dover settare alcuni bit di controllo come RST, ACK, SEG.PRC. Quindi la proprietà corretta è la seguente:

AG(event = SEGMENT & state = LISTEN & syn_flag & !rst_flag & !ack_flag & prc_flag = EQUAL → AX(state = SYN-RECEIVED))

Dalle proprietà verificate concludiamo che ci sono tre passi nel nostro attacco:

1. Inviare un corretto SYN segment alla vittima;
2. Non rispondere a nessun messaggio inviato dalla vittima e far si che la connessione vada in timeout. Un modo per fare ciò è di inserire un indirizzo IP falso.
3. Quando la memoria del backlog è piena tutte le richieste di connessione vengono ignorate

L'HOST E' SOTTO UN ATTACCO SYN FLOODING !!!

4.1 Un nuovo attacco

Usando il nostro modello cerchiamo di trovare un nuovo modo di attaccare un host con attacchi di tipo denial-of-service.

Abbiamo notato in precedenza che un segmento con un SYN control bit può causare la chiusura della connessione.

Ci chiediamo ora se la connessione potrebbe essere chiusa quando c'è un segmento con un corretto sequence number e un SYN control bit e quando lo stato corrente non è né LISTEN né SYN-SENT.

AG(!(state = LISTEN | state = SYN-SENT) & event = SEGMENT & seq_ok & syn_flag → AX(state = CLOSED))

Ovviamente la proprietà non è vera. Il motivo è perché il TCP controlla il RST bit e il flag di sicurezza e precedenza prima del SYN flag. Quindi la proprietà corretta è la seguente:

AG(!(state = LISTEN | state = SYN-SENT) & event = SEGMENT & seq_ok & !rst_flag & (prc_flag = EQUAL) & syn_flag → AX(state = CLOSED))

Il seguente potrebbe essere un attacco realistico:

- Il cracker sa che c'è una connessione tra due host A e B
- Il cracker vuole rompere la connessione
- Per fare ciò deve fare in modo che A o B chiudano la connessione
- Supponiamo che voglia far chiudere la connessione ad A. Esso falsifica l'indirizzo di B ed invia il segmento SYN ad A.
  In seguito A chiuderà la connessione.

Così se il cracker può scoprire e distruggere ogni connessione che l'host vittima vuole stabilire, allora l'host è sotto un attacco di tipo DoS.
La difficoltà di questo attacco risiede nel poter acquisire nella maniera esatta i numeri di sequenza. Un modo per far ciò è attraverso lo sniffing.

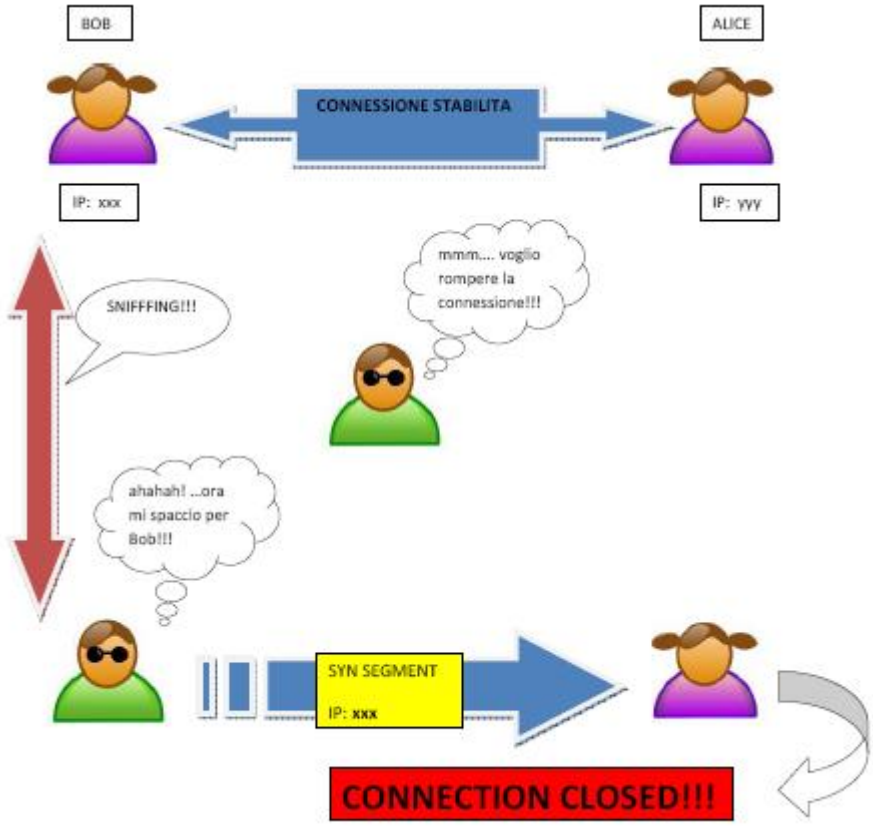

Fig. 4.4: Esempio di un nuovo attacco

Capitolo 5

Conclusioni

In questo progetto è stato modellato con successo il TCP, è stato in seguito confermato il ben conosciuto attacco TCP SYN flooding ed infine si è cercato di trovare un nuovo attacco al TCP.

I risultati di questo progetto hanno mostrato come i metodi formali sono di grande utilità nella verifica di protocolli. Sono inoltre di grande aiuto nella modellazione di sistemi complessi. Tuttavia come in qualsiasi progetto degno di nota bisogna tener conto che alcune astrazioni sono necessarie.

Capitolo 6

Sviluppi futuri

È possibile modellare con NuSMV anche il concetto di memoria, quindi sarebbe molto interessante come possibile sviluppo cercare di modellare il concetto di memoria di back-log e verificare mediante controesempi l'impossibilità di effettuare connessioni pari ad un numero superiore alla memoria di back-log.

Bibliografia

[1] Douglas E. Comer. *Internetworking with TCP/IP, volume 1.*

[2] NuSMV. NuSMV. URL: nusmv.irst.itc.it.

[3] Wikipedia. Computational Tree Logic da Wikipedia. URL: en.wikipedia.org/wiki/Computational_tree_logic.

[4] Wikipedia. Denial of Service da Wikipedia. URL: it.wikipedia.org/wiki/Denial_of_service.

[5] Wikipedia. Jon Postel da Wikipedia. URL: it.wikipedia.org/wiki/Jon_Postel.

[6] Wikipedia. Linear Temporal Logic da Wikipedia. URL: http://en.wikipedia.org/wiki/Linear_temporal_logic.

[7] Wikipedia. SYN flooding da Wikipedia. URL: it.wikipedia.org/wiki/SYN_flood.

Elenco delle figure