UNIVERSITÀ DI PISA

Facoltà di Ingegneria

Corso di Laurea in Ingegneria delle Telecomunicazioni

Tesi di Laurea Magistrale

Realizzazione di un Test-Bed per l'implementazione di protocolli di routing sicuri in reti MANET

Relatori:

Prof. Michele Pagano

Prof. Stefano Giordano

Ing. Christian Callegari

Candidato:

Giuseppe Cocilovo

Anno Accademico 2012/2013

Alla mia famiglia.

Indice

Elenco delle tabelle

Elenco delle figure

Introduzione

Ultimamente la società si sta evolvendo verso scenari dominati dalla mobilità. Tale mobilità è dettata principalmente dalla necessità di ogni persona di essere continuamente, indipendentemente dal luogo dove si trova, in comunicazione con altre persone ad essa legate da interessi di lavoro, culturali, sociali. Ognuno di noi quotidianamente si trova di fronte alla necessità di dover risolvere dei problemi pur essendo lontani fisicamente dal sito dove si è verificato il problema stesso. Per rispondere a tali ed altre esigenze l'innovazione tecnologica ha prodotto i dispositivi mobili e nuove tipologie di rete. In particolare, l'attenzione è stata rivolta alle reti MANET.

Dal punto di vista delle telecomunicazioni, una Mobile Ad-hoc NETwork (MANET) è definita come un sistema autonomo di terminali mobili connessi mediante collegamenti wireless di tipo ad-hoc. La definizione di rete ad hoc fornita dall'IETF è:

"Una MANET è un sistema autonomo di router mobili e host associati, connessi con collegamenti di tipo wireless, formando un grafo di forma arbitraria. Tali router sono liberi di muoversi casualmente e di auto organizzarsi arbitrariamente, sebbene la topologia wireless vari rapidamente ed in modo imprevedibile. Tale rete può operare in isolamento oppure essere connessa alla rete Internet" [11].

Tutti i nodi del sistema collaborano con lo scopo di instradare i pacchetti nel modo corretto secondo la modalità di forwarding di tipo multihop. Una MANET è quindi una rete wireless dinamica che si può costruire all'occorrenza ed utilizzare in ambienti estremamente dinamici senza un'infrastruttura preesistente o una gestione centralizzata. In queste reti ogni nodo può comportarsi oltre che da sorgente o destinazione anche come router, ragion per cui le MANET sono utilizzate in applicazioni quali comunica-

zioni sui campi di battaglia, scenari di emergenza o eventi locali come conferenze e meeting pubblici.

Quando si parla di MANET bisogna tener presente che, essendo create al momento, non godono di grandi risorse energetiche e che per loro natura non hanno una chiara linea di difesa. In modo particolare il fatto di non avere una propria 'linea di difesa' fa si che ne possono entrare a far parte sia utenti legittimi sia attackers, ossia utenti malintenzionati che hanno libero accesso al canale di comunicazione. Generalmente ogni nodo annuncia la sua presenza nella rete ed ascolta la comunicazione tra gli altri nodi, che dunque diventano conosciuti. Col passare del tempo ogni nodo acquisisce la conoscenza di tutti i nodi della rete e di uno o più modi per comunicare con loro; quindi, non essendoci router dedicati, ogni nodo, come detto pocanzi, può funzionare sia da host che da router e inoltrare quindi pacchetti ad altri host della rete.

Le politiche di routing presenti ad oggi nelle MANET suppongono di lavorare in una ambiente basato sulla fiducia ed è proprio questo uno dei punti deboli di tali reti. Infatti può accadere, con estrema facilità, che un attac-

ker diventi un router e andando contro quelle che sono le specifiche del protocollo, riesca a compromettere le operazioni della rete. In un contesto del genere ha un suo peso, quindi, il fatto di rendere la rete configurata ad-hoc il più sicura possibile, cercando di proteggerla attraverso l'analisi dei punti deboli nei vari attacchi. Tipicamente gli attacchi vengono fatti sfruttando i messaggi previsti dal protocollo stesso, poichè l'invio di questi permette appunto di diffondere informazioni non veritiere, compromettere le varie route, l'esclusione di alcuni nodi e così via. In uno scenario come quello delle MANET, diventa quindi importante trovare una soluzione per evitare la presenza di nodi malevoli e render sicura la rete proteggendola dai vari attacchi; nello stesso tempo però si suppone che i terminali non abbiamo elevate risorse, sia in termini di velocità computazionale che di potenza disponibile per le applicazioni e di durata di batteria, ed è proprio per questo motivo che queste reti sono perennemente soggette ad attacchi come il Blackhole, Wormhole, Replay, Jellyfish, Neighbor piuttosto che attacchi di tipo IP Spoofing o Link Spoofing.

In questo testo vengono studiati e descritti gli attac-

chi precedentemente elencati con il fine di cercare di implementare, sfruttando al meglio le risorse, una versione sicura di due dei possibili protocolli utilizzati nelle reti MANET.

L'elaborato è strutturato come segue.

Nel capitolo 1, viene fatta una panoramica sulle reti MANET e sui vari protocolli di routing utilizzati, focalizzando l'attenzione sui protocolli utilizzati per implementare le operazioni di routing e gli attacchi.

Nel capitolo 2, viene descritto il principio base di ciascun attacco e con esso vengono descritte anche le possibili conseguenze, facendo quindi, attraverso la presentazione di un'opportuna topologia, considerazioni in merito alla posizione dell'attacker.

Nel capitolo 3, vengono descritti la realizzazione del test-bed e gli strumenti utilizzati per cercare di creare una versione sicura dei protocolli. In particolare viene descritto il protocollo OLSR e come si può implementare una prima linea di difesa.

Nel capitolo 4, vengono analizzati i vari attacchi ed in particolare le debolezze di ognuno di essi; viene spiegato

inoltre come partendo appunto da tali debolezze è stata implementata una versione sicura del protocollo OLSR.

Nel capitolo 5, viene fatto lo studio degli attacchi rivolti ad un protocollo reattivo quale l'AODV. Successivamente viene descritto come attraverso uno studio teorico di questi è possibile implementare delle soluzioni che potrebbero contribuire a garantire la sicurezza nelle reti in questione.

Capitolo 1

Le Reti Manet

Le MANET [1](Mobile Ad hoc NETwork) sono un tipo di rete innovativo formato da dispositivi wireless, quindi mobili, capaci di comunicare tra loro svolgendo operazioni di instradamento, senza alcun bisogno di una infrastruttura, come avviene invece per la telefonia cellulare che ha bisogno per esempio di antenne e ripetitori. Le reti ad-hoc sono un sistema di comunicazione dati basato su trasmissioni radio, nelle quali i nodi comunicano esclusivamente su canali wireless. Le reti MANET si basano sul concetto Peer-to-

Peer (P2P), dove i peer non sono altro che normalissimi dispositivi quali PC, palmari, smartphone, laptop,etc... Il fulcro di questo tipo di architettura è appunto l'assenza di un server centrale che amministri le connessioni e le risorse. Una rete MANET dunque è una rete temporanea, senza punti di accesso, che non viene preconfigurata, ma si forma per la sola presenza dei vari dispositivi in un dato territorio, auto-configurandosi ed auto-organizzandosi. I nodi nella rete in questione si comportano non solo come host ma anche come router.

Nella Figura 1.1 viene presentata una possibile topologia della rete, da cui si evince che i dispositivi che partecipano alla MANET possono comunicare, a seconda delle loro esigenze, con tutti gli altri nodi.

Ovviamente, come in tutte le altre tipologie di rete anche nelle reti configurate ad-hoc quando si vuole trovare un percorso che permetta di inoltrare pacchetti da sorgente a destinazione bisogna adottare delle opportune procedure di routing. Nel caso di MANET bisogna tener presente che comunque, a differenza delle reti cablate, si hanno link asimmetrici, interferenze tra le varie comunicazioni e una

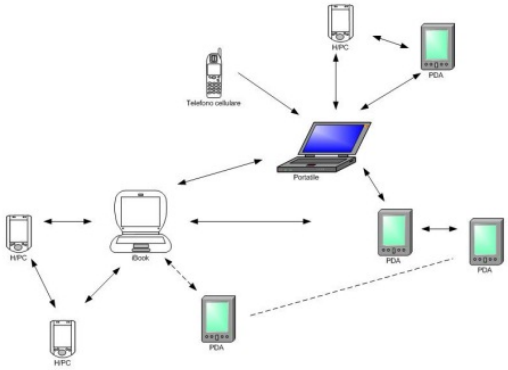


Figura 1.1: Rete Ad-hoc

topologia in continua evoluzione [2]. Uno dei principali problemi presenti in queste tipo di rete, data la mobilità dell'ambiente e la disponibilità limitata di risorse, è appunto quello della scelta delle varie politiche di routing; infatti si è sempre in conflitto poichè la rete per sua natura richiede uno scambio di informazioni constante, ma per i motivi precedentemente elencati si ha la necessità di limitare il traffico il più possibile.

1.1 Caratteristiche delle MANET

É possibile riassumere le caratteristiche più salienti di una rete a d-hoc, quale la MANET, come segue:

- Topologia Dinamica.

  I nodi sono liberi di muoversi arbitrariamente, perciò la topologia della rete può cambiare in maniera imprevedibile in maniera random (nello spazio).

- Vincoli di banda e capacità variabile dei link.

  A differenza delle reti 'infrastrutturate', in queste i collegamenti wireless tra nodi non godono di grosse capacità trasmissive.

- Operazioni vincolate dall'energia.

  Tipicamente l'energia dei nodi di una rete ad-hoc dipende da batterie o strumenti energetici simili. Per questi nodi quindi è importante studiare dei criteri di progettazione cosicchè si possa ottimizzare il consumo energetico.

- Sicurezza limitata.

Per la loro natura, le reti mobili wireless sono generalmente più esposte ad attacchi rispetto alle reti cablate. Pertanto, la sempre più crescente possibilità di attacchi quali ascolti, spoofing e denial-of-service deve essere tenuta in conto dai progettisti che realizzano sistemi per queste reti.

1.2 I protocolli

Come detto pocanzi uno dei problemi principali è la scelta delle opportune politiche di routing. In particolare gli algoritmi di routing devono:

- creare le tabelle di routing in modo tale da essere più piccole possibili;
- aggiornare costantemente le tabelle di routing;
- eventualmente offrire path multipli per una stessa destinazione;
- valutare i vari percorsi e riuscire a scegliere il migliore per raggiungere una destinazione.

I protocolli possono essere classificati in base alla modalità di reperimento delle informazioni e di instradamento. Possiamo distinguere protocolli **Proattivi** (che trasmettono informazioni in **modalità proactive**) e protocolli **Reattivi** (che scambiano informazioni in **modalità reactive**).

1.2.1 Modalità Proactive

In modalità Proactive i nodi si scambiano informazioni di routing a intervalli fissi di tempo; questa caratteristica permette che ci sia un percorso immediatamente disponibile ad ogni richiesta; purtroppo ciò comporta un elevato traffico di segnalazione.

Negli algoritmi Proactive (o Table-Driven) l'instradamento ad ogni richiesta è disponibile immediatamente grazie al fatto che i vari dispositivi mobili si scambiano periodicamente informazioni di routing. L'idea è di mantenere costantemente aggiornate le informazioni di instradamento tra tutti i dispositivi che compongono la rete.

I protocolli che fanno parte di questa classe sono i seguenti:

1. DSDV (Destination Sequenced Distance Vector).
2. WRP (Wireless Routing Protocol).
3. CSGR (Cluster Switch Gatewey Routing).
4. STAR (Source Tree Adaptive Routing).
5. OLSR (Optimized Link State Routing).

I protocolli che fanno parte della famiglia Proactive sono accomunati dalle seguenti caratteristiche:

- Scambio di pacchetti ad intervalli fissi.
- Uso di tabelle.
- Aggiornamento delle tabelle.

I pacchetti rivelano sia le informazioni di instradamento sia i relativi cambiamenti topologici della rete. L'uso di una o più tabelle che memorizzano tutte le informazioni riguardanti la topologia della rete è una caratteristica peculiare della classe Proactive. In particolare andiamo a vedere come funziona l'OLSR, poichè quello che rispecchia più le nostre esigenze.

OLSR

OLSR [3] è un algoritmo di tipo proattivo, in cui inizialmente ogni nodo collabora con i suoi vicini per costruire una tabella di instradamento contenente i path verso tutti i nodi esistenti. Dopo questa inizializzazione i nodi si scambiano periodicamente dei messaggi per mantenere aggiornati i propri dati topologici. Come suggerisce il nome dell'algoritmo, OLSR utilizza il broadcast delle informazioni sullo stato di tutti i collegamenti conosciuti ad ogni nodo per permettere la ricostruzione della topologia di rete. Questo broadcast viene però ottimizzato, per limitare al massimo lo spreco di banda, utilizzando un sistema chiamato **MultiPoint Relaying (MPR)**. La tecnica MPR nasce dall'osservazione che in una situazione di broadcast non ottimizzato ogni nodo riceve più volte le stesse informazioni causando un notevole spreco di banda e di potenza di elaborazione. Questa situazione può essere migliorata scegliendo un particolare sottoinsieme di nodi che possono ritrasmettere le informazioni.

In OLSR (vedi Fig. 1.2) ogni nodo sceglie un sottoinsieme dei suoi vicini simmetrici tale che ogni secondo vi-

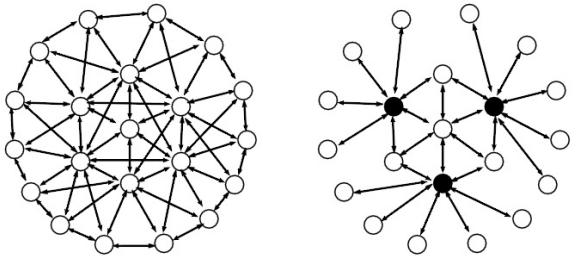


Figura 1.2: Invio di pacchetti senza e con l'utilizzo di MPR

cino sia raggiungibile tramite questo sottoinsieme. Il sistema MPR viene usato come metodo di default per la ritrasmissione dei messaggi. Il traffico di controllo che viene utilizzato per il calcolo dei percorsi è generato dai nodi scelti come MPR per accettare anche la perdita di qualche messaggio; inoltre poichè in questo protocollo ogni pacchetto ha un proprio sequence number che lo identifica in maniera univoca, non è richiesta la consegna ordinata. In una rete possono esistere più MPR, infatti ogni nodo della MANET seleziona, tra i vicini con i quali ha un link bidirezionale, un set di multi point relay; questo set è scelto con l'unica peculiarità di far arrivare le informazioni di

routing a tutti gli host a due hop di distanza passando dal minor numero di MPR. I neighbors che non appartengono al set ricevono ed elaborano i messaggi broadcast ma non li ritrasmettono.

OLSR utilizza pacchetti UDP, con numero di porta pari a 698, per trasferire le informazioni di controllo, ogni pacchetto può contenere più di un messaggio proveniente da mittenti differenti, per ottimizzare il tempo di trasmissione. I messaggi richiesti dalla funzionalità base di OLSR sono:

HELLO: inviati ad intervalli regolari, compiono le funzioni di rilevamento dei vicini, comunicazione dei nodi MPR e link sensing

TC (Topology Control): servono a comunicare informazioni topologiche dal punto di vista di ogni nodo

MID (Multiple Interface Declaration): usati dai nodi con più interfacce per dichiararne l'esistenza al resto della rete.

Un generico pacchetto è descritto nella Figura 1.3.
Descriviamo quindi alcuni campi:

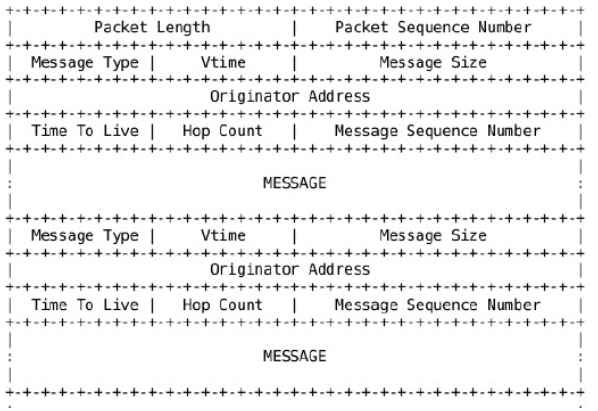


Figura 1.3: Formato di un pacchetto OLSR generico contenente due messaggi.

- *Packet Length*: valore che indica la lunghezza del pacchetto

- *Packet Sequence Number*: valore che identifica in maniera univoca ogni singolo pacchetto

- *Message Type*: tipo di messaggio (HELLO, TC oppure MID)

Andiamo quindi a descrivere brevemente il funziona-

mento di OLSR. Ogni nodo ad intervalli regolari, manda un messaggio di HELLO. Grazie all'utilizzo di questo messaggio i nodi annunciano la lista dei vicini e anche il tipo, indicando attraverso il campo Link Code se un nodo è un MPR oppure un semplice host. I messaggi di HELLO a differenza di tutti gli altri vengono scambiati localmente senza appunto esser inoltrati, i messaggi TC invece vengono utilizzati per calcolare i vari percorsi e non vengono inviati in locale. Ogni MPR ha il compito inoltre di inviare periodicamente TC messages i quali contengono appunto la lista dei vicini che lo hanno scelto. In questa tipologia di messaggi assume particolare importanza il parametro ANSN (Advertised Neighbor Sequence Number), un numero associato al set di vicini annunciati che viene incrementato di uno ogni volta che il nodo rileva un cambiamento nel set stesso. Da ciò che è stato detto sin ora, si evince il fatto che, come previsto dall'rfc [3], per il calcolo della tabella di routing, si preferiscono path con il minor numero di hop, ciò implica però, il fatto che sarà preferito un percorso attraverso un singolo link, anche se non buono, ad una route che passa da due link eccellenti, nonostante la seconda scel-

ta sia la migliore. Quello descritto pocanzi è appunto un problema che viene risolto attraverso l'utilizzo dell'estensione 'Link Quality' di OLSR che permette di conoscere la qualità dei link. Ogni nodo non fa altro che valutare la packet loss per i pacchetti che riceve dai vicini. Attraverso l'invio di più pacchetti di HELLO viene calcolata la LQ (Link Quality), cioè la probabilità che la trasmissione vada a buon fine; altro parametro calcolato è la NLQ (Neighbor Link Quality) che indica la qualità del link nella direzione opposta. Infine attraverso la combinazione di queste due viene calcolata la ETX (Expected Trasmission Count) che indica la qualità del link in entrambe le direzioni.

$$\text{EXT} = 1/(\text{LQ} * \text{NLQ})$$

Se si ha a che fare con una route che passa da più link, calcolare l'ETX totale vuol dire semplicemente fare la somma dei singoli ETX.

Infine, dato che comunque nelle MANET uno dei problemi principali è sempre quello di ridurre al massimo l'uso di risorse, si utilizzano altri due tipi di messaggi:

LQ-HELLO: per ogni link annunciato, l'originator comunica anche il valore di LQ che si è calcolato.

LQ-TC: porta informazioni su quanto sono buoni i link.

Il fine unico è quello dunque di scegliere il percorso con un ETX totale minore e non più in base al numero di hop.

1.2.2 Modalità Reactive

In modalità Reactive, viene invocata una procedura per determinare il corretto instradamento solo se si vuole trasmettere un pacchetto. Ciò porta ad avere un ritardo nella trasmissione dei pacchetti dati, riuscendo però ad abbassare notevolmente il traffico di segnalazione. In un contesto del genere si dice che i percorsi vengono creati *On Demand*, il concetto che sta alla base di questi protocolli è appunto quello di evitare di salvare le varie route poichè queste, a causa della mobilità dei nodi, sono in continuo mutamento.

I protocolli che fanno parte di questa classe sono i seguenti:

1. AODV (Ad Hoc On-Demand Distance Vector Routing).
2. DSR (Dynamic Source Routing).
3. TORA (Temporally Ordered Routing Algorithm).
4. PAR (Power-Aware Routing).
5. LAR (Location-Aided Routing).

I protocolli che fanno parte della famiglia Reactive sono accomunati dalle seguenti caratteristiche:

- Scoperta del percorso tramite Route discovery.
- Mantenimento del percorso tramite Route Manteinence.
- Cancellazione del percorso con Route Deletion.

AODV

AODV [4], in quanto facente parte della classe dei protocolli reattivi, permette di ottenere un percorso per una

nuova destinazione senza richiedere ai nodi di mantenere routes per destinazioni che non sono attive. Uno dei concetti che sta alla base di questo protocollo è quello di *fiducia reciproca* ed inoltre, affinchè si possano migliorare la scalabilità e le performance, si propone di ridurre overhead e quantità di traffico di controllo presente nella MANET.

Il protocollo in questione utilizza pacchetti UDP, con numero di porta 654.

I messaggi richiesti dalla funzionalità base di AODV sono:

RREQ (Route REQuest): inviati quando un nodo vuole raggiungere una destinazione della quale non ha alcuna route disponibile (vedi Fig. 1.4).

RREP (Route REPly): utilizzati per rispondere ad una richiesta arrivata tramite RREQ (vedi Fig. 1.5).

RERR (Route ERRor): inviati in presenza di errori o nel caso in cui viene a mancare qualche link (vedi Fig. 1.6).

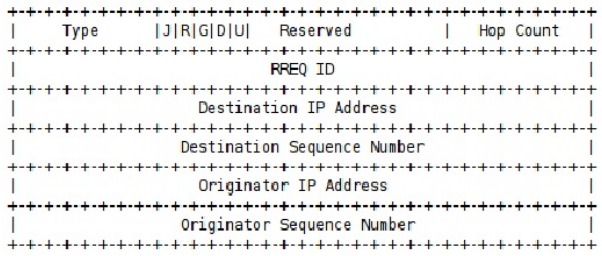


Figura 1.4: Struttura messaggio RREQ

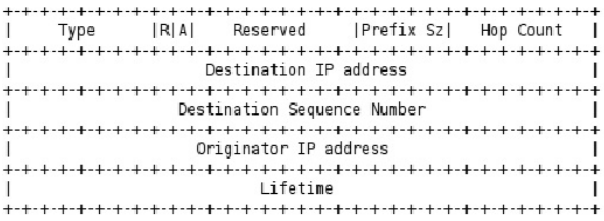


Figura 1.5: Struttura messaggio RREP

Il parametro che caratterizza l'AODV Protocol è il *Destination Sequence Number*. Esso è creato dal nodo di destinazione, per accompagnare le informazioni relative al percorso a raggiungerlo, evitando così la creazione di loop. Un nodo che desidera offrire informazioni sulla connettività invia, ogni HELLO_INTERVAL, in broadcast HELLO

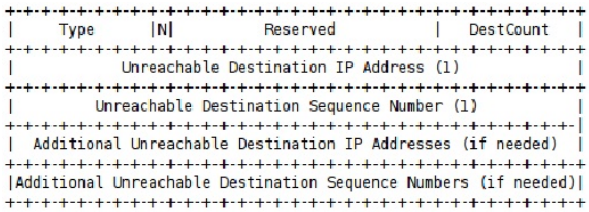


Figura 1.6: Struttura messaggio RERR

messages, possiamo brevemente descrivere la fase di 'Route Discovery' nel seguente modo:

quando il nodo sorgente richiede un percorso, per raggiungere il nodo di destinazione inizia una fase di flooding inviando richieste a tutta la rete. Questa richiesta può arrivare sia direttamente a destinazione, sia ad un nodo intermedio, il quale deve essere in possesso di un'informazione sufficientemente aggiornata. Una volta che un qualsiasi nodo, indistintamente dal fatto che possa essere la destinazione o no, riceve una richiesta, deve immediatamente rimandare al nodo sorgente una risposta, la quale indica l'avvenuta ricezione dell'informazione. Ciò può essere fatto utilizzando il percorso inverso. Per essere certi di costruire percorsi senza loop e contenenti informazioni

aggiornate, ogni nodo mantiene un numero di sequenza. Tale numero, identifica univocamente ogni percorso ed è per questa ragione che viene ogni volta incrementato, così che un nodo riesca a capire se ne è già in possesso o meno. Entrando più nel dettaglio, quando il nodo sorgente richiede una route, oltre ad effettuare una richiesta di scoperta dei vicini, invia un pacchetto RREQ che rappresenta l'instanza di un percorso per giungere a destinazione. La fase di Route Discovery continua attraverso l'invio dei pacchetti RREQ anche dai nodi intermedi, che contattano altri nodi a loro adiacenti, finchè si raggiunge il nodo richiesto. Il pacchetto RREQ viene identificato dall'ID e dall'indirizzo IP del nodo sorgente.

Per concludere nella Figura 1.7 viene mostrata l'operazione di Route Discovery, in cui il nodo N1 inoltra una richiesta a tutti i suoi vicini (ciò viene determinato dall'orientamento delle frecce), che a loro volta spediscono pacchetti RREQ giungendo a destinazione. Data la bidirezionalità dei link, una volta raggiunta la destinazione, quest'ultima rimanderà tramite un percorso inverso (nel caso proposto il più corto, ma non è sempre detto) un pacchet-

to RREP (Route Reply) contenente il percorso (insieme di indirizzi IP dei nodi) che è stato fatto per raggiungerla.

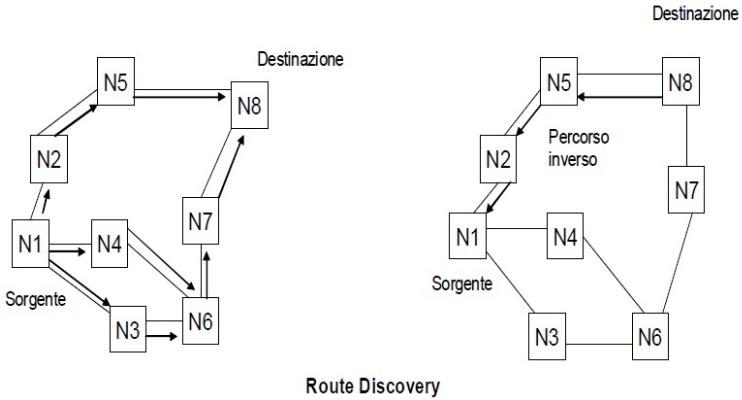


Figura 1.7: Protocollo AODV

Capitolo 2

Attacchi alle reti MANET

Le caratteristiche delle MANET, quali utilizzo di un canale wireless accessibile a tutti, topologia in continuo cambiamento, mancanza di gestione e monitoraggio centralizzato e assenza di alcun tipo di linea di difesa, fanno si che esse siano vittime di attackers. Infatti, dato che qualsiasi nodo può, liberamente e come meglio crede, unirsi alla rete e diventare parte integrante di essa, comunicare, inoltrare,

ricevere pacchetti e staccarsi dalla rete senza alcun tipo di opposizione, gli attacchi avvengono con estrema facilità e senza alcun impedimento. Alla base di tutto ciò vi è inoltre la mancanza di un vero e proprio organo centrale che si occupi di gestire la rete, evitando così di far accedere ad essa chiunque. Nel presente capitolo vengono introdotti e descritti i vari attacchi che una rete MANET può subire. Come detto pocanzi un ipotetico attacker può portare a termine vari attacchi, che posso essere raggruppati secondo due aspetti:

- ATTACCHI ESTERNI e INTERNI
- ATTACCHI ATTIVI e PASSIVI

Come è facile intuire la prima classificazione è basata sulla provenienza degli attacchi mentre la seconda sul comportamento dell'attacker.

Gli *attacchi esterni* hanno il fine unico di causare congestione o disturbare i nodi nel fornire servizi e vengono compiuti da nodi che non fanno parte della rete. Gli *attacchi interni* invece vengono compiuti da uno o più nodi malevoli che riescono ad entrare a fare parte della rete,

partecipando quindi alle attività di routine, oppure riescono a compromettere un nodo già attivo inducendolo ad un comportamento scorretto. Ovviamente tutto ciò è possibile proprio perchè ogni nodo ripone piena fiducia negli altri host della rete. Gli attacchi di questa classe sono estremamente pericolosi, infatti un nodo malevolo può compiere operazioni sui pacchetti, quali modificarli, replicarli e crearne nuovi; inoltre l'attacker può deviare il traffico, negare un servizio o peggiorare le prestazioni di un link. Passando alla seconda classificazione possiamo dire che gli attackers che compiono *attacchi attivi* si comportano in maniera tale da compromettere i messaggi che viaggiano in rete, modificandoli o scartandoli, in modo tale da distruggere le funzionalità della rete [5]. Gli *attacchi passivi* invece non distruggono le normali operazioni della rete, ma lo scopo dell'attacker è appunto quello di ricavare più informazioni possibili attraverso lo sniffing dei dati che viaggiano nella rete. Questi, a differenza di tutti gli altri attacchi, sono più difficili da rilevare poichè le normali attività della rete non vengono compromesse.

Passiamo quindi alla descrizione nel dettaglio dei vari attacchi.

2.1 Byzantine Attack

In questo attacco vi è la presenza di un nodo corrotto che, senza danneggiare le normali operazioni della rete, riesce a distruggere o degradare i servizi di routing, creare loop o far cadere link attraverso appunto l'inoltro di pacchetti per percorsi non ottimi o attraverso lo scarto selettivo di alcuni pacchetti. Questa tipologia di attacco è difficile da scoprire perchè dal punto di vista dei nodi la rete sembra comportarsi in maniera corretta. Il Byzantine Attack può essere realizzato secondo le modalità descritte nei seguenti paragrafi.

2.1.1 Link Spoofing Attack

Lo scopo in questo caso è appunto quello di distruggere le operazioni di routing e per far ciò il nodo malevolo annuncia un falso link con un nodo che non è un vicino. Questo attacco nel caso in cui si utilizza il protocollo OLSR con-

siste nell'annuncio, da parte del nodo malevolo, di un link con un nodo vicino a due hop della vittima. Così facendo si provoca l'assegnazione del titolo di MPR all'attacker. A questo punto l'attacker, essendo un MPR, può manipolare o scartare il traffico portando a termine un attacco di tipo DoS (Denial of Service).

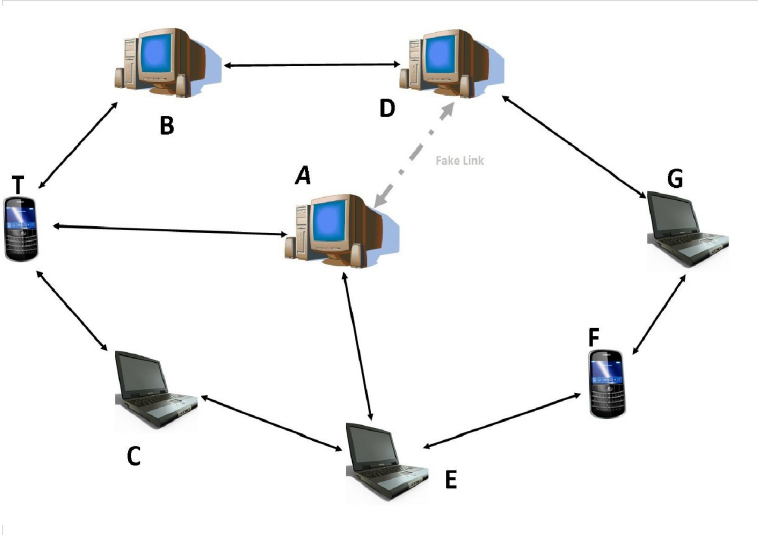


Figura 2.1: Topologia Link Spoofing Attack

Nel caso di protocollo OLSR (vedi Fig. 2.1). Il nodo A è l'attacker mentre il nodo T è la vittima. Prima dell'attacco per il nodo T gli MPRs sono i nodi A e B. Durante l'attacco, il nodo malevolo annuncia un link con il nodo D che dista due hops dalla vittima, quindi, in accordo con OLSR, A diventa l'unico MPR perchè con soli due hops riesce a raggiungere tutti i vicini. Fatto ciò l'attacker può operare sui pacchetti inviati dalla vittima e decidere se inoltrarli o meno [6].

2.1.2 Routing Loop Attack

Questo attacco è realizzabile solo contro le reti che utilizzano un Reactive Protocol. Un nodo malevolo attraverso la modifica dei pacchetti che vengono inviati durante la fase di route discovery riesce a creare un loop facendo si che il traffico giri indefinitivamente tra i nodi intermedi senza che appunto questo arrivi a destinazione. In particolare l'attacker riesce a farsi includere nel percorso e a dirigere il traffico in modo tale da formare un loop se si trova vicino al percorso tra sorgente e destinazione ed è vicino di due nodi successivi appartenenti alla route.

Analizziamo la Figura 2.2 e vediamo un chiaro esempio di Routing Loop Attack nel caso in cui si utilizzi il protocollo AODV.

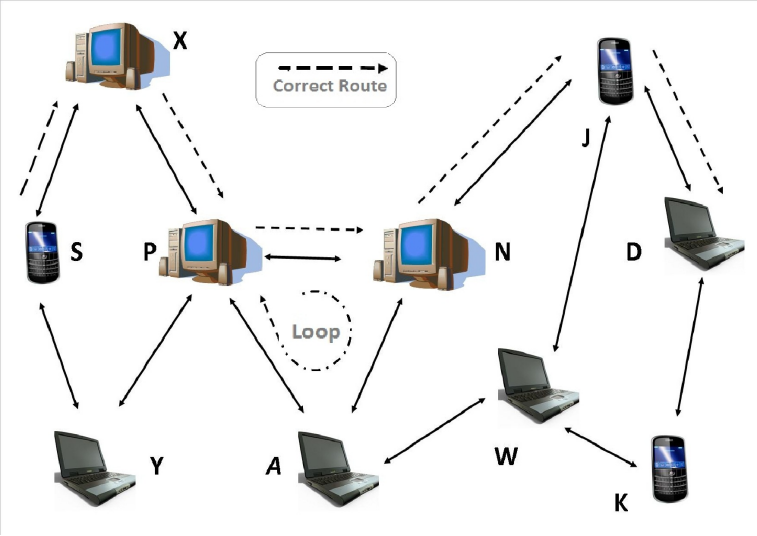


Figura 2.2: Topologia Routing Loop Attack

Ipotizzando che S sia la sorgente, D la destinazione e A il nodo malevolo, possiamo considerare come percorso corretto il seguente S-X-P-N-J-D. Il nodo A per prima cosa

aggiunge una route statica facendo si che, affinchè si possa raggiungere la destinazione, si debba passare da P. Dopodichè invia una RREP, con un Sequence Number maggiore di quello utilizzato nelle RREPs non corrotte, a N. In tal modo avviene che, quando il nodo P riceve un pacchetto da parte di S, lo inoltra correttamente a N, il quale però a sua volta lo invia ad A poichè crede che il miglior percorso per raggiungere la destinazione sia appunto quello che passa dall'attacker; in seguito A inoltra nuovamente il pacchetto in questione a P facendo si che il giro inizi nuovamente. Alla fine di tutto ciò si verifica la creazione di un loop tra S e D il quale porta allo scarto dei pacchetti una volta che il TTL di questi raggiunge il valore zero.

Lo stesso attacco può essere implementato anche in una topologia in cui l'attacker non è vicino al nodo designato come bersaglio. Come prima cosa l'attacker cerca un nodo vittima che si trova nelle vicinanze del path in questione e lo forza ad aggiornare la sua tabella di routing con la route statica precedentemente descritta. Anche in questo caso, affinchè N invii alla vittima i pacchetti destinati a D creando il loop, l'attacker deve inviare una RREP falsa.

2.1.3 Packet Dropping Attack

Il Packet Dropping Attack consiste nello scartare pacchetti indistintamente dalla loro tipologia, quindi indistintamente dal fatto che si tratti di pacchetti dati o di routing. Tipicamente l'attacker, durante la fase di route discovery, inizia col comportarsi onestamente, facendo si che esso stesso possa esser incluso come nodo intermedio nel percorso in questione, per poi successivamente scartare i pacchetti che da sorgente dovrebbero arrivare a destinazione. Lo scarto dei pacchetti può avvenire:

- Constantemente.
- In maniera Random.
- Periodicamente.

In particolare attraverso lo scarto periodico l'attacker riesce a tener nascosto il proprio comportamento scorretto [5]. Altro modo per raggiungere lo stesso risultato è quello di non cooperare con gli altri nodi della rete; infatti il nodo malevolo (attraverso la mancata collaborazione, il non inoltro dei pacchetti, il non invio dei messaggi di

errore o addirittura attraverso il suo spegnimento) riesce ad interrompere le normali operazioni delle rete rendendo inoltre indisponibili alcuni servizi. A volte questo tipo di attacco può essere compiuto anche da nodi che scartano pacchetti senza avere cattive intenzioni, ma che hanno come obiettivo quello di salvaguardare le proprie risorse [7].

2.1.4 Message Modification Attack

Pur di compromettere l'integrità dei pacchetti in rete, gli attackers possono anche apportare alcuni cambiamenti ai messaggi di routing. Come conseguenza del fatto che i nodi sono liberi di muoversi e di auto-organizzarsi, facendo si che il nodo malevolo possa essere, in totale tranquillità, incluso nei collegamenti, si ha che quest'ultimo può decidere se comportarsi correttamente partecipando quindi ai processi di packet forwarding oppure, decidere di lanciare l'attacco modificando alcuni campi del pacchetti. Se per esempio si prende in considerazione il protocollo AODV l'attacco può avvenire attraverso l'incremento dell'id di una vecchia RREQ rendendola valida, oppure abbassan-

do il valore del campo hop count per aggiornare la tabelle di routing degli altri nodi; così facendo è possibile anche ridirigere il traffico dal path originale verso un percorso più lungo e quindi non ottimo.[8]

2.1.5 Message Fabrication Attack

In questo caso, piuttosto che modificare o bloccare i pacchetti di routing già esistenti nella rete, l'attacker preferisce costruirne nuovi in modo tale da creare caos nelle operazioni della rete [5].

Nel caso specifico di protocolli proattivi l'attacker può sia annunciare link inesistenti attraverso l'invio di falsi pacchetti di HELLO che diffondere informazioni di topologia sbagliate attraverso la creazione di TC messages non veri. Nel caso di protocolli reattivi invece i falsi messaggi possono essere quelli di route error; inviando questi messaggi l'attacker fa credere agli altri nodi della rete che uno specifico collegamento è down isolando così il nodo che lo usa.

2.2 Replay Attack

Il Replay Attack, come dice il nome stesso, consiste nella trasmissione ripetuta in maniera fraudolenta di dati validi; ovviamente questo attacco è compiuto da un nodo che fa parte della rete; questo intercetta i pacchetti e li ritrasmette con l'intenzione o di deviare il traffico, evitando magari di utilizzare il path migliore, oppure di avviare un attacco di tipo Denial of Service [5]. In questo caso per evitare di esser scoperto il nodo può replicare tutti i pacchetti tranne quelli di controllo dell'area stessa in cui li ha sniffati. Solitamente per deviare questo attacco, identificando così anche le informazioni più recenti, i protocolli di routing utilizzano i sequence number. E' possibile portare a compimento questo attacco in un'altra area senza essere scoperti. Infatti in quest'ultima non ci sono comunicazioni precedenti con l'originator del pacchetto e di conseguenza non ci sono messaggi con cui poter confrontare il sequence number. Nel caso di protocolli Proactive, possono essere replicati messaggi di Topology Control per reintrodurre vecchi link o per rimuoverne uno esistente; però, affinchè

ciò possa esser fatto, l'attacker deve fare attenzione al fatto che i TC messages sono forniti di un sequence number il quale permette a chi riceve i pacchetti di valutare la 'freschezza' dell'informazione. Considerando il fatto che questo attacco è più efficace nel momento in cui vengono usati pacchetti che non sono propagati per più di un hop, si può valutare il caso in cui ad essere replicati sono i messaggi di HELLO; in questo caso l'attacco è limitato nel suo effetto perchè un Hello Packet contiene una lista dei vicini e il nodo malevolo dovrà spostarsi in un'altra MANET per far si che l'attacco vada a buon fine [9]. Quindi l'attaccante inoltra una RREQ in un'altra area; se i nodi di questa area hanno informazioni più recenti il pacchetto viene scartato, se invece non le hanno viene preso in considerazione e si da inizio ad un attacco di tipo Denial of Service poichè i nuovi pacchetti iniettati provocano traffico non necessario di route discovery [10]. Il nodo può comportarsi anche diversamente, infatti può sniffare e salvare, per poi replicarlo in un secondo momento, un messaggio di RERR, poichè su questi messaggi non viene fatto alcun controllo sulla 'freschezza'. Come conseguenza di questo attacco, i no-

di credono che una route non sia più valida e potrebbero consumare risorse per scoprire un ulteriore percorso che, con alta probabilità, alla fine si rivela esser uguale a quello che era stato scelto prima dell'attacco. Nel caso di protocolli proattivi invece esiste un meccanismo esplicito di link recovery, limitandosi quindi a non annunciare più il nodo vicino; questo permette ai protocolli Proactive di difendersi da questo particolare attacco, anche se rimangono vulnerabili al replay attack con altri pacchetti [9].

2.3 Neighbor Attack

L'attacker ha come obiettivo quello di sconvolgere il routing e per fare ciò fa credere a due nodi, che in realtà non sono connessi e non sono neanche nel range di comunicazione, di essere connessi direttamente l'un all'altro. Conseguenza di questa modifica apportata alla rete è appunto quella di far perdere i pacchetti che teoricamente dovrebbero arrivare a destinazione. Neighbor attack, come si vedrà successivamente, è simile al Blackhole attack nel senso che impedisce la consegna dei pacchetti a destina-

zione; però in questo caso, una volta che viene modificato il link, non è obbligatorio che il neighbor partecipi attivamente all'operazione di scarto di pacchetti [5]. Affinchè il nodo malevolo riesca a far credere a due nodi di esser direttamente connessi, si deve trovare tra le due vittime e mandare i messaggi ricevuti da uno all'altro e viceversa; i messaggi vengono consegnati senza modifiche, causando così una cattiva percezione della topologia della rete.

Prendendo per esempio una rete con topologia come quella in Figura 2.3 le due vittime sono V1 e V2 mentre A è l'attacker; la topologia che i nodi vittima vedranno subito dopo l'attacco è quella in Figura 2.4.

Infine si può ottenere un risultato simile anche facendo credere solamente ad uno dei due nodi di trovarsi vicino ad un altro, ma non il viceversa. Ovviamente, affinchè tutto ciò sia possibile, anche in questo caso il nodo malevolo deve trovarsi direttamente connesso alla vittima.

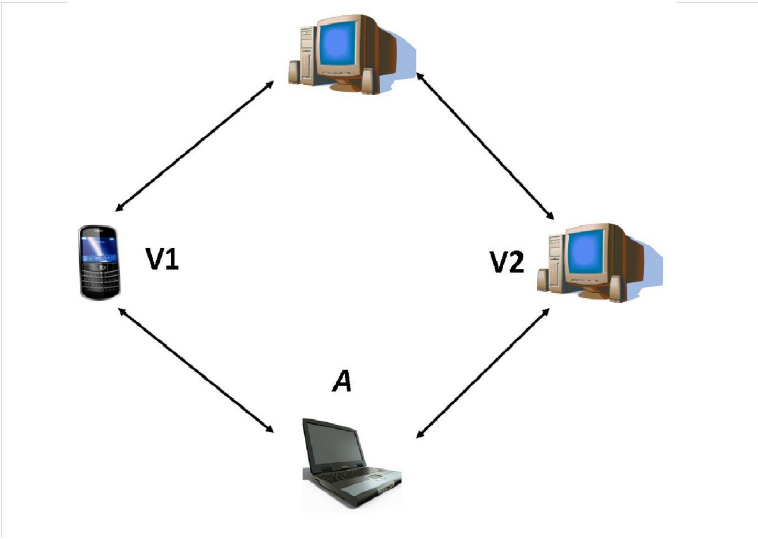


Figura 2.3: Topologia rete prima di un Neighbor Attack

2.4 Blackhole Attack

Il Blackhole Attack consta di due fasi, nella prima il nodo malevolo usa i messaggi del protocollo di routing per fare link spoofing e annunciare se stesso come nodo che gode del percorso migliore e più corto per arrivare alla destinazione che poi sarà anche la vittima. Il tutto avviene attraverso

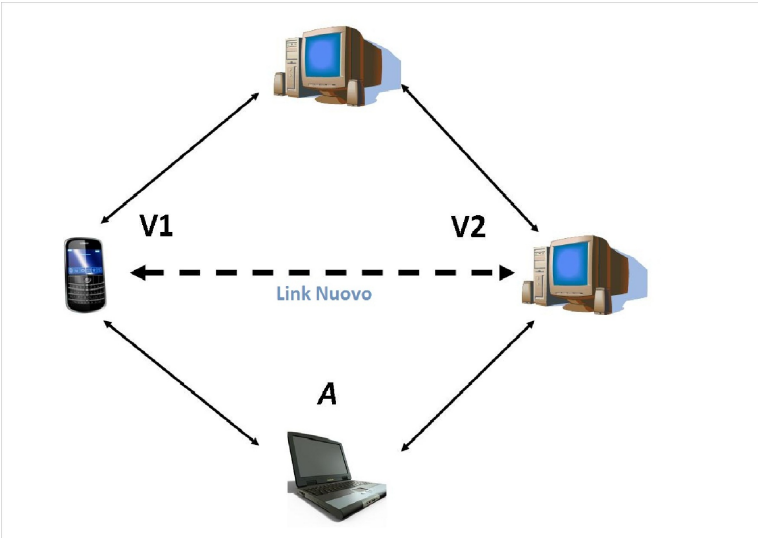


Figura 2.4: Topologia rete dopo un Neighbor Attack

l'inserimento nella tabella di routing di una route valida e più recente. Una volta che il nodo ostile si è inserito tra i due nodi che stanno comunicando, può dar inizio alla seconda fase dell'attacco passando quindi dall'inoltro allo scarto dei pacchetti che lo attraversano. Anche se molto remota, data la scarsa disponibilità di risorse, vi è la

possibilità che i neighbors attraverso un'accurata analisi della rete e un monitoraggio del traffico riescano a scoprire l'attacker e a denunciarlo. Per cercare di evitare che ciò si verifichi esiste una forma un pò più ingegnosa dello stesso attacco; questa non consiste nello scartare tutti i pacchetti, ma solo alcuni di essi oppure modificare i pacchetti provenienti da alcuni nodi evitando così di destare sospetti.

Facciamo un semplice esempio esplicativo.

Ipotizzando che nella rete mostrata in Figura 2.5 i nodi comunicano seguendo le regole del protocollo AODV, consideriamo il nodo A come l'attacker, mentre S e D siano rispettivamente i nodi sorgente e destinazione. L'attacker si pone l'obiettivo di far saltare la comunicazione tra S e D. Come descritto pocanzi, nella prima parte dell'attacco A cerca di far passare il traffico da esso; per fare ciò, quando S inizia il processo di route discovery, manda immediatamente una RREP fasulla, anche se non ha una route per D; attraverso questo messaggio il nodo malevolo annuncia che dista un salto da D, ponendo il campo Hop Count ad 1, e modifica il Destination Sequence Number incremen-

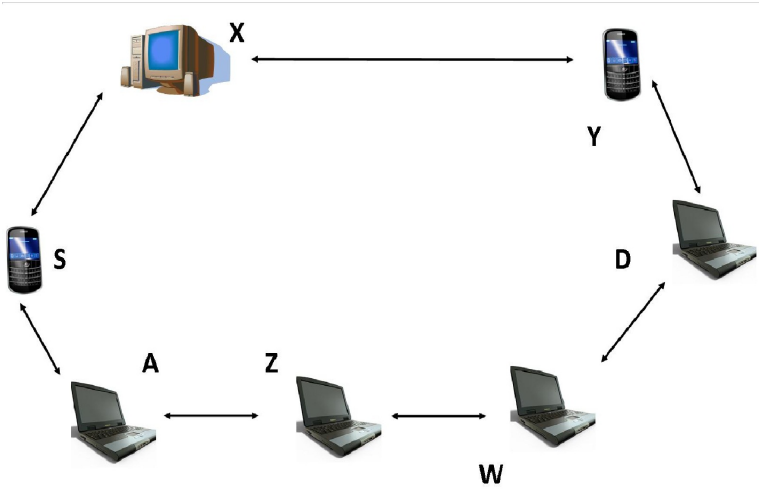


Figura 2.5: Topologia Blackhole Attack

tandolo di 1 in modo tale da assicurarsi che la sua risposta venga considerata la più recente. Così facendo accade che la RREP reale che arriva al nodo X viene scartata poichè il Destination Sequence Number è inferiore e il numero di salti per arrivare a destinazione è maggiore. Anche se A e X dovessero annunciare la stessa distanza, viene scelta la

prima route poichè con sequence number maggiore.

2.5 Grayhole Attack

Anche questo attacco consiste in due fasi, nella prima fase, come per il Blackhole Attack, l'attacker, annunciando una route migliore passante proprio da lui, cerca di farsi includere nel path che da sorgente arriva a destinazione; una volta intercettato il traffico proveniente dalla vittima, l'attacker da inizio alla seconda fase e quindi decide come trattare il traffico in questione. L'attacker può:

- Scartare i pacchetti intercettati con una certa probabilità.
- Scartare i pacchetti per intervalli di tempo random.
- Unire i due comportamente elencati sopra.

Grazie al suo modo di operare, un nodo malevolo che esegue un Grayhole Attack è meno 'esposto' rispetto ad uno che esegue un Blackhole Attack perchè in questo caso non si opera su tutti i pacchetti, ma solo su alcuni ed in periodi random.

2.6 Jellyfish Attack

Jellyfish Attack, come i due attacchi precedenti, consta di due fasi: nella prima anche in questo caso l'attacker, annuncia una route migliore, facendo si che il traffico passi da lui per arrivare a destinazione, nella seconda invece agisce sui pacchetti. Gli obiettivi dell'attacker nel Jellyfish sono quelli di:

- Aumentare il ritardo end to end dei pacchetti.
- Portare a zero il Goodput cioè la quantità di dati utili trasferiti per unità di tempo [11].

Continuando con l'analisi del Jellyfish Attack si può notare che i meccanismi utilizzati per portarlo a termine sono più di uno, quali:

1. Reorder Buffer Attack.
2. Delay Variance Attack.
3. Periodic Dropping Attack.

Il primo dei tre meccanismi consiste nell'inoltrare tutti i pacchetti ricevuti, ma con ordine diverso da quello originale; il tutto avviene sfruttando un buffer nel quale prima vengono accodati i pacchetti per poi inoltrarli; non rispettando l'ordine FIFO (First In First Out). Il secondo invece consiste nel trattenere i pacchetti ricevuti per un periodo random, per poi processarli e inoltrarli incrementando quindi la varianza del ritardo. Per esempio in una connessione TCP, questo causa che il traffico sia mandato in burst, aumentando così la possibilità di collisioni e perdite; viene aumentato anche il valore di RTO (Retrasmission Time Out), provocando una stima sbagliata della banda disponibile nei protocolli che si basano sul ritardo dei pacchetti per il controllo della congestione. Infine il Periodic Dropping Attack scarta i pacchetti per un periodo breve, ogni RTO secondi. Ciò causa perdite multiple e quindi il flusso vittima entra in timeout; quando sta per uscire, RTO secondi dopo, l'attacker scarta nuovamente i pacchetti. Affinchè tutto ciò sia possibile necessita che l'attacker conosca quando il flusso entra in timeout per appunto sapere quando effettuare un nuovo scarto.

É difficile distinguere un attacco di tipo Jellyfish dalla congestione o dalle perdite di pacchetti che avvengono normalmente nella rete, per cui è molto complicato rivelarvo [12].

2.7 Wormhole Attack

Fin ora sono stati descritti ed analizzati attacchi nei quali il nodo malevolo era uno solo, nel caso di Wormhole, uno degli attacchi più complicati possibili contro le reti MANET, gli attackers sono due. Infatti il primo dei due, che si trova in un determinato punto della rete, riceve pacchetti per poi mandarli al secondo il quale li inietta nella zona della rete alla quale esso stesso appartiene. Il tutto è possibile solo se si sfrutta il concetto di *tunnel*, cioè un link diretto tra i due nodi malevoli. A render ancora più dannoso il Wormhole attack contribuisce il fatto che il tutto può avvenire anche in presenza di comunicazioni che comunque riescono a garantire *segretezza* e *autenticità* anche se gli attackers non possiedono le chiavi di crittografia [13]. Le modalità utilizzate per stabilire il tunnel sono:

- Sfruttare l'incapsulamento dei pacchetti per stabilirlo a livello logico.
- Attraverso un link cablato.
- Attraverso un out-of-band long-range wireless link.

Un Wormhole tunnel creato usando l'incapsulamento dei pacchetti è più lento, però è facile da stabilire e non richiede particolari capacità hardware o particolari protocolli di routing. Se invece si parla di link cablato o di out-of-band long-range wireless link è più semplice fare in modo che i pacchetti che attraversano il tunnel arrivino a destinazione con una metrica migliore, poichè, i due attackers si trovano a distanza maggiore di quella che copre un normale range di trasmissione di un singolo hop [14]. Inoltre per utilizzare un link, cablato o wireless, privato e garantire alte velocità è richiesto l'utilizzo di un particolare hardware che permetta appunto ai due nodi di comunicare tra loro. Grazie a questo attacco gli attackers possono:

- Sniffare il traffico.
- Scartare alcuni o tutti i pacchetti.

- Effettuare attacchi di tipo 'Man-in-the-Middle'.

Vediamo quindi alcune modalità, distinguendo il caso in cui si utilizza un protocollo di tipo reattivo da quello in cui si utilizza un protocollo proattivo, per la realizzazione del Wormhole Attack.

2.7.1 Wormhole contro protocolli proattivi

Nei protocolli di tipo Proactive un attacco del genere può mirare a colpire la costruzione della topologia, poichè i nodi scambiano periodicamente pacchetti per fare neighbors discovery e per raccogliere informazioni sulla topologia.

Analizziamo la Figura 2.6, ipotizzando che il protocollo utilizzato è l'OLSR e che S e D siano sorgente e destinazione, mentre A1 e A2 siano i due nodi malevoli. Quando il nodo S invia un messaggio di HELLO, A1 lo copia e lo invia attraverso il tunnel a A2, che a sua volta lo inietta nella sua zona di rete di appartenenza. Così facendo il nodo W nel momento in cui riceve il pacchetto di HELLO replicato, crede che S sia un neighbor. Ovviamente la stessa procedura avviene in senso contrario facendo si che S considera

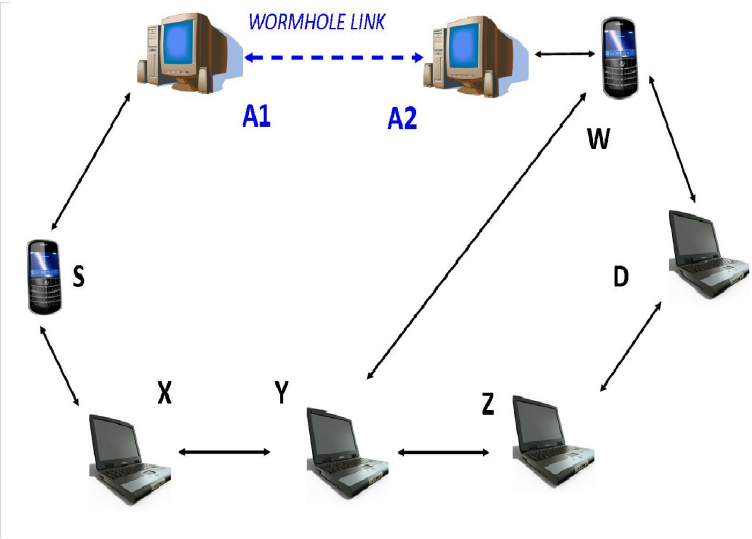


Figura 2.6: Topologia Wormhole Attack con out-of-band channel

W come vicino; in tal modo si è riusciti a creare tra i due nodi un false simmetric link. Inoltre il fatto che questo path sia quello con il minor numero di hops fa si che sia preferito a tutti gli altri. Concludendo, i due nodi possono scegliersi a vicenda come MPRs e scambiarsi anche informazioni in merito alla topologia sfruttando il *Wormhole*

Tunnel, divulgando informazioni sbagliate, distruggendo il routing e degradando le prestazioni dell'intera rete [14].

2.7.2 Wormhole contro protocolli reattivi

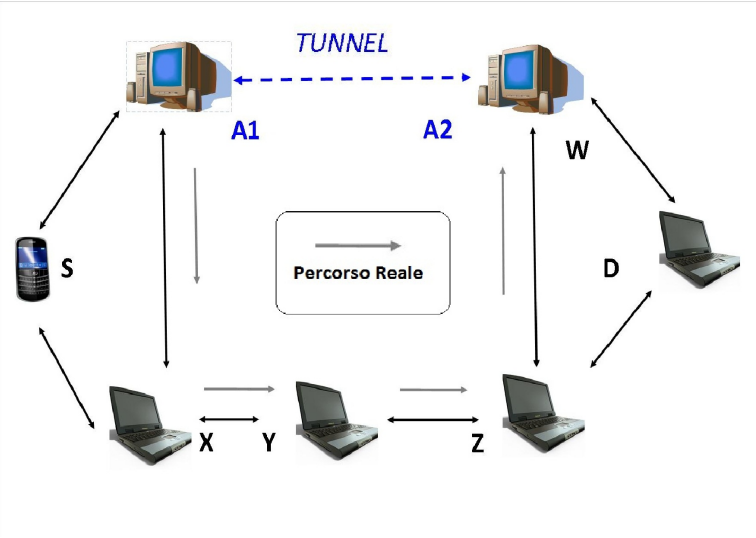


Figura 2.7: Topologia Wormhole Attack con packet encapsulation

Metodo del tunnel logico

Come si vede in Figura 2.7, i compiti di incapsulare i pacchetti e falsificare la lunghezza del percorso spettano ad A1 e A2. Quando S (sorgente) vuole raggiungere D (destinazione) inizia la fase di route discovery. Nel momento in cui A1 si vede arrivare una RREQ da parte di S, la incapsula e la invia verso A2 sfruttando il tunnel. A2, vedendosi arrivare una RREQ dal tunnel, la decapsula e la manda a D mostrando che questa per arrivare a destinazione è passata da A1 e A2, poichè l'header non viene aggiornato dagli attackers. Alla fine a destinazione arrivano due RREQ relative allo stesso processo di route iniziato appunto da S: una da Z con un numero di hops pari a 4, poichè il percorso seguito è S-X-Y-Z-D, mentre l'altra inviata da A2 con un numero di salti pari a 3 poichè il percorso seguito dal pacchetto è stato S-A1-A2-D. Ovviamente per D il percorso migliore è il secondo, e il nodo di destinazione invia una RREP ad A2, che lo invia attraverso il tunnel ad A1 il quale lo inoltrerà alla sorgente. L'effetto finale è appunto quello di far credere a S che il percorso migliore sia quello che passa per A1 e non quello che passa per X. Sfruttando

la creazione del tunnel gli attackers sono riusciti ad evitare che i nodi intermedi e onesti possano incrementare la metrica associata alla lunghezza della route [15].

Metodo del packet encapsulation

Prendiamo come riferimento la Figura 2.6 e consideriamo S e D rispettivamente come i nodi sorgente e destinazione, mentre A1 e A2 come i nodi malevoli. L'attacco ha inizio quando il nodo S attraverso l'invio in broadcast di una RREQ inizia la fase di route discovery. Nel momento in cui il primo dei due attackers A1 si vede arrivare la RREQ, la inoltra ad A2 senza apportare alcuna modifica; A2 a sua volta la consegna inalterata al suo neighbor W, il quale invia il pacchetto, ancora intatto, a D. Dato che la RREQ in questione arriva a destinazione attraverso un canale ad alta velocità e quindi è anche la prima a giungere a D quest'ultimo sceglie la route S-W-D e risponde in unicast. Alla fine il nodo sorgente sceglie questo percorso ignorando il fatto che esso contiene anche i nodi A1 e A2 [6].

2.8 Rushing Attack

Come scritto nel primo capitolo, esistono protocolli di tipo *On Deman* e protocolli classificabili come *Table-Driven*; l'attacco in questione agisce solo contro i protocolli facenti parte della prima classe, in particolare quelli che riescono a sopprimere i duplicati durante la fase di route discovery. Un punto a favore del Rushing Attack è dato dalla difficoltà nell'esser scoperto.

Vediamo come avviene l'attacco: ogni nodo, affinchè si possa limitare l'overhead del flooding, inoltra solo una RREQ per ogni route discovery. Nel momento in cui l'attacker riceve una RREQ, la inoltra velocemente in rete ancor prima che gli altri nodi che hanno ricevuto lo stesso messaggio possano rispondere. Un nodo che riceve il pacchetto di RREQ leggittimo lo considera come un duplicato di quello precedentemente ricevuto dall'attacker e lo scarta. In tal modo il nodo malevolo è riuscito a farsi inserire nella route da sorgente a destinazione; inoltre non sarà possibile da parte della sorgente trovare altre route che non lo contengano [15]. Un attacco del genere non

richiede una grande quantità di risorse. In condizioni normali ne consegue che il forwarding delle RREQ è ritardato, infatti se a livello MAC si ha time division, i nodi devono aspettare il proprio time slot per trasmettere; oppure, nel caso in cui non viene specificato il ritardo di MAC; a livello di routing quando si hanno dei pacchetti di broadcast vengono imposti dei ritardi random poichè grazie al settaggio di questi è difficile rilevare le collisioni. Da parte dell'attacker diventa quindi possibile rimuovere questi attacchi per appunto velocizzare l'instradamento. Il forwarding può essere velocizzato, rispetto agli altri nodi, tenendo le interfacce degli altri partecipanti alla rete impegnate, per esempio mandando false RREQ così da rallentare il processing e il forwarding della RREQ legittima. Inoltre è possibile trasmettere anche i pacchetti di RREQ lavorando sulla potenza poichè trasmettere ad un livello di potenza maggiore implica la riduzione del numero di hop che i messaggi devono attraversare per arrivare a destinazione [16].

Vediamo un esempio pratico (Fig. 2.8):

S inizia il processo di route discovery per raggiunge-

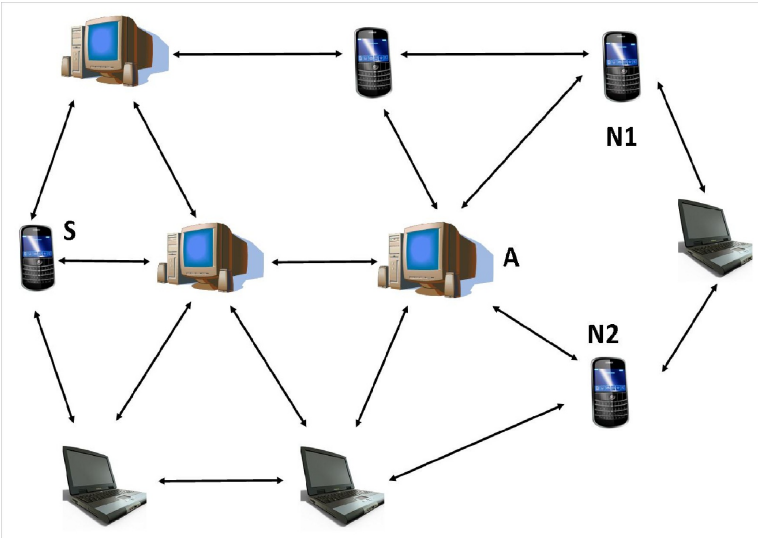


Figura 2.8: Topologia Rushing Attack

re la destinazione D; se la RREQ inoltrata dall'attacker A raggiunge per prima tutti i nodi vicini alla vittima D, allora ciascuna route scoperta includerà un salto verso il nodo malevolo. Quando i neighbors della destinazione (N1 e N2) ricevono la richiesta corrotta la inoltrano senza inoltrare alcun'altra RREQ, poichè, come descritto sopra, si desi-

dera limitare l'overhead. Quando arrivano in un secondo momento le altre RREQ inoltrate dai nodi non malevoli, queste verranno scartate. Come risultato di tutto ciò si ha che la sorgente non riuscirà a trovare altre route migliori che non contengano il nodo malevolo.

2.9 Sybil Attack

Affinchè l'attacco vada a buon fine il nodo malevolo impersona un nodo:

- Fisicamente presente nella rete.
- Del tutto inesistente.

  Nel primo caso l'hacker assume le sembianze di un nodo che è realmente esistente e che realmente appartiene alla rete; così facendo il nodo malevolo riesce ad intercettare i messaggi che sono diretti al nodo vittima. Usando questo metodo l'attacker non solo maschera la propria identità, ma, fingendosi il nodo attaccato, può decidere come rispondere ai messaggi di routing o comportarsi in maniera malevola. Nel

secondo caso invece, affinchè vengono compromessi i vari servizi della rete, l'attacker impersona uno o più nodi.[5] Con questo atteggiamento non solo vengano a saltare i servizi quando appunto è necessaria la cooperazione tra i nodi della rete, ma si colpiscono anche gli schemi di configurazione che si basano prettamente su un modello di fiducia.

L'attacco di tipo man-in-the-middle per esempio vede l'attacker assumere l'identità di un nodo appartenente alla rete. Questo ha la capacità di leggere e modificare i messaggi scambiati tra due nodi; S e D stanno comunicando tra loro, l'hacker può impersonare S nei confronti di D e viceversa [15].

Volendo fare invece un esempio pratico relativo al caso in cui l'attacker impersona uno o più nodi inesistenti, come si può notare in Figura 2.9 X è connesso sia a Y che a Z ma anche al nodo malevolo A. Nel momento in cui A decide di rappresentare anche A1, A2, A3, fa credere a X di avere sei vicini piuttosto che tre con la conseguenza di aumentare la possibilità che i vari percorsi all'interno della rete

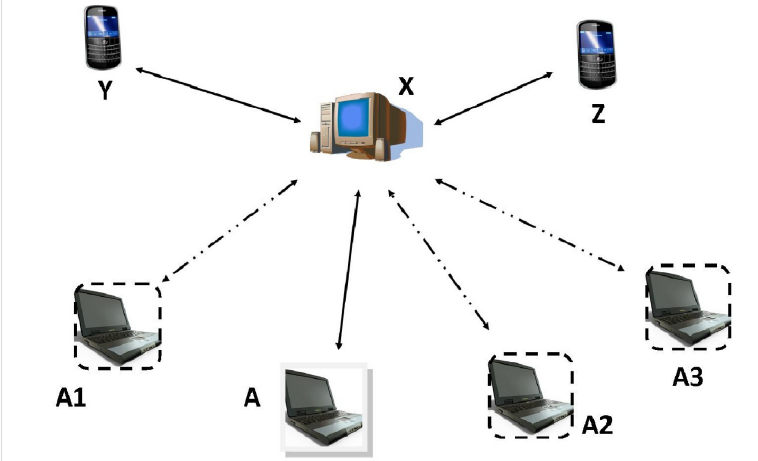


Figura 2.9: Topologia Rushing Attack

contengano l'attacker [8].

2.10 Resourse Consumption Attack

Il fine unico di questo tipo di attacco è appunto quello di consumare e sprecare le risorse di uno o più no-

di [1]; Il Resourse Consumption Attack può avvenire attraverso:

- la formulazione di continue richieste per vari percorsi non necessari
- frequente creazione di pacchetti di beacon
- inoltro di pacchetti vecchi o obsoleti

Affinchè la vittima sia continuamente occupata si può lavorare sia a livello MAC che a livello di routing [15].

2.10.1 Routing Table Overflow

Lo scopo dell'attacker nel momento in cui decide di portare avanti un Routing Table Overflow è quello di consumare le risorse della memoria dedicate alla tabella di routing del nodo vittima. In questo caso vengono create delle route per nodi inesistenti. Il

[1] per risorse di un nodo si intendono: carica della batteria, banda disponibile, capacità computazionali che come noto nei devices che fanno parte delle reti MANET sono abbastanza limitate.

fine è quindi quello di crearne una quantità tale da far si che nella routing table non possa esser inserita nessun'altra entry corrispondente a una route valida o anche confondere l'algoritmo di routing. [15]

2.10.2 Jamming Attack

Questa tipologia di attacco avviene a livello MAC e ha come obiettivo quello di interferire con le comunicazioni wireless legittime. Col termine Jammer si indicano dei disturbi creati intenzionalmente, infatti in questo caso l'attacker, affinchè i nodi che desiderano iniziare una comunicazione non trasmettono nulla poichè vedono il canale occupato, non fa altro che emettere un segnale radio che rappresenta dei bit random o dei pacchetti che magari sono costituiti da un header corretto e un payload inutile e quindi semi-validi. L'invio di questi pacchetti può avvenire secondo tre tipologie:

1. constantemente
2. random

3. quando ci sono pacchetti in rete

in particolare la terza tipologia fa si che in rete si venga a creare rumore che disturbando la comunicazione costringe i nodi a ritrasmettere [17].

2.10.3 Sleep Deprivation Attack

Affinchè si possa raggiungere l'obiettivo di consumare le risorse, l'attacker non fa altro che tenere i nodi constantemente impegnati a processare pacchetti del tutto inutili. L'idea base è quella di richiedere un servizio che il nodo offre più e più volte in modo che questo non possa entrare nello stato idle o di power preserving, deprivando così dal *'sonno'* [5]. Affinchè il nodo vittima diventi incapace di partecipare ai meccanismi di routing e conseguentemente diventi irrangiungibile dagli altri nodi della MANET l'attacker per esempio può 'spammare' il nodo in questione con messaggi di RREQ piuttosto che di RREP o di RERR. Alternativamente l'hacker, affinchè venga utilizzata ulteriore banda e vengano consumate le ri-

sorse della batteria, può sempre replicare pacchetti vecchi reimmettendoli in rete [15].

Capitolo 3

Test-bed

Dopo aver descritto ed analizzato le reti MANET ed i protocolli implementati per far si che i nodi possano dialogare tra di loro e scambiare dati nel miglior modo possibile, è giunto il momento di descrivere il Test-bed. Questo capitolo, come già detto nell'introduzione, descrive come è stato realizzato il Test-bed e gli strumenti utilizzati per implementare sia gli attacchi precedentemente descritti, sia una versione sicura del protocollo OLSR, verrà anche presentato

un modo per implementare una prima linea di difesa nelle reti in questione.

3.1 La rete MANET

3.1.1 Realizzare e configurare una rete ad-hoc

Nel caso in questione il Test-bed è stato realizzato attraverso l'utilizzo di 7 laptop (vedi Fig. 3.1, Fig. 3.2). Su ognuno di essi è stato installato il sistema operativo LINUX ed in particolare la release 10.4 LTE di Ubuntu. Per realizzare una rete ad-hoc, necessita avere degli adattatori senza fili; pertanto su ogni calcolatore è stato necessario installare un'opportuna scheda di rete Wifi (per le procedure di installazione basta seguire le istruzioni presenti nel manuale presente a fine testo). Dopo l'installazione si è passato alla configurazione della rete; per far ciò è stato creato un opportuno script chiamato **'test.sh'**

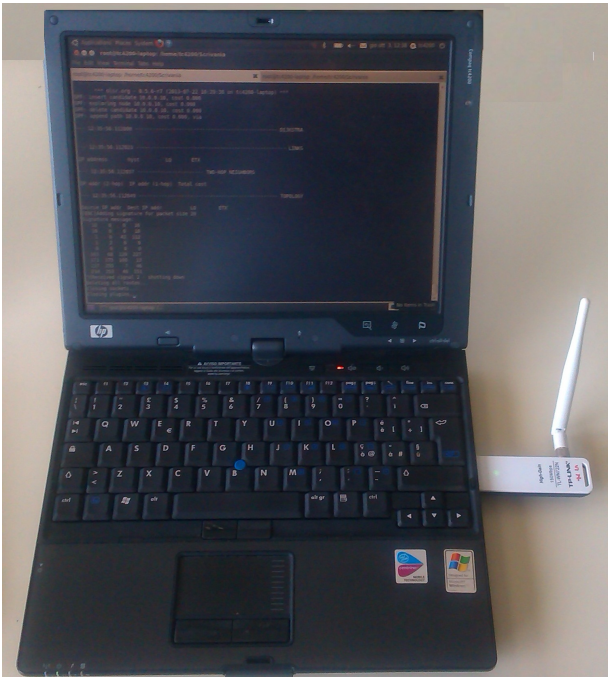


Figura 3.1: Laptop utilizzato

all'interno del quale sono stati inseriti gli opportuni comandi:

```
# ifconfig INTERFACCIA down

# service network-manager stop

# iw INTERFACCIA set type ibss

# ifconfig INTERFACCIA INDIRIZZO IP up

# iw INTERFACCIA ibss join test 2412
```

Questi comandi vengono utilizzati per disabilitare (vedi seconda riga) il tool di *Network Manager* poichè se attivo rivela tutte le reti disponibili e fa si che il sistema possa autoconfigurarsi [11]; viene quindi impostata la modalità ad-hoc, assegnato un indirizzo ip all'interfaccia (nel nostro caso si sono sfruttati indirizzi di classe A) e riattivato il tutto. I comandi presenti nell'ultima riga di codice sono stati utilizzati per sintonizzare la schede di rete alla frequenza

2.412 GHz (Banda ISM solitamente usata dalle reti wireless [11]). Nel caso reale non tutti i nodi sono direttamente connessi e quindi a causa delle distanze non tutti si trovano in visibilità diretta tra loro; nel nostro caso invece, dato che comunque tutti i laptop si trovano nello stesso laboratorio, affinchè si possa riprodurre una situazione realistica si è utilizzato il tool *Iptables*. Grazie a quest'ultimo è stato possibile ricreare la topologia più opportuna e consona alle nostre esigenze e quindi formare i vari link; facendolo agire a livello MAC è stato possibile impostare lo scarto dei pacchetti provenienti dai nodi che non si vogliono avere in visibilità (in accordo con la topologia desiderata).

Per far ciò il comando utilizzato è stato:

```
# iptables -I INPUT -m mac
--mac-source ADDRESS -j DROP
```

Analizziamolo:

Figura 3.2: Esempio di Topologia

-I INPUT: questa parte del comando serve per indicare che comunque si stanno prendendo in considerazione solamente i pacchetti in ingresso;

-m mac...ADDRESS: indica che si sta lavorando a livello di MAC ed in modo particolare all'indirizzo ADDRESS;

-j DROP: dopo aver selezionato il traffico sul quale agire, bisogna indicare come operare: in que-

sto caso, dato che si è deciso di 'Scartare' dei pacchetti il comando che si utilizza è appunto quello di DROP; in caso contrario avremmo utilizzato l'ACCEPT.

3.1.2 Protocolli di Routing

Come accennato i test sono stati effettuati sfruttando due tipologie di protocolli, Reactive e Proactive; per la prima classe di protocolli si è lavorato con AODV, mentre per la seconda si è sfruttato il protocollo OLSR.

AODV

Di AODV, come per tutti gli altri protocolli, esistono varie release; nel Test-bed in questione è stata utilizzata l'implementazione *aodv-uu-0.9.5* sviluppata da Uppsala University. Per installarla è stato necessario digitare i comandi presenti nel manuale; dopo l'installazione è stato necessario attivarlo su tutte le macchine utilizzando il comando

```
# aodv -l -d -D -r SEC
```

-l : utilizzato per scrivere l'output di questo comando nel file di log: /var/log/aodv.log;

-d : per far girare il protocollo in background;

-D : per non attivare il tempo di attesa come *reboot delay* quando il protocollo fallisce;

-r SEC : per salvare nel file di log la tabella di routing ogni 'SEC' secondi.

OLSR

Anche nel caso del protocollo OLSR prima si è dovuto scaricare il pacchetto per poi installare il tutto; successivamente, affinchè il protocollo possa partire senza alcun intoppo, sono stati settati i parametri presenti nel file di configurazione *olsrd.conf* presente nella cartella *root/.../RELEASESCARICATA/etc*; in particolare si sono settati:

- Debug
- Ipversion
- Interface

Quando si lavora con questo protocollo, è possibile anche, utilizzare le opzioni del comando che fa partire il demone del protocollo, per sovrascrivere il file di configurazione precedentemente menzionato; tra le principali possiamo trovare:

-d : *debug level* indica quante informazioni di debug si desidera visualizzare in output. Esso può assumene un valore compreso tra 0 e 9, dove 0 significa che il protocollo gira in background; nel nostro caso, come vedremo successivamente, si è utilizzato il 2.

-i : *interface* specifica l'interfaccia della scheda di rete.

-ipv6 : utilizzato per indicare al protocollo che si deve utilizzare IPv6 e non, come farebbe di default, IPv4.

-bcast : *broadcast address* permette di decidere quale indirizzo broadcast utilizzare per i messaggi di controllo.

-hint : *seconds* utilizzato per settare il valore di HELLO_INTERVAL.

Un attacker potrebbe operare anche attraverso la modifica di alcuni parametri presenti nel file di configurazione; egli potrebbe modificare sia il *Willingness* (mettendolo per esempio a 7 per farsi scegliere come MPR quando normalmente è 3) che il *HELLO_INTERVAL* (e mandare messaggi di Hello molto più spesso e impegnare continuamente la vittima).

OLSR e un primo livello di sucurezza

Come già evidenziato, OLSR è un protocollo di tipo 'Table-Drive' o proattivo e non è capace di fornire alcun tipo di sicurezza. Il testbed in questione è stato realizzato in un primo momento per studiare e valutare quelle che so-

no le debolezze, dal punto di vista della sicurezza informatica, delle MANET e dei relativi protocolli; successivamente invece la rete creata in laboratorio è stata utilizzata per realizzare un sistema che la proteggesse dalle minacce precedentemente implementate.

Il punto di partenza per questa seconda fase è stato l'installazione di una *'OLSR plugin Library'*; nel caso di OLSR viene definito come plugin una dinamic linked library (DLL), cioè un pezzo di codice eseguibile che contiene funzioni e dati. Le DLLs di solito sono librerie o funzioni condivise da più processi e vengono utilizzate per aggiungere un'interfaccia tra il demone del protocollo e altre applicazioni. Grazie all'utilizzo dei plugins è possibile ottenere servizi di Network come il DNS, ma questi possono anche contribuire ad ottimizzare per esempio le funzionalità dei nodi MPR, riducendo così il carico di rete. É stato utilizzato come punto di partenza un plugin perchè grazie ad esso è

stato possibile aggiungere una nuova funzionalità senza alterare quelle già presenti; inoltre, essendo scritto in linguaggio 'C', utilizzabile e compatibile con l'ambiente linux, è stato anche possibile, come si vedrà nel capitolo successivo, apportare delle modifiche utili per rendere il più possibile sicuro OLSR.

Il plugin utilizzato è **olsrd_secure.so.0.5**, dove '0.5' indica che è stato implementato per una specifica versione '0.5.6' del protocollo OLSR mentre '.so' è l'estensione tipica dei plugins [18].

Prima di installare il plugin è stato necessario installare la release **olsrd-0.5.6**; dato che potrebbe esser stata installata nelle varie macchine una versione differente del medesimo protocollo, è ragionevole effettuare un'operazione di cleaner attraverso i comandi

```
# make clean && make && make install
```

successivamente si è passati all'installazione del plugin, per far ciò è bastato digitare

```
# cd /olsrd-0.5.6/lib/secure
# make && make install
```

A differenza di OLSR (per il quale era possibile settare i parametri anche attraverso opportuni comandi), utilizzabili quando si fa partire il demone del protocollo, in questo caso si è dovuto editare direttamente il file di configurazione 'olsr.conf'; in particolare, oltre a settare interfaccia, level debug ed altri parametri per attivare e far si che il protocollo potesse riconoscere il plugin, è stato necessario aggiungere in fondo al file le seguenti righe:

LoadPlugin 'olsrd_secure.so.0.5' {
PlParam 'keyfile' '/etc/olsrd.d/olsrd_secure_key'
}

Leggendo le poche righe da aggiungere al file è facile imbattersi nella parola 'Key': infatti è richiesto il settaggio di una chiave.

Nei capitoli precedenti, in particolare nel secondo dove sono stati descritti i vari attacchi, si è detto che un attacker solitamente è un nodo che non ha alcun interesse nel partecipare alla MANET se non quello di recare danni; detto ciò è facile intuire che una prima linea di difesa può essere implementata limitando l'accesso alla rete ad alcuni host; gli sviluppatori del plugin sono riusciti a far ciò imponendo l'utilizzo di una chiave da parte di tutti i nodi che desiderano partecipare alle operazioni, di routing o di scambio dati, tipiche delle MANET. Nel test-bed ogni nodo è stato dotato di una chiave preinstallata di 128 bit. Nel momento in cui, attraverso la digitazione del comando

```
# olsrd -d 2
```

viene fatto partire il demone del protocollo OLSR, viene verificata la disponibilità da parte del nodo di una chiave che abbia la lunghezza richiesta. Inoltre, utilizzando questo plugin che a sua volta utilizza la crittografia simmetrica, gli

sviluppatori hanno fatto si che nella rete venga garantita l'integrità dei dati. La conferma di quanto detto è il fatto che un nodo che non ha accesso alla chiave condivisa non può produrre un 'digest' verificabile e non può quindi entrare a far parte della rete.

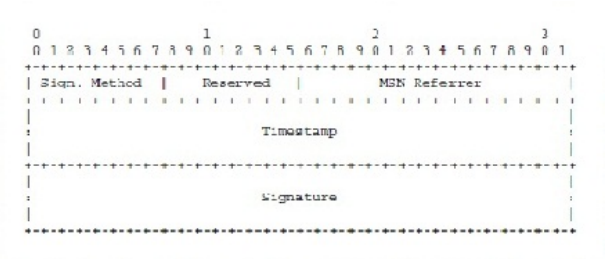


Figura 3.3: Struttura messaggio con firma OLSR

Concludendo possiamo dire che questa 'Secure extension to the OLSR protocol' [19] prevede l'utilizzo di una chiave condivisa per creare una prima linea di difesa attraverso la creazione e la verifica della firma partendo dal presupposto che solo i nodi onesti possano accedere alla rete. In Figura 3.3 è possibile osservare la struttura

del messaggio contenente la firma.

3.1.3 Sniffing del traffico

Per effettuare lo sniffing del traffico, per poi estrarre le informazioni utili per la detection di alcuni attacchi, è stata utilizzata la libreria **Libpcap-1.1.1** anche essa, come il plugin precedente presentato, scritta in C.

Affinchè il suo utilizzo sia stato possibile è stato necessario come prima cosa specificare per poi configurare l'interfaccia di rete da usare:

```
pcap_t *handle;
handle = pcap_open_live
(dev, BUFSIZ, 1, 1000, errbuf);
if (handle == NULL) {
fprintf( stderr,
 "Couldn't open device %s: %s\n",
dev, errbuf);
return(2);}
```

Attraverso la funzione **'pcap_open_live( )'** è stato possibile quindi:

- settare l'interfaccia per la cattura
- settare il numero massimo di bytes da catturare
- indicare il timeout prima che l'operazione di lettura finisca
- settare le dimensioni di un buffer che viene utilizzato in caso di messaggi di errore

Dato che di default vengono catturati tutti i pacchetti, indistintamente dalla loro provenienza o dalla loro tipologia, affinchè non si consumino troppe risorse è stato creato un filtro attraverso l'utilizzo delle funzioni di **'pcap_compile( )'** e **'pcap_setfilter( )'**

```
struct bpf_program fp;
char filter_exp[] =
"ip src host ADDR && udp port PORT";
```

```
bpf_u_int32 mask;
bpf_u_int32 net;

if (pcap_lookupnet
(dev, &net, &mask, errbuf) == -1) {
fprintf(stderr,
"Can't get netmask for device %s\n", dev);
net = 0;
mask = 0;}

if (pcap_compile
(handle, &fp, filter_exp, 0, net) == -1) {
fprintf(stderr,
"Couldn't parse filter %s: %s\n",filter_exp,
pcap_geterr(handle));
return(2);}

if (pcap_setfilter
(handle, &fp) == -1) {
fprintf(stderr,
"Couldn't install filter %s: %s\n",
```

```
filter_exp, pcap_geterr(handle));
return(2);}
```

Nel codice, 'bpf' (berkeley packet filter) indica appunto il filtro, ADDR indica l'indirizzo della sorgente dei pacchetti, mentre con PORT è stato identificato l'indirizzo di porta dal quale arriva il traffico (nel caso OLSR è 698).

La particolarità di Libpcap è appunto quella di fornire diverse opzioni per la cattura del traffico. Nel test-bed sono state utilizzate anche le funzioni:

- **pcap_next();**
- **pcap_loop();**

La prima delle due è stata utilizzata, nella fase di detection del Replay Attack, per catturare i pacchetti singolarmente, la seconda invece per analizzare il contenuto dei pacchetti sniffati e verificare se essi erano unici oppure sono delle repliche. Il codice che corrisponde alla prima operazione è il seguente:

```
packet=pcap_next(handle, &header);
pcap_sendpacket(handle,packet,header.len);
```

per la seconda operazione è invece questo:

```
if (pcap_loop(handle,-1,
process_packet,NULL)==-1){
fprintf (stderr,"%s",
pcap_geterr(handle));
exit (1);
}
```

Affinchè la cattura dei pacchetti desiderati abbia buon esito è stato necessario anche definire le strutture degli header.

```
/* OLSR header */
struct sniff_olsr {
u_int16_t pkt_len ;
u_int16_t pkt_seqno ;
};
```

```
/* OLSR message header */
struct sniff_msg_header {
u_int8_t msg_type ;
u_int8_t vtime ;
u_int16_t msg_size ;
struct in_addr orig_addr ;
u_int8_t ttl;
u_int8_t hcount ;
u_int16_t msg_seqno ;
};
```

Infine, come si può vedere più in dettaglio nel manuale (Appendice A), per individuare l'inizio del pacchetto è stato utilizzato il seguente codice:

```
ethernet = (struct sniff_ethernet*)(packet);
ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
size_ip = IP_HL(ip)*4;
udp = (struct sniff_udp*)(packet
+ SIZE_ETHERNET + size_ip);
```

mentre per individuare solamente un parametro dell'header (nel nostro caso il Sequence Number del generico pacchetto sniffato) sono state impiegate le seguenti righe di codice:

```
olsr = (struct sniff_olsr*)(packet +
SIZE_ETHERNET + size_ip + size_udp);
```

Capitolo 4

Implementazione

Dopo aver descritto, nel capitolo precedente, la realizzazione del Test-bed e di una prima linea di difesa usando un opportuno plugin passiamo a descrivere come è stata implementata una versione sicura del PROACTIVE PROTOCOL OLSR.

In questo capitolo verranno descritti per ogni tipologia di attacco quelle che sono appunto le rispettive linee di difesa. Bisogna tener pre-

sente che le varie contromisure sono state implementate partendo dall'analisi del comportamento che ha il nodo malevolo in ogni singolo attacco.

4.1 Replay Attack

In un attacco di tipo Replay l'attacker prima sniffa il traffico e successivamente decide quali pacchetti replicare, il tutto viene compiuto grazie all'utilizzo di opportune librerie come per esempio la Libpcap.

La detection di questo attacco è stata aggiunta al protocollo grazie all'installazione del plugin *olsr_secure.so.0.5*. Gli implementatori sono riusciti a far ciò attuando una nuova strategia, di solito tutti i protocolli che fanno le detection di questo attacco partono dal presupposto che se ad un nodo arrivano due pacchetti con lo stesso Sequence Number, allora uno è la replica dell'altro e quindi sta avvenendo un Replay At-

tack. Tuttavia nel protocollo OLSR il numero di sequenza è un parametro 'debole' (solamente 16-bit) e un attacker può prendere un pacchetto replicarlo e iniettarlo in una nuova rete portando ugualmente l'attacco a termine. Per evitare ciò nei pacchetti è stato aggiunto un nuovo parametro, il **Timestamp**.

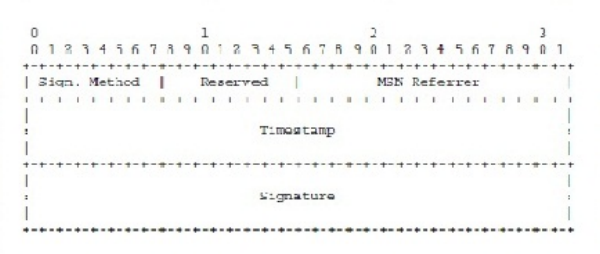


Figura 4.1: Struttura messaggio con firma OLSR

Vediamo quindi come avviene lo scambio del timestamp e come viene fatta la detection.

Il timestamp viene calcolato in maniera del tutto implicita ogni volta che un nodo entra a far parte delle rete e quindi subito dopo aver su-

perato la prima linea di difesa; il tutto avviene attraverso lo scambio di tre appositi messaggi:

- Challenge.
- Challenge - Response.
- Response - Response.

Le figure 4.2, 4.3, 4.4 rappresentano le rispettive strutture dei messaggi sovraelencati.

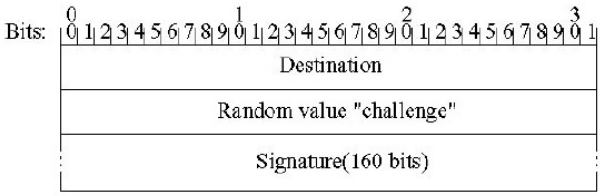


Figura 4.2: Struttura messaggio Challenge

Considerando due nodi A e B, che fanno parte della stessa rete e che si vedono per la prima volta, lo scambio avviene come segue:

A → B : Cha D(IPb,M,K)

B → A: Chb Tsb D(IPb,CHa,K)D(M,K)

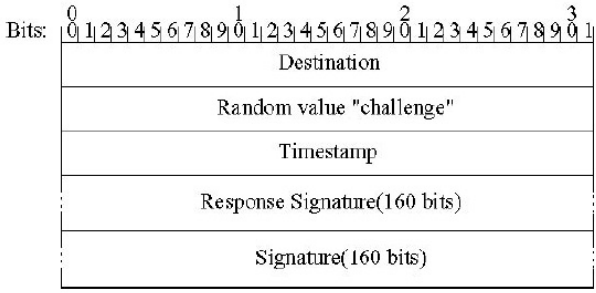


Figura 4.3: Struttura messaggio Challenge - Response

A → B : Tsa D(IPa,CHb,K)D(M,K) .

Quando A riceve, per la prima volta, un messaggio firmato da un vicino B non ha alcun valore temporale registrato, A inizia il processo di scambio del TIMESTAMP.

- A invia un messaggio di **CHALLENGE** a B. Il messaggio contiene l'indirizzo IP di B e un nonce, Cha, di 32 bit random (numero usato una volta). Questo è un numero casuale, che viene utilizzato per accodare dati casuali a dati reali per prevenire un attacco

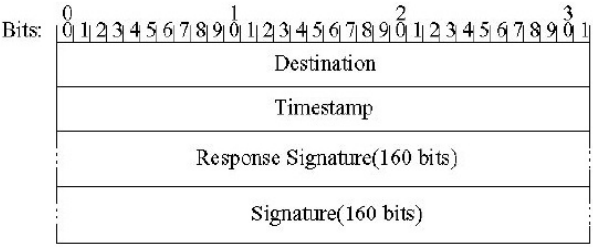


Figura 4.4: Struttura messaggio Response - Response

di replay. A poi 'firma' il messaggio con un digest di tutto il messaggio e usando la key condivisa D(M, K)

- B risponde a questo messaggio con un messaggio **CHALLENGERESPONSE**. B prima genera il digest del suo IP address, del nonce ricevuto e della chiave condivisa D (IPB, cha, K), quindi genera un nonce a 32 bit e trasmette ad A, il nonce, il timestamp di B, il digest D (IPb, CHa, K) e un riassunto di tutto il messaggio e la key condivisa D(M, K).

- Quando A riceve il messaggio di **CHALLENGERESPONSE** da B, cerca prima di convalidare i dati. Se il digest D(IPb, CHa, K) e D(M, K) possono essere verificati, allora il timestamp di B è utilizzato per creare la differenza di tempo tra A e B. A allora genera un messaggio di RESPONSE-RESPONSE e lo trasmette a B. Questo messaggio contiene l'indirizzo IP del ricevitore(B), un timestamp, il nonce ricevuto da B, la chiave condivisa D(M, K) e un digest del messaggio e l'intera key D(M, K).
- Quando B riceve il messaggio da A, si verifica il digest. Se la convalida avviene allora B usa il timestamp ricevuto per registrare la sua differenza di tempo ad A.
  E lo scambio del timestamp é completo.

Una volta che i due nodi hanno scambiato le informazioni in merito al timestamp si può passare alla detection; Considerando **Tsender** (Ts) il timestamp, valore univoco, messo dal sender

nel messaggio e **T** il valore del clock del nodo ricevente, per ogni coppia di nodi si fissa un valore delta che indica la discrepanza del timestamp tra i clock dei due nodi. Se il modulo della differenza tra Ts e T è maggiore o uguale di delta vuol dire che il pacchetto è replicato e viene scartato.

$$|Ts - T| < \delta$$

La conferma del fatto che il Replay Attack non va a buon fine si ha anche da un algoritmo appositamente implementato sfruttando la libreria Libpcap (vedi Fig. 4.5).

```
/* Sniffo due pacchetti */

packet1 = pcap_next( handle, &header);
packet2 = pcap_next( handle, &header);

/* Confronto i pacchetti */
```

```
if(packet1 == packet2){
printf("‘Pacchetto già
 presente nella rete\n"’);
}

if(packet1 != packet2){
printf("‘Diversi!\n"’);
}
```

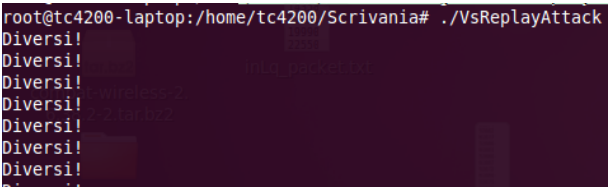


Figura 4.5: Output script VsReplayAttack

4.2 Drop Attack

L’obiettivo di questo attacco è quello di creare confusione nel Routing e nella rete. Il tutto con-

sta nello scarto di alcuni pacchetti attraverso comandi come

```
# iptables -I FORWARD -j DROP -s SOURCE
```

che a sua volta sfrutta il tool *iptables*.

Per fare detection è stato modificato il codice sorgente del protocollo, riadattandolo alle nostre esigenze riuscendo quindi ad analizzare i parametri di nostro interesse dei messaggi che arrivano a destinazione. Come noto che nel protocollo OLSR le informazioni sulla topologia vengono scambiate grazie ai messaggi di TC (Topology Control); nell'attacco in questione la detection viene fatta partendo da un'analisi dei Sequence Number di questi messaggi, considerando una topologia come quella in Figura 4.6:

i TCSeqno che arrivano a destinazione sono alternati tra quelli di A e quelli di S, quindi nel momento in cui l'attacco va a buon fine e 'salta' il link, a destinazione arrivano solamente i

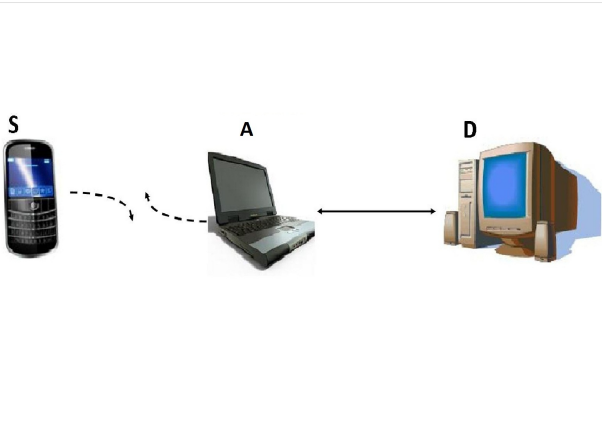


Figura 4.6: Topologia Drop Attack

pacchetti di A. Come si può vedere in Figura 4.7, dopo l'attacco si avranno solo i TCSeqno di A, ovviamente tutti consecutivi. Sfruttando questo concetto si è riusciti a fare detection e nel momento in cui il protocollo si rende conto che qualcosa non va viene stampato un apposito messaggio (vedi Fig. 4.8).

Volendo analizzare il codice aggiunto possiamo dire che sono state utilizzate le funzioni

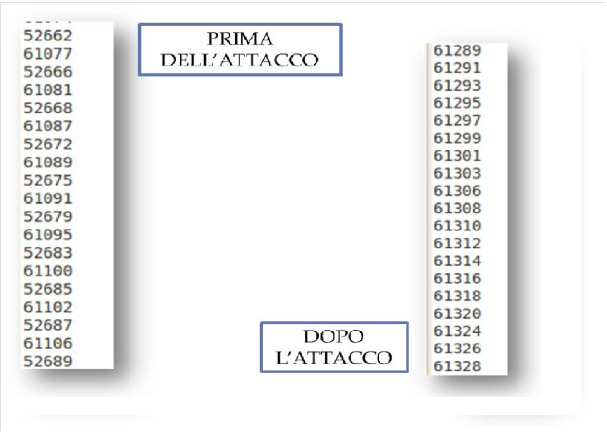


Figura 4.7: Detection Drop Attack

- fopen( );
- fprintf( );
- fscanf( );
- fclose( );

previste dal linguaggio C per la gestione dei file.

```
fp_mio_file = fopen ("‘/home/tc4200/
Scrivania/TC.txt"’, "‘a+"’);
```

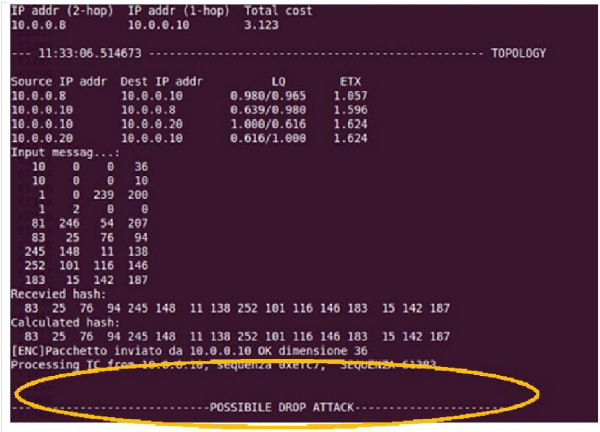


Figura 4.8: Output Detection Drop Attack

```
if(fp_mio_file == NULL){

printf("‘File vuoto\n"’);
}

fprintf(fp_mio_file, "‘%d\n"’, seqno);
fclose(fp_mio_file);}
```

```
static int dropattackdetection(){
int i, j, max, min, differenza;
int valori [100000];
....
....
fp_mio_file = fopen ("‘/home/tc4200/
Scrivania/TC.txt"’, "‘r"’);

if(fp_mio_file == NULL){

perror("‘Errore\n"’);
}

...
```

4.3 Neighbor Attack

L'obiettivo dell'attacker è quello di 'sconvolgere' il routing facendo credere a due nodi, che in realtà non sono connessi tra loro, di poter comu-

nicare direttamente. In OLSR i nodi conoscono la topologia ancor prima di iniziare una comunicazione, quindi affinchè i due nodi vittima possano credere di poter direttamente comunicare bisogna mandare periodicamente messaggi di HELLO. L'attacco consta di due operazioni, sniffing di pacchetti prima ed invio di Hello packet opportunamente modificati dopo.

Per la detection si è partiti dal presupposto che a destinazione arrivano pacchetti falsi e corrotti quindi è stato prima di tutto necessario valutare la loro correttezza; al setaccio sono stati passati:

- Message_Type
- Message_size
- TTL
- HopCnt

Fondamentalmente tutti i nodi durante il tentativo di attacco continuano ad avere salvata al loro interno la topologia corretta della rete; la destinazione quindi si vede arrivare pacchetti con parametri tipici di un nodo direttamen-

te connesso anche se sa benissimo che non è così. L'attacker genera false packet sia per 'Size', poichè privi di firma, che per 'HopCnt'. Il codice utilizzato è stato il seguente:

```
if((sig -> olsr_msgtype != MESSAGE_TYPE) ||
 (sig -> olsr_msg_size != ntohs(sizeof
(struct s_olsrmsg))) || (sig -> ttl !=1) ||
 (sig -> hopcnt !=0)){
olsr_printf(1, "‘ATTACCO -->
Packet not sane!\n"’);
return 0;
}
```

Conferma alla detection si ha dall'output emesso a destinazione ogni volta che si verifica un'anomalia (vedi Fig. 4.9).

Inoltre è possibile far si che dopo la detection vengano riattivate le procedure utilizzate ad inizio connessione o quando un nodo entra a far parte della rete.

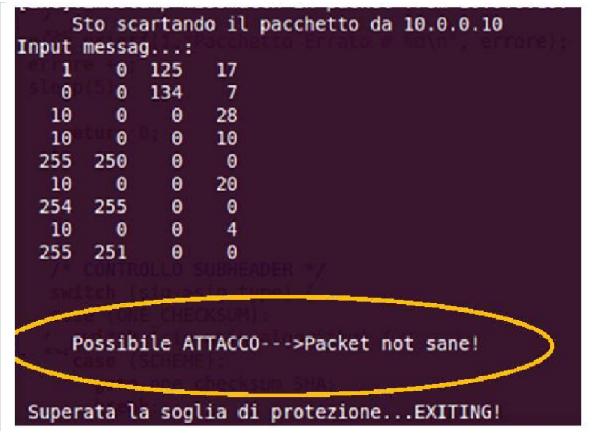


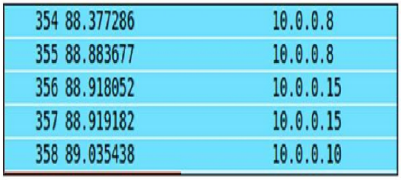
Figura 4.9: Output Detection Neighbor Attack

4.4 Sybil Attack

Il nodo malevolo pur di compromettere i servizi della rete impersona un altro nodo del tutto inesistente (vedi Fig. 4.10, 4.11); in OLSR vengono emessi messaggi di HELLO e TC rispettivamente ogni HELLO_INTERVAL e ogni TC_INTERVAL costruiti in maniera del tutto fittizia.

Figura 4.10: Sybil Mode On

La detection (vedi Fig. 4.12) per questo attacco è stata implementata in un duplice modo; infatti questa volta oltre ad analizzare i parametri dell'header vengono valutate le condizioni sui pacchetti prendendo in considerazione il relativo subheader; grazie a ciò è stato facile capire se si stanno cercando di apportare modifiche alla topologia attraverso l'annuncio di vicini inesistenti.



354	88.377286	10.0.0.8
355	88.883677	10.0.0.8
356	88.918052	10.0.0.15
357	88.919182	10.0.0.15
358	89.035438	10.0.0.10

Figura 4.11: Traffico Sniffato con Wireshark durante un Sybil Attack

```
switch ( sig -> sig.type){
case (ONE_CHECKSUM);
switch (sig -> sig.algorithm) {
case (SCHEME);
goto one_checksum_SHA;
break;
}
```

```
break;
default:
olsr_printf(1, "‘TENTATIVO DI
MODIFICA TOPOLOGIA!\n"’);
return 0;
}
```

Mentre avviene la validazione dei pacchetti ci si può accorgere che comunque si è sotto attacco perchè questi hanno dimensione diversa da quella prevista dal protocollo a causa della mancanza della firma e del timestamp, parametri non presenti poichè il nodo è inesistente e ogni valore di timestamp è univoco per ogni coppia di nodi.

4.5 Blackhole, Grayhole e Jellyfish

Per la detection (vedi Fig.4.13) in questo caso è stato necessario valutare il comportamento e le

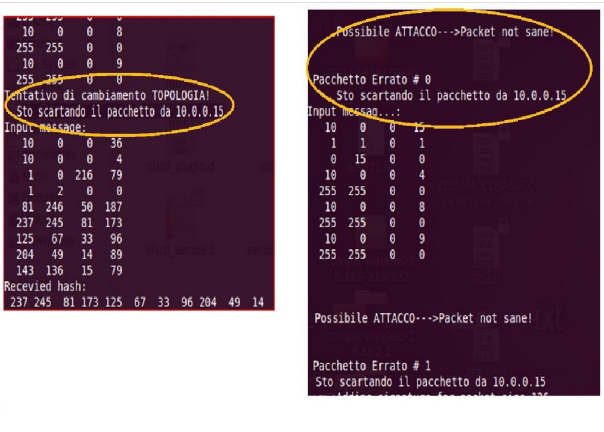


Figura 4.12: Output Detection Sybil Attack

mosse che compie l'attacker affinchè il suo obiettivo venga raggiunto. Nel capitolo due è stato evidenziato che questi tre attacchi constano di una duplice fase, la prima, praticamente uguale, è quella in cui l'attacker cerca di far si che il traffico passi da esso, mentre la seconda varia da attacco ad attacco. Infatti nel BLACKHOLE, piuttosto che inoltrare pacchetti, l'attacker li scarta, mentre nel GRAYHOLE il nodo

malevolo può scartare i pacchetti che arrivano sia con una certa probabilità, sia ad intervalli random; infine nel JELLYFISH si opera sui pacchetti per aumentare il ritardo end-to-end o portare a zero il goodput.

Per questi tre attacchi la detection è stata implementata in un duplice modo. Valutiamo la prima parte degli attacchi; il nodo malevolo, per far credere agli altri nodi della rete, che lui ha il percorso migliore, genera ed invia pacchetti non sani o corrotti; poichè ogni nodo, come previsto dalle regole del protocollo OLSR, mantiene salvato al suo interno la routing table corretta, è bastata l'analisi dell'header dei pacchetti per rendersi conto che vi è qualche anomalia.

```
if((sig -> olsr_msgtype != MESSAGE_TYPE)
&& (sig -> olsr_msg_size != ntohs(sizeof
(struct s_olsrmsg))) && (sig ->ttl !=1)){
olsr_printf(1, "‘ATTENZIONE
 PROBLEMI DI TIPOLOGIA, DIMENSIONE e
TTL --> BLACKHOLE/GRAYHOLE/JELLYFISH!\n"’);
```

```
sleep(2);
return 0;
}
```

É stato implementato anche un secondo controllo, che ha senso nel momento in cui il nodo malevolo riesce in un modo o in un altro a generare ed inviare pacchetti sani e validi. Il tutto nasce come conseguenza del fatto che inizialmente l'attacker, pur di far passare da esso il traffico, invia pacchetti corrotti con il fine unico di modificare la topologia della rete. Per cercare di render il più sicuro possibile il protocollo OLSR si è implementato un ulteriore controllo del traffico ed in particolare i controlli sono stati fatti sui parametri dei pacchetti contenenti i messaggi di TOPOLOGY CONTROL

```
switch ( sig -> sig.type){

...
```

```
switch (sig -> sig.algorithm) {
case (SCHEME);
goto one_checksum_SHA;
break;
}

...

return 0;
}
```

Analizzando il codice è facile intuire che viene prima valutata la 'signature' e poi lo 'scheme' ed è proprio quest'ultimo che viene corrotto nel momento in cui si desidera modificare la topologia. Dopo che la detection è andata a buon fine, la tabella di routing rimane inalterata ma grazie al 'return 0;' vengono rispediti tutti i messaggi di challenge affinchè si instauri nuovamente un rapporto di fiducia tra i neighbors. Volendo si poteva utilizzare un 'exit(0);' ed isolare il nodo vittima affinchè potessero essere fatti tutti gli

accertamenti del caso.

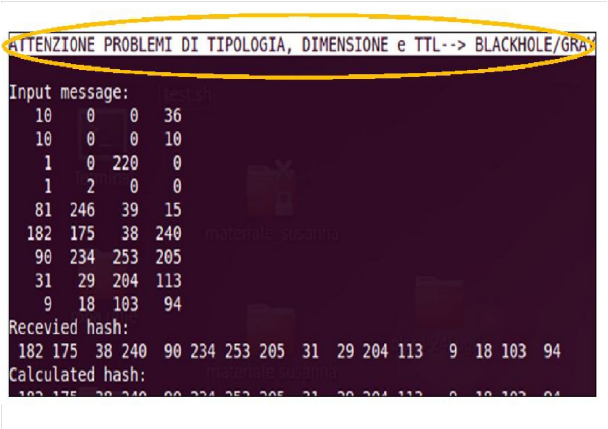


Figura 4.13: Output Detection

4.6 Wormhole

Dato che il principio base di questo attacco è la collaborazione dei due attackers, i quali attraverso l'utilizzo di un tunnel riescono a falsificare la lunghezza del percorso, è stato possibile implementare una contromisura semplicemen-

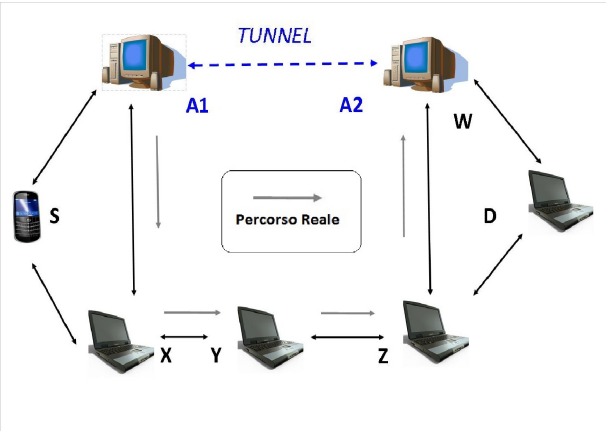


Figura 4.14: Figura Topologia Wormhole

te installando il plugin e sfruttando, come nel caso di attacco Replay, il valore di timestamp; infatti, nel momento in cui un pacchetto viene portato a destinazione da A2 (vedi Fig 4.14), non viene modificato alcun valore nell'header proprio e questo ha contribuito a trovare una soluzione al problema. Quando due nodi si vedono per la prima volta attraverso lo scambio di opportuni messaggi viene calcolata la δ, che in-

dica la discrepanza del timestamp tra i clock dei due nodi; il fatto che a destinazione arriva un pacchetto da parte di una determinata sorgente ed il valore della δ attuale non corrisponde con quello salvato in memoria fa si che scatti la detection facendo entrare il protocollo in uno stato di allerta. A schermo viene stampato un opportuno messaggio di ALARM ATTACK.

4.7 Discussione dei risultati e confronto con lo stato dell'arte

Dopo aver installato su tutte le macchine OLSR 0.5.6 e fatto partire il plugin, si è passati alla valutazione del livello di sicurezza che quest'ultimo riesce a garantire. I risultati sono stati quelli che gli unici attacchi che non sono andati a buon fine sono stati appunto quelli di tipo Replay e Wormhole poichè la destinazione si rendeva conto che, nonostante la 'correttezza' dei pacchetti, i valori del timestamp erano

in discrepanza rispetto a quelli calcolati e memorizzati dai vari nodi nel momento in cui è avvenuta la prima connessione. Partendo dallo studio dei vari attacchi e apportando opportune modifiche ai source code del protocollo e del plugin, si è riusciti a trovare almeno un modo per la detection di tutti gli altri attacchi.

Nel Drop Attack il protocollo è stato modificato in maniera tale da poter ricavare e salvare su file, i Sequence Number dei TC messages; per quanto riguarda l'attacco di tipo Neighbor il sistema di detection è stato implementato in seguito allo studio dei valori che vengono modificati per creare nuovi pacchetti; infatti, a seconda del 'MESSAGE_TYPE', il pacchetto si è dimostrato corrotto per particolari valori dei parametri. Nel caso di Sybil Attack il tutto è stato implementato in maniera abbastanza semplice perchè, dato che l'attacker cerca di impersonare un nodo inesistente, questo non fa altro che inviare messaggi non criptati ed inoltre cer-

ca di apportare modifiche alla topologia della rete. Per quanto riguarda invece gli attacchi di tipo Blackhole, Grayhole e Jellyfish, la linea che si è seguita è stata quella di sfruttare al massimo le debolezze che presentano questi tre attacchi nella prima parte, che è uguale per tutti; infatti grazie ad un duplice controllo sui parametri dell'header e del subhearder dei pacchetti è stato possibile non solo sventare questi attacchi, ma nello stesso tempo anche evitare di appesantire il protocollo mantenendolo il più snello possibile poichè utilizzare tre detection differenti implicava un aumento del carico computazionale su ogni macchina portando quindi ad un eccessivo spreco di risorse.

La tabella in Figura 4.15 riassume lo stato dell'arte e permette di confrontare le precedenti versioni sicure di OLSR [20][21][22][23] con la nostra.

ATTACKS	HYBRID Scheme	SIGNATURE Scheme	SOLSR	SA-OLSR	OLSR con PLUGIN	Cocilovo OLSR
Wormhole	√	X	√	√	√	√
Black hole	√	X	√	X	X	√
Replay	×	√	X	X	√	√
Drop	√	X	√	X	X	√
Jellyfish	-	-	-	-	X	√
Grayhole	-	-	-	-	X	√
Neighbor	√	X	√	√	X	√
Sybil	-	-	-	-	X	√

Figura 4.15: Tabella Riassuntiva

Capitolo 5

Sicurezza AODV

In questo capitolo ci si propone di analizzare i vari attacchi alle reti MANET che utilizzano l'AODV protocol, per trovare, attraverso una serie di valutazioni teoriche, quelle che potrebbero essere delle soluzioni da implementare per garantire come nel caso di OLSR un maggior livello di sicurezza.

5.1 Messaggi scambiati

Il protocollo AODV, essendo di tipo reattivo, non prevede il salvataggio da parte di ogni singolo nodo delle routes esistenti nella rete e, di conseguenza non è previsto neanche il salvataggio delle routing tables. Ogni volta che un nodo desidera scambiare informazioni o comunque offre connettività entrando a far parte, in maniera attiva della rete, non fa altro che comunicare la sua presenza in modo tale che gli altri nodi possano calcolare la route migliore per arrivare ad esso.

Prima di passare alla descrizione di come sia possibile implementare una versione del protocollo capace di difendersi e fare detection dei vari attacchi, analizziamo velocemente i messaggi scambiati; essi sono RREQ, RREP e RRER vengono scambiati anche messaggi di tipo HELLO. Vediamo quindi la loro utilità e come sono settati i parametri che li costituiscono:

RREQ

Questo messaggio viene mandato da un nodo nel momento in cui questo desidera raggiungere una determinata destinazione per la quale non ha alcuna ROUTE.

Il parametri vengono settati nel seguente modo:

DEST. SEQUENCE NUMBER=0; Se non si conosce quello della destinazione altrimenti si setta uguale all'ultimo 'Seqno' conosciuto per quella destinazione

RREQ_ID=x; con 'x' numero intero che viene incrementato ogni volta che viene mandata una richiesta

HOPCOUNT= 0;

RREP

Questo messaggio viene inviato come risposta ad una richiesta precedentemente inoltrata. Ovviamente è la destinazione che manda questo tipo di messaggio.

RRER

Messaggio mandato in caso di errori o anomalie varie come nel caso in cui dovesse cadere un link.

HELLO

Pacchetto che viene periodicamente mandato, da un nodo ai neighbor, per indicare la propria disponibilità a connettersi. Esso non è altro che un RREP con i parametri settati nel seguente modo:

TTL=1;

DESTINATION_ADDRESS = id di chi riceve il messaggio;

ORIGINATOR_ADDRESS = id di chi genera il messaggio;

HOP_COUNT=0;

LIFETIME= tempo dopo il quale se non si riceve un nuovo HELLO il link si considera perso.

5.2 Attacchi

5.2.1 Replay

Questo tipo di attacco consiste nel replicare i pacchetti che arrivano all'attacker e che sono destinati a qualsiasi altro nodo della rete. I pacchetti di tipo RREQ hanno un parametro che li rappresenta in maniera univoca, esso è l'ID un primo controllo lo si può fare considerando la coppia di parametri **RREQ_ID, ORIGINATOR_IP_ADDRESS**; infatti, se a destinazione ci si rende conto che un nodo con un determinato IP ha emesso due RREQ con lo stesso ID, vuol dire che il pacchetto è appunto replicato; inoltre si può fare un controllo anche su tutti i pacchetti RREP, poichè essendo contraddistinti dalla presenza di un sequence number è facile intuire che se esistono due pacchetti con lo stesso numero di sequenza uno dei due è una replica.

In generale, si può vedere attraverso 'WIRE-

SHARK' che a destinazione un eventuale pacchetto replicato arriva subito dopo il pacchetto originale, ragion per cui si può adottare la stessa tecnica utilizzata per fare detection nel protocollo OLSR. Infatti utilizzando la libreria 'Libpcap' si può creare un filtro e sniffare il traffico in maniera tale da analizzare gli header dei vari pacchetti e vedere se questi sono replicati o meno.

```
char filter_exp[] = "ip src host
 10.0.0.8 && udp port 698";
bpf_u_int32 mask;

...

if (pcap_lookupnet(dev, &net,
&mask, errbuf) == -1) {
fprintf(stderr, "Can't get netmask
for device %s\n", dev);
net = 0;
mask = 0;
```

```
 }

....
....

if (pcap_setfilter(handle, &fp) == -1) {
 fprintf(stderr, "Couldn't
install filter %s: %s\n",
filter_exp, pcap_geterr(handle));
return(2);
 }

while(1){

packet1 = pcap_next(handle, &header);
packet2 = pcap_next(handle, &header);
/*Confronto*/
if(packet1==packet2){
printf ("Uguali!\n");
}
```

```
if(packet1!=packet2){
printf ("Diversi!\n");
}
}


/* Cleaning up */

pcap_freecode(&fp);
pcap_close(handle);
```

5.2.2 Drop

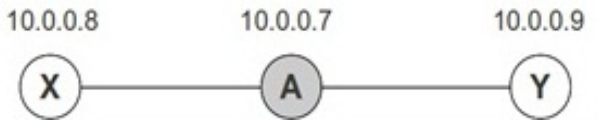


Figura 5.1: Topologia Drop Attack

Come noto, nel momento in cui avviene la connessione (vedi Fig. 5.1) sia il nodo **X** che il nodo

Y sanno che per dialogare l'uno con l'altro devono comunque inviare i pacchetti ad **A** che a sua volta li consegnerà al nodo di destinazione. L'attacco consiste nel far saltare il link che passa per **A (attacker)** e che collega sorgente e destinazione. Dato che se l'attacker applica la regola

```
# iptables -I FORWARD -j drop -s 10.0.0.8
```

questa fa si che i pacchetti di **X** non arrivano a destinazione un'ipotetica soluzione per fare detection potrebbe esser appunto quella di interrogare ogni τ_s sec la destinazione per accertarsi che comunque il link è attivo. Si può pensare di fare questa operazione in vari modi, tra questi:

- Invio di una nuova RREQ
- Pingare la destinazione.

In entrambi i casi la sorgente si aspetta un messaggio di risposta; nel caso in cui il link dovesse esser danneggiato, la prima delle due opzioni contribuirà a ripristinare il tutto, mentre la

seconda richiederà l'invio di una nuova RREQ dopo un certo periodo τ_r . É inoltre consigliato implementare l'algoritmo in maniera tale da far si che τ_s sia random perchè altrimenti in caso contrario l'attaccante potrebbe impersonificare il nodo di destinazione e mandare periodicamente messaggi di RREP.

5.2.3 Neighbor

L'attacco consiste nel far credere a **S** di esser direttamente connesso a **D** e viceversa. Per far ciò **A** opera nel seguente modo (vedi Fig.5.2):
se la sorgente invia una RREQ ad **A**, quest'ultimo prende il pacchetto ricevuto, ne copia tutti i campi e ne crea uno nuovo da mandare a destinazione. Subito dopo aver fatto questa operazione, l'attacker crea una RREP da inviare a **S**. Dato che comunque alla base di tutto vi è un continuo furto di identità (infatti **A** impersona una volta **S** ed un'altra **D**), si potrebbe sfruttare la crittografia asimmetrica garantendo così

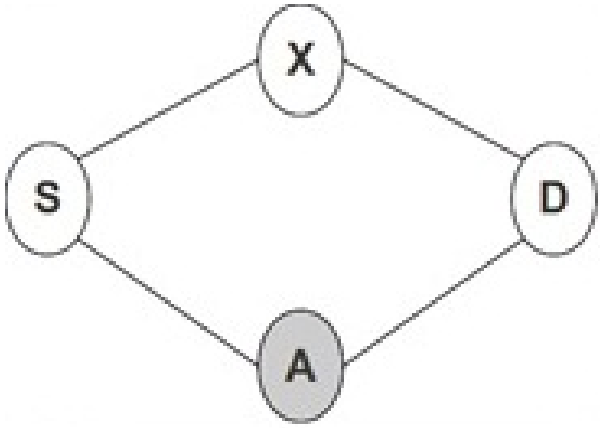


Figura 5.2: Topologia Neighbor Attack

che solamente a destinazione il pacchetto possa essere letto; inoltre si verrebbe a creare anche un canale sicuro per lo scambio delle chiavi. Se **A** invece riceve un HELLO packet da **S**, tutto ciò non ha più senso perchè l'attacker può leggere ugualmente il messaggio e ricreare il tutto; una possibile soluzione potrebbe essere quindi quella che prevede l'utilizzo del **timestamp**.

Considerando che quando due nodi si vedono per la prima volta viene calcolata e memorizzata la differenza δ tra i clock, non appena l'attacker processa decifra e reinvia l'HELLO packet, perderà un tempo maggiore di δ facendo si che,

$$|Ts - T| < \delta$$

non sia più verificata.

5.2.4 Sybil

Questo attacco consiste nel prendere le veci di un nodo che non esiste e far si che questo, essendo visto come un nodo benevolo, diventi parte integrante della rete riuscendo così a scambiare informazioni e pacchetti vari con tutti gli altri utenti della MANET (vedi Fig. 5.3).

Nel momento in cui si voglia garantire nella rete configurata ad-hoc un minimo di sicurezza necessita non fidarsi più dei propri vicini, o comunque dei nodi che vogliono entrare a fare

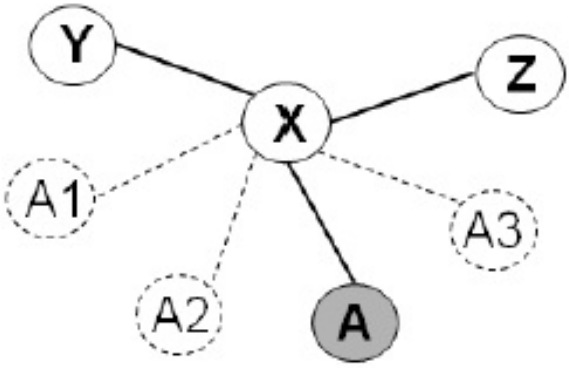


Figura 5.3: Topologia Sybil Attack

parte della MANET, e richiedere delle informazioni per avere certezze. Alla base di questo attacco vi è proprio la violazione della fiducia, ragion per cui una possibile soluzione potrebbe essere appunto quella di creare una rete nella quale è implementata una crittografia asimmetrica e basata sulla presenza della **CA** $Certification Authority$; quest'ultima, attraverso la generazione dei certificati, riesce a garantire l'autenticità del nodo e nel caso spe-

cifico dell'attacco Sybil il nodo nuovo, essendo inesistente, è non identificabile.

5.2.5 Blackhole, Grayhole, Jellyfish

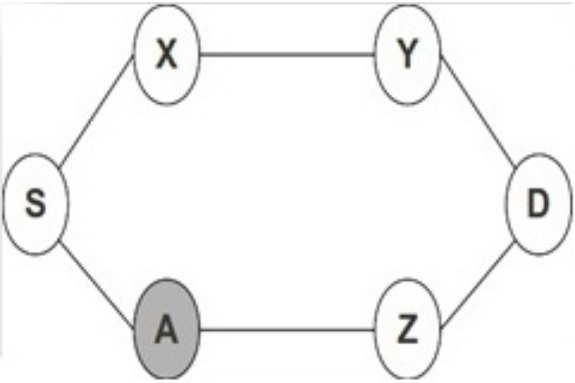


Figura 5.4: Topologia Blackhole/Grayhole/Jellyfish Attack

Anche alla base di questi tre attacchi vi è una continua operazione di sniffing di traffico e una continua falsificazione di pacchetti da parte dell'attacker. Utilizzando AODV, nel momento in cui un nodo desidera raggiungere una destina-

zione, per la quale non ha una route, dissemina una RREQ e quando il nodo di destinazione si vede arrivare questo messaggio risponde con una RREP.

Nella prima fase degli attacchi in questione l'attacker si mette in ascolto cercando di captare il traffico quindi o una RREQ proveniente dalla sorgente vittima oppure un pacchetto di HELLO proveniente da un neighbor, poichè appunto il parametro di interesse è il **Destination Sequence Number**; acquisito questo parametro, l'attacker passa alla creazione di una RREP falsa.

Dopo aver analizzato la procedura precedentemente descritta, le riflessioni fatte hanno portato alla conclusione che una crittografia asimmetrica potrebbe risolvere i problemi riscontrati; infatti, nel momento in cui un nodo che non è la destinazione si vede arrivare una RREQ, non riesce a decifrarla evitando così di leggere e modificare i vari campi; inoltre, grazie all'utilizzo

della coppia di chiavi pubblica-privata, la sorgente riesce a verificare se la RREP è autentica e quindi è stata inviata realmente da **D**.

Supponendo che la prima parte degli attacchi sia andata a buon fine, possiamo pensare di fare detection basandoci sul comportamento che ha l'attacker nel momento in cui desidera portare a termine il suo lavoro. Nel caso di Blackhole, una volta che l'attacker è sicuro che il traffico passi da esso, non fa altro che applicare le regole di routing ed in modo particolare fa si che i pacchetti non arrivino più a destinazione come nel caso di Drop Attack. Per evitare che questo attacco vada a buon fine, si può pensare di attuare la stessa strategia di difesa presentata per il Drop Attack, inviando quindi in maniera random una RREQ per verificare la presenza o meno del link.

Per quanto riguarda il Grayhole Attack la seconda fase dell'attacco consiste nell'accodamento dei pacchetti e nell'invio di alcuni di essi ga-

rantendo così che solo una parte dei dati arrivi a destinazione. Qui non viene presentata una vera e propria soluzione al problema; da un punto di vista teorico si può pensare di implementare un counter e attraverso la valutazione della percentuale di traffico perso fare delle ipotesi in merito all'affidabilità dei link (il fatto che molto traffico venga perso implica che comunque sta succedendo qualcosa di strano).
Analogalmente nel Jellyfish Attack, previa studio accurato delle prestazioni della rete, si possono settare delle soglie e valutare, per esempio, un delay massimo oltre il quale non si dovrebbe normalmente andare.

5.2.6 Sleep Deprivation

Lo Sleep Deprivation mira a consumare le risorse dei nodi tenendoli constantemente impegnati a processare pacchetti inutili. L'attacker può, per esempio, inviare al nodo vittima una grande quantità di pacchetti di route request, di route

Attack	Esito
Replay	OK
Drop	OK
Neighbor	OK
Sybil	OK
Blackhole	OK
Grayhole	OK
Jellyfish	OK
Sleep Deprivation	OK

Tabella 5.1: AODV protection

replay o di route error. Come risultato finale il nodo in questione diventa incapace di partecipare ai vari meccanismi di routing poichè perennemente impegnato in altre attività. Dato che comunque questo attacco è simile al Replay Attack, si può pensare di attuare le stesse tecniche di difesa descritte precedentemente.

5.2.7 Considerazioni finali

Dopo aver brevemente descritto i vari attacchi che un nodo malevolo può compiere in una rete MANET nella quale viene usato il protocollo AODV e aver trovato, almeno una possibile soluzione per ognuno di essi, possiamo concludere che le tecniche precedente descritte hanno senso nel momento in cui l'attacker è parte integrante della rete. Pertanto, risulta necessaria una prima autenticazione da parte dei nodi ogni qual volta che uno di essi desidera entrare a far parte della rete. Per far ciò si può pensare di sfruttare, come nel caso in cui si utilizzi OLSR, una crittografia simmetrica, poichè più leggera dal punto di vista del carico computazionale però dato che comunque per contrastare molti attacchi è stato proposto di utilizzare una crittografia asimmetrica è consigliato sfruttare quest'ultima anche per cercare di autenticare i vari nodi che desiderano entrare a far parte della network.

Conclusioni

Lo scopo della presente tesi è stato quello di mostrare come costruire un test-bed per valutare e successivamente migliorare il grado di sicurezza garantito nelle reti MANET.

Si è partiti analizzando lo scenario ed il contesto in cui nascono e vengono utilizzate queste reti. Lo sviluppo delle MANET è sempre in continua evoluzione poichè vi è la tendenza a migliorare le prestazioni cercando il modo più efficiente per far funzionare le reti create ad-hoc. Come detto pocanzi è stato costruito un test-bed ponendoci come obiettivo quello di risolvere i problemi di

sicurezza poichè il futuro di tali reti dipende fortemente dalla loro risoluzione.

Dopo l'analisi iniziale si è passati allo studio dei protocolli di routing presenti nello stato dell'arte; essi si dividono in Proactive e Reactive routing Protocol e per ciascuna delle due classi si è scelto un protocollo. Per la classe dei protocolli proattivi è stato scelto OLSR (Optimized Link State Routing Protocol), mentre per la seconda classe AODV (Ad hoc On-Demand Distance Vector Routing); in entrambi i casi sono state analizzate sia le procedure attraverso le quali vengono inoltrati i pacchetti sia appunto i messaggi e i pacchetti scambiati.

Successivamente, è stata fatta una ricerca di tutti quelli che sono i possibili attacchi implementati dagli hackers per queste reti, giungendo alla conclusione che un nodo malevolo può recare danni in vari modi; tra questi vi è la possibilità di scartare, ritardare o inviare fuori sequenza i pacchetti che passano da esso stesso.

Infine dopo una valutazione di quelli che sarebbero stati i tool necessari per il raggiungimento del nostro obiettivo, si è passati all'implementazione, nel caso di OLSR, di una versione sicura del protocollo; il tutto è stato possibile grazie in un primo momento ad un'accurata e minuziosa analisi e alle opportune modifiche in un secondo momento del source code.

Per testare il nuovo protocollo sono state create varie topologie e sono stati simulati i vari attacchi precedentemente analizzati. Il risultato è stato appunto quello che il nuovo protocollo implementato riesce a fare detection di ogni singolo attacco mettendo i nodi coinvolti in uno stato di ALERT ed evitando così che questi vadano a buon fine.

Inoltre, per quanto riguarda il protocollo AODV, è stato fatto uno studio teorico alla luce delle contromisure proposte per OLSR, di quelle che potrebbero essere le soluzioni, implementabili in futuro, per cercare di rendere anch'esso sicu-

ro, raggiungendo quindi lo stesso risultato del protocollo proattivo.

Appendice A

Manuale

A.1 Fase di setup e installazione

A.1.1 Configurare scheda di rete

- Installare Ubuntu 10.04 su tutti i pc
- Eseguire i seguenti comandi per aggiornare i pacchetti presenti ed installare quelli necessari (connesso a Internet)

```
# sudo su  \(Immettere la password \TC4200"\)
# apt-get update
# apt-get upgrade
# reboot

# apt-get update
# apt-get install build-essential
# apt-get install linux-source
# apt-get install linux-headers-generic
# apt-get install linux-libc-dev

# apt-get install iw
```

- Per installare la scheda di rete da connettere tramite porta USB bisogna salvare il firmware **ar9271.fw** e la cartella **compact-wireless-2.6.38.2-2.tar.bz2**
- Eseguire i comandi

```
# cp ar9271.fw  /lib/firmware
```

```
# tar xvf  compact-wireless-2.6.38.2-2.tar.bz2
# cd compact-wireless-2.6.38.2-2
# ./scripts/driver-select ath9k_htc
# make
# make install
```

– Per controllare se il driver della scheda è caricato

```
# lsmod | grep ath9
```

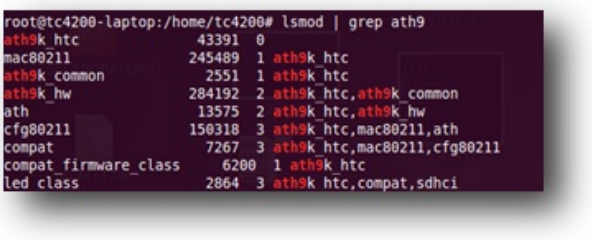


Figura A.1: Verifica installazione scheda di rete

A.1.2 Configurare protocollo

I seguenti protocolli devono essere installati su tutte le macchine che fanno parte del Test-bed.

AODV

- Salvare la cartella **aodv-uu-0.9.6.tar.gz**
- Eseguire i seguenti comandi

```
# tar xvf  aodv-uu-0.9.6.tar.gz
# cd aodv-uu-0.9.6
# make && make install
```

- Per far partire il protocollo

```
# aodvd -l -d -D -r 3
(così gira in background e per vedere l'output:
# cat /var/log/aodvd.log)

# aodvd -l -D -r 3
(così NON in background )
```

OLSR

– Eseguire i seguenti comandi

```
# apt-get install olsrd
# cp /etc/olsrd/olsrd.conf/etc
```

– Per far partire il protocollo

```
# olsrd -i interfaccia -d valore
```

valore: indica quante informazioni di debug vengono visualizzate in output, assume i valori da 0 a 9, dove 0 significa che il protocollo gira in background .

OLSR Versione Sicura

– Eseguire i seguenti comandi

```
# cd  /...../olsrd-Cocilovo
# make clean && make && make install
# cd /olsrd-Cocilovo/lib/secure
# make clean && make && make install
```

Considerazioni e problemi riscontrati

– Se l'interfaccia della scheda di rete non è corretta bisogna editare il file **"olsrd.conf"**

e modificare l'apposito parametro di **"Interface"** .

```
# gedit  /etc/olsrd.conf
```

– Se non viene caricato il Plugin di sicurezza, dopo l'opportuna installazione, bisogna editare il file **"olsrd.conf"** e aggiungere le seguenti linee a fine file:
LoadPlugin "olsrd_secure.so.0.5" {
PlParam "Keyfile" "/etc/olsrd.d/olsrd_secure_key"
}

Ovviamente bisogna assicurarsi che esistano tutte le directory richieste, in caso contrario seguire la seguente procedura:

– Creare la directory opportura attraverso il comando

```
# mkdir /etc/olsrd.d
```

– Creare il file che conterrà la chiave

```
# gedit /etc/olsrd.d/olsrd\_secure\_key
```

– Creare la chiave a 128-bit

```
#md5sum package-name
```

Figura A.2: Generazione Chiave

– Prendere la key restituita dal passaggio precedente e copiarlo all'interno del file creato al passaggio 2.

NOTA: La chiave può essere calcolata in qualsiasi altro modo pur che sia di 128-bits. Io ho preferito l'MD5 solo perché sono sicuro che mi veniva restituita una stringa di quella dimensione. Ricorda che la chiave deve essere uguale per tutte le macchine. Per esempio al posto di package-name si può mettere il seguente path /etc/olsrd.d/olsrd_secure_key.

– Per far partire il protocollo

```
# olsrd -d valore
```

valore: indica quante informazioni di debug vengono visualizzate in output, i valori vanno da 0 a 9, dove 0 significa che il

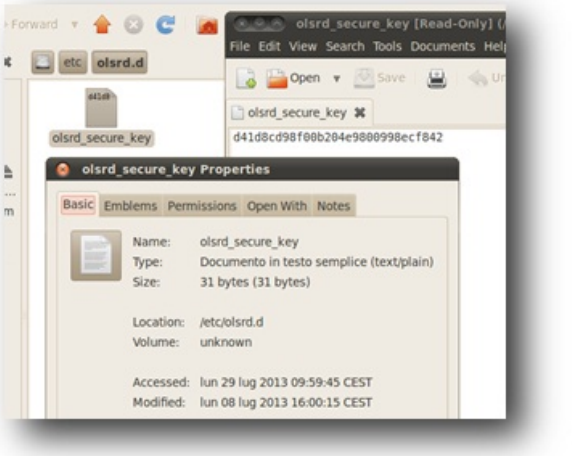


Figura A.3: Salvataggio Chiave

protocollo gira in background mentre 2 è il valore tipicamente utilizzato per ricevere in output le info di nostro interesse.

A.2 Parte Attacker

A.2.1 Installare librerie

- Salvare le cartelle
  - libnet-1.1.5.tar.gz
  - libpcap-1.1.1.tar.gz
  - iptables-1.4.13.tar.bz2
  - libnfnetlink-1.0.0.tar.bz2
  - libnetfilter_queue-1.0.1.tar.bz2
- Per installare le varie librerie:
  - scompattare la cartella
  - entrare nella cartella
  - eseguire i comandi

```
# ./configure
# make && make install
```

Considerazioni e problemi riscontrati

- Libnet

1. Questa libreria è stata modificata per poter creare i pacchetti dei protocolli usati (dettagli nella prossima sezione)
2. Se quando si compila non vine visto libnet.so.1 necessita creare un link simbolico eseguendo il seguente comando

```
# ln -s /usr/local/lib/libnet.so.1
/usr/lib
```

- Libpcap
  - Può capitare che quandi si esegue il comando

    ```
    # ./configure
    ```

    si riscontrano degli errori, in tal caso bisogna digitare

    ```
    # ./configure --without-flex
    --without-bison
    ```

    (il tutto è specificato nel MANUALE presente nella cartella della libreria)
  - Sui tablet (usati come attackers solo in Wormhole Attack) può capitare che

vengano restituiti errori quindi necessita installare anche le seguenti librerie nel seguente ordine:

· m4-1.4.13

· bison-2.4.1

· flex-2.5.35

· libpcap-1.0.0

– Libnetfilter_queue

1. Per installare questa libreria bisogna installare anche le due precedenti

2. Se in fase di compilazione si dovesse presentare lo stesso errore di Libnet digitare

```
# ln -s ... .so.1  /usr/local/lib /usr/lib
```

A.2.2 Modifica di Libnet per aggiungere header

(esempio in libnet-1.1.5/doc/PACKET_BUILDING)

– definire la grandezza del nuovo header (in bytes) e aggiungerla in ...libnet-1.1.5/include/libnet/libnet-headers.h

- creare la struttura del nuovo header e aggiungerla alla fine di libnet-headers.h
- aggiungere il protocol block identifier alla fine della lista in ...libnet-1.1.5/include/libnet/libnet-structures.h
- rieseguire i comandi

  ```
  # cd libnet-1.1.5
  # ./configure
  # make && make install
  ```

  Sono stati introdotti gli header dei protocolli usati quindi create le strutture mostrate nel file *strutture-header*.

A.3 Fase di implementazione

A.3.1 Configurare rete ad hoc

- Configurare rete ad hoc

  ```
  # ifconfig
  ```

- Creare uno script utilizzando il comando

  ```
  # gedit test.sh
  ```

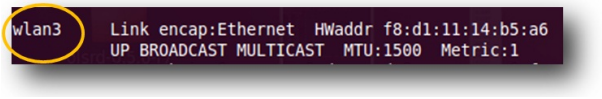


Figura A.4: Interfaccia scheda di rete

- Scrivere all'interno dello script precedente creato i seguenti comandi:

  ```
  # ifconfig INTERFACCIA down
  # service network-manager stop
  # iw INTERFACCIA set type ibss
  # ifconfig INTERFACCIA INDIRIZZO up
  (indirizzo IP  che gli vuoi assegnare)
  # iw INTERFACCIA ibss join test 2412
  ```

- Dare gli opportuni permessi allo script appena creato

  ```
  # chmod a + x /root/.../test.sh
  ```

- Per configurare la rete ad-hoc basta semplicemente far partire lo script

  ```
  # ./ test.sh
  ```

- Per emulare la topologia fisica dare il seguente comando su tutti i pc per OGNI no-

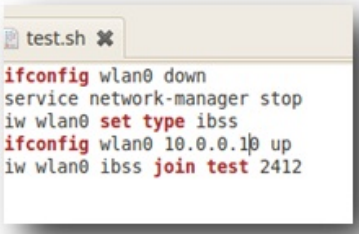


Figura A.5: Script

do che NON è in visibilità nella topologia desiderata

```
# iptables -I INPUT -m mac
--mac-source  indirizzo -j DROP
```

(indirizzo: MAC del nodo che NON vede)

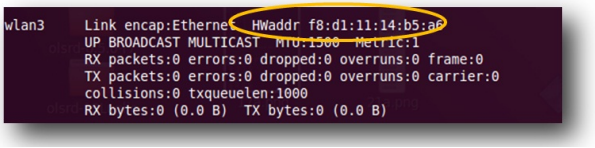


Figura A.6: Indirizzo Scheda di Rete

– Avviare il protocollo di routing

A.3.2 Configurare tunnel

Estremi del tunnel A1 e A2

Indirizzi dati:

S = 10.0.0.3

D = 10.0.0.7

A1 = 10.0.0.9

A2 = 10.0.0.8

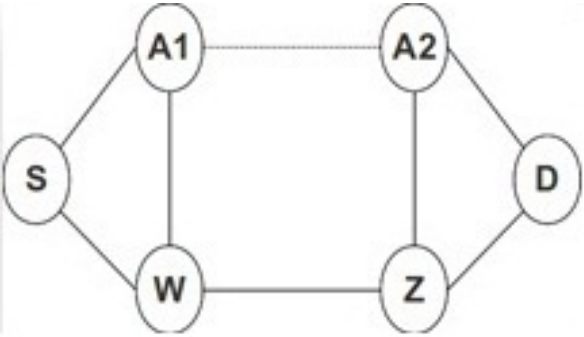


Figura A.7: Topologia con Tunnel

Su A1 dare questi comandi :

```
# iptunnel add tun mode gre remote 10.0.0.8
local 10.0.0.9 ttl 255
# ifconfig tun 10.0.0.19 up
# ifconfig tun pointopoint 10.0.0.18
# route add 10.0.0.7 gw 10.0.0.19 dev tun
(route per mandare traffico nel tunnel)
```

Su A2 dare questi comandi :

```
# iptunnel add tun mode gre remote 10.0.0.9
local 10.0.0.8 ttl 255
# ifconfig tun 10.0.0.18 up
# ifconfig tun pointopoint 10.0.0.19
# route add 10.0.0.3 gw 10.0.0.18 dev tun
(route per mandare traffico nel tunnel)
```

A.4 Attacchi

(fare attenzione alle interfacce che siano corrette negli script)

A.4.1 Far partire un attacco

Bisogna rifarsi alla cartella "Menu". Per far partire qualsiasi tipologia di attacco basta spostarsi all'interno della cartella, digitare

```
# ./selezione
```

e seguire il menu che si aprirà scegliendo l'attacco da far partire.

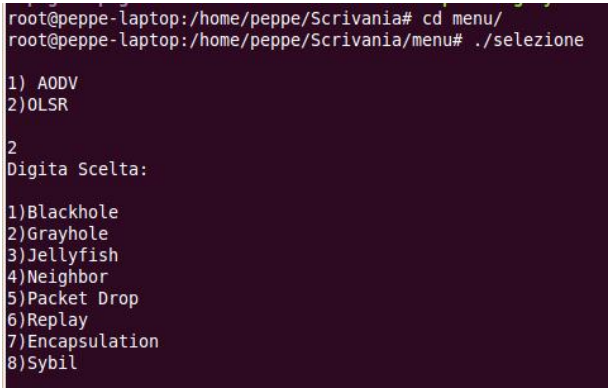


Figura A.8: Menu

Se dovessero verificarsi problemi di compatibilità tra i vari scripts ed il pc che funge da at-

Figura A.9: Attacco Mode ON

tacker basta ricompilare il tutto eseguendo i seguenti comandi:

```
# cd .../menu
#gcc -o selezione selezionetotale.c
#gcc -o selezioneOLSR selezioneOLSR.c
#gcc -o selezioneAODV selezioneAODV.c
```

AODV cartella menu/AODV

OLSR cartella menu/OLSR

```
#cd .../menu/AODV (OLSR)
```

A.4.2 Blackhole

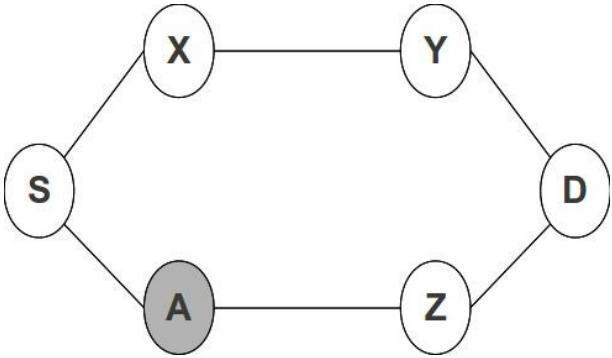


Figura A.10: Topologia Blackhole Attack

Attacker A

Victims S e D

S = 10.0.0.9

D = 10.0.0.8

AODV

Modo 1 (scarto dei pacchetti effettuato da A)

```
# gcc -o blackhole1ok blackhole1ok.c
 -lnet -lpcap
```

Modo 2 (i pacchetti vengono inviati a un nodo inesistente)

```
# gcc -o blackhole2ok blackhole2ok.c
 -lnet -lpcap
```

OLSR

```
#gcc -o provablackhole provablackhole.c
-lnet -lpcap
```

A.4.3 Grayhole

(stessa topologia di Blackhole)

AODV

```
# gcc -o grayhole grayhole.c -lnet -lpcap
-lnetfilter_queue -lpthread
```

OLSR

```
#gcc -o grayholeOLSR grayholeOLSR.c  -lnet
-lpcap -lnetfilter_queue -lpthread
```

A.4.4 Jellyfish

(stessa topologia di Blackhole)

AODV

Delay Variance

```
# gcc -o jf_delayvariance  jf_delayvariance.c
-lnet -lpcap -lnetfilter_queue -lpthread
```

Reorder Buffer

```
# gcc -o jf_reorderbuffer  jf_reorderbuffer.c
-lnet -lpcap -lnetfilter_queue -lpthread
```

OLSR

```
# gcc -o jf_reorder_OLSR  jf_reorder_OLSR.c
-lnet -lpcap -lnetfilter_queue -lpthread
```

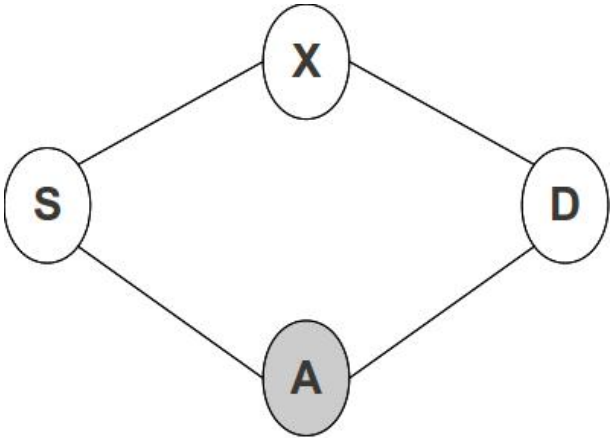


Figura A.11: Topologia Neighbor Attack

A.4.5 Neighbor

Attacker A

Victims S e D

S = 10.0.0.8

D = 10.0.0.9

AODV/OLSR

```
# gcc -o neighbor neighbor.c -lnet
-lpcap -lnetfilter_queue -lpthread
```

A.4.6 Packet Drop

Attacker A

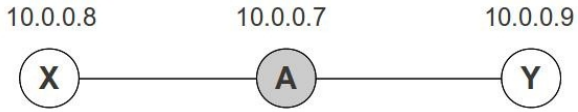


Figura A.12: Topologia Packet Drop Attack

AODV/OLSR

```
# gcc -o pkt_drop pkt_drop.c
```

A.4.7 Replay

(non è stata usata nessuna configurazione particolare, ho solo verificato con Wireshark che replicasse effettivamente i pacchetti voluti)

AODV/OLSR

```
# gcc -o replay replay.c -lpcap
```

A.4.8 Sleep Deprivation

(stesso discorso)

AODV

```
# gcc -o sleepdeprivation sleepdeprivation.c
 -lnet
```

A.4.9 Sybil

(stesso discorso)

AODV

```
# gcc -o sybil sybil.c -lnet -lpcap
-lnetfilter_queue -lpthread
```

OLSR

```
#gcc -o provaSybilOLSR provaSybilOLSR.c
-lnet -lpcap -lnetfilter_queue -lpthread
```

A.4.10 Whormhole

(topologia descritta quando si è parlato di tunnel)

Sulla macchina con indirizzo 10.0.0.8

```
#gcc -o encaps8 encaps8.c -lnet -lpcap
```

```
-lnetfilter_queue
#./ encaps8
```

Sulla macchina con indirizzo 10.0.0.9

```
#gcc -o encaps9 encaps9.c -lnet -lpcap
-lnetfilter_queue
#./ encaps9
```

Nota: per evitare eventuali malfunzionamenti è consigliato durante la procedura di compilazione non modificare i nomi dei file di output ed digitare i comandi così come sono scritti. Nel momento in cui si vuole far partire un attacco conviene prima editare il file per vedere se gli indirizzi IP sono corretti dopodiché basta salvare e ricompilare .

A.5 Operazioni effettuate sui pacchetti

Le operazioni principali che sono state fatte sui pacchetti sono quelle di Sniffing, Creazione e In-

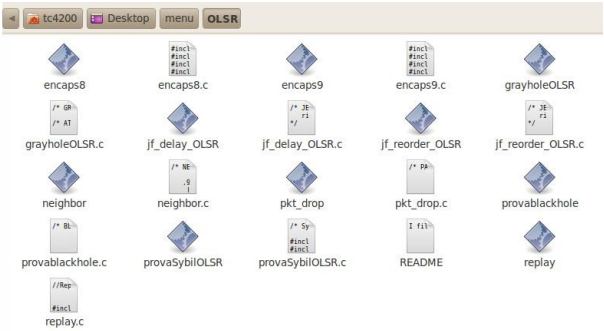


Figura A.13: Cartella con gli script degli attacchi contro l'OLSR

codamento. Esse sono state implementate come segue.

A.5.1 Sniffing

La libreria **"Libpcap"** è stata utilizzata per appunto sniffare il traffico. Esistono varie release di questa API (Application Programming Interface), nel caso specifico è stata utilizzata la Libpcap-1.1.1 . L'operazione di sniffing è composta da vari passaggi.

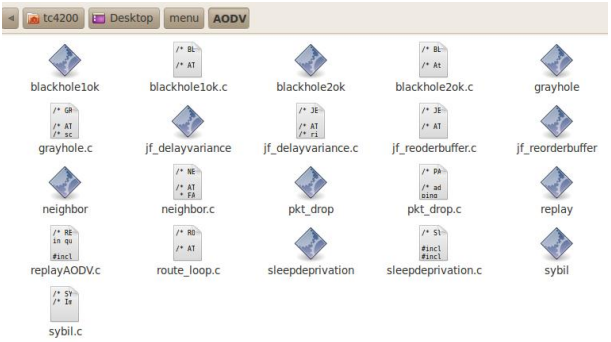


Figura A.14: Cartella con gli script degli attacchi contro l'AODV

Prima di tutto bisogna specificare l'interfaccia di rete da configurare affinché la cattura dei pacchetti abbia esito positivo.

```
pcap_t *handle;
handle = pcap_open_live(dev,BUFSIZ,1,
1000,errbuf);
if (handle == NULL) {
fprintf( stderr, "Error Couldn't
open device %s: %s",dev, errbuf);
return(2);
```

```
}
```

Da notare la funzione **"pcap_open_live()"** la quale prende in ingresso i seguenti parametri:

device indica l'interfaccia utilizzata dalla rete ad-hoc per la cattura del traffico.

snaplen indica il numero massimo di bytes da catturare per pacchetto.

promisc flag che viene configurato per determinare la modalità di settaggio dell'interfaccia di rete, "promiscua o no".

to ms è un timeout che conta i millisecondi che passano prima che l'operazione di lettura finisca.

errbuf buffer per i messaggi di errore.

Utilizzando questa libreria è possibile anche creare un filtro **"bpf"** (berkey packet filter) selezionando quindi solamente i pacchetti a cui si è interessati. Il bpf viene compilato tramite la funzione **"pcap compile()"** e impostato con la funzione **"pcap setfilter()"**.

```
struct bpf_program fp;
char filter_exp[] = "ip src host address
&& udp port port";
bpf_u_int32 mask;
bpf_u_int32 net;
if (pcap_lookupnet(dev, &net, &mask,
errbuf) == -1) {
fprintf(stderr, "Can't get netmask %s", dev);
net = 0;
mask = 0;
}

if (pcap_compile(handle, &fp,
filter_exp, 0, net)==-1) {
fprintf(stderr,"Couldn't parse filter
%s: %s",filter_exp,
pcap_geterr(handle));
return(2);
}

if (pcap_setfilter(handle,&fp)==-1) {
```

```
fprintf(stderr, "Couldn't install filter
 %s: %s", filter_exp, pcap_geterr(handle));
return(2);
}
```

Libpcap offre diverse opzioni per la cattura e il processing dei pacchetti. Le due funzioni utilizzate nel test-bed sono **"pcap next()"** e **"pcap loop()"**. La prima viene utilizzata per sniffare un singolo pacchetto e spesso è utilizzata anche per memorizzare il pacchetto da replicare successivamente (vedi attacco Replay).

```
packet = pcap_next(handle, &header);
pcap_sendpacket(handle,packet,header.len);
```

La seconda funzione per eseguire l'analisi, quindi capire il tipo di pacchetto ed eventualmente memorizzare i campi a cui si è interessati, su ogni messaggio catturato chiama una funzione di tipo **"process packet()"**. Uno dei parametri più significativi della funzione è il puntatore al pacchetto, **"process_packet"**.

```
if (pcap_loop
(handle, -1,process_packet,NULL)==-1){
fprintf (stderr,"%s",pcap_geterr(handle));
exit (1);
}
```

Si definiscono quindi le strutture degli header:

```
/* Ethernet header */
struct sniff_ethernet {
u_char ether_dhost [ETHER_ADDR_LEN];
u_char ether_shost [ETHER_ADDR_LEN];
u_short ether_type ;
};

/* IP header */
struct sniff_ip {
u_char ip_vhl ;
u_char ip_tos ;
u_short ip_len ;
u_short ip_id ;
u_short ip_off ;
```

```
# define IP_RF 0 x8000
# define IP_DF 0 x4000
# define IP_MF 0 x2000
# define IP_OFFMASK 0 x1fff
u_char ip_ttl ;
u_char ip_p ;
u_short ip_sum ;
struct in_addr ip_src, ip_dst ;
};

# define IP_HL(ip) (((ip)->ip_vhl)&0x0f)
# define IP_V(ip) (((ip)->ip_vhl)>>4)

/* UDP header */
struct sniff_udp {
u_short uh_sport ;
u_short uh_dport ;
u_short uh_ulen ;
u_short uh_sum ;
};
```

```
/* AODV header */
struct sniff_type {
u_int8_t tipo ;
};


/* RREQ */
struct sniff_rreq {
u_int8_t type ;
u_int8_t flags ;
u_int8_t res2 ;
u_int8_t hcnt ;
u_int32_t rreq_id ;
struct in_addr dest_addr ;
u_int32_t dest_seqno ;
struct in_addr orig_addr ;
u_int32_t orig_seqno ;
};

/* RREP */
struct sniff_rrep {
```

```
u_int8_t type ;
u_int8_t flags ;
u_int8_t res2 :3;
u_int8_t prefix :5;
u_int8_t hcnt ;
struct in_addr dest_addr ;
u_int32_t dest_seqno ;
struct in_addr orig_addr ;
u_int32_t lifetime ;
};

/* OLSR header */
struct sniff_olsr {
u_int16_t pkt_len ;
u_int16_t pkt_seqno ;
};

/* OLSR message header */
struct sniff_msg_header {
u_int8_t msg_type ;
u_int8_t vtime ;
```

```
u_int16_t msg_size ;
struct in_addr orig_addr ;
u_int8_t ttl;
u_int8_t hcount ;
u_int16_t msg_seqno ;
};
```

Si incrementa il puntatore per individuare l'inizio del pacchetto successivo:

```
ethernet = (struct sniff_ethernet*)(packet);
ip = (struct sniff_ip*)
(packet+SIZE_ETHERNET);
size_ip = IP_HL(ip)*4;
udp = (struct sniff_udp*)
(packet+SIZE_ETHERNET
+ size_ip);
```

Una volta che i bits del pacchetto sniffato sono associati alla struttura che lo definisce si può passare alla lettura dei vari campi.

Infine vengono chiamate due apposite funzioni per effettuare il cleaning delle risorse, esse sono:

```
pcap_freecode(&fp);
pcap_close(handle);
```

Packet Length		Packet Sequence Number	
Type	Vtime	Message Size	
Origin Address			
Ttl	Hops	Message Sequence Number	
Reserved		Htime	Willingness
Link Code	Reserved	Message Size	
Neighbor Address			
Lq	Nlq	Padding	
Link Code	Reserved	Message Size	
Neighbor Address			
Lq	Nlq	Padding	
Neighbor Address			
Lq	Nlq	Padding	

Figura A.15: Struttura pacchetto OLSR

A.5.2 Creazione pacchetti

La libreria utilizzata per la creazione di un pacchetto è la **Libnet-1.1.5**. Essa è stata modificata attraverso l'aggiunta degli headers dei protocolli AODV e OLSR. Grazie alla Libnet è pos-

sibile creare pacchetti sia a livello IP che a livello 2. Sono presenti molti protocolli e funzioni per assemblare i rispettivi header, ma nel caso in cui si dovesse costruire un header non definito nella libreria basta aggiungere nuovi packet builder con header di dimensione fissa. Vediamo quindi la procedura che porta alla creazione degli headers dei pacchetti utilizzati, in particolar modo RREQ e RREP per il protocollo AODV, HELLO e TC per OLSR.

Operazioni per creare un nuovo packet builder:
1)Stabilire la dimensione degli header ed editare il file *libnet-headers.h*:

```
#define LIBNET_AODV_RREQ_H 0x18
#define LIBNET_AODV_RREP_H 0x14
#define LIBNET_OLSR_TC 0x2c
#define LIBNET_OLSR_HL 0x34
```

2) Creare le strutture e aggiungerle in *libnet-headers.h* :

```
/* AODV RREQ */
```

```
struct libnet_aodv_rreq {
/* RREQ Flags : */
# define RREQ_JOIN 0x1
# define RREQ_REPAIR 0x2
# define RREQ_GRATUITOUS 0x4
# define RREQ_DEST_ONLY 0x8
# define RREQ_UNKNOWN_SEQNO 0x10
# define RREQ_GRP_REBUILD 0x20

u_int8_t type ;
u_int8_t j:1;
u_int8_t r:1;
u_int8_t g:1;
u_int8_t d:1;
u_int8_t u:1;
u_int8_t res1 :3;
u_int8_t res2 ;
u_int8_t hcnt ;
u_int32_t rreq_id ;
u_int32_t dest_addr ;
u_int32_t dest_seqno ;
```

```
u_int32_t orig_addr ;
u_int32_t orig_seqno ;
};

/* AODV RREP */
struct libnet_aodv_rrep {
u_int8_t type ;
u_int8_t r:1;
u_int8_t a:1;
u_int8_t res1 :6;
u_int8_t res2 :3;
u_int8_t prefix :5;
u_int8_t hcnt ;
u_int32_t dest_addr ;
u_int32_t dest_seqno ;
u_int32_t orig_addr ;
u_int32_t lifetime ;
};

/* OLSR HELLO */
struct neighbor {
```

```
uint32_t addr ;
uint8_t lq;
uint8_t nlq ;
uint16_t pad;
};

struct lq_hello_info_header {
uint8_t link_code;
uint8_t reserved;
uint16_t size;
struct neighbor neigh1;
struct neighbor neigh2;
struct neighbor neigh3;
};

struct lq_hello_header {
uint16_t reserved;
uint8_t htime;
uint8_t will;
struct lq_hello_info_header info_h;
};
```

```
struct olsr_msg_header {
uint8_t type;
uint8_t vtime;
uint16_t size;
  uint32_t orig;
  uint8_t ttl;
  uint8_t hops;
  uint16_t seqno;
  struct lq_hello_header hello_h;
};


struct libnet_olsr_hello {
  uint16_t olsr_packlen;
  uint16_t olsr_seqno;
  struct olsr_msg_header msg_h;
};

/* OLSR HELLO con neighbor con
```

```
 diversi link type */

struct libnet_olsr_hello_linktype {

u_int16_t pkt_len;
u_int16_t pkt_seqno;
u_int8_t type;
u_int8_t vtime;
u_int16_t msg_size;
u_int32_t orig_addr;
u_int8_t ttl;
u_int8_t hcount;
u_int16_t msg_seqno;
u_int16_t res1;
u_int8_t htime;
u_int8_t will;
u_int8_t link_code1;
u_int8_t res2;
u_int16_t link_msg_size1;
uint32_t neigh_addr1;
uint8_t link_type1;
```

```
  uint8_t neigh_type1;
u_int16_t pad1;
u_int8_t link_code2;
u_int8_t res3;
u_int16_t link_msg_size2;
uint32_t neigh_addr2;
uint8_t link_type2;
  uint8_t neigh_type2;
u_int16_t pad2;
uint32_t neigh_addr3;
uint8_t link_type3;
  uint8_t neigh_type3;
u_int16_t pad3;
};



/* OLSR TC */
struct libnet_olsr_tc {
u_int16_t pkt_len ;
```

```
u_int16_t pkt_seqno ;
u_int8_t type ;
u_int8_t vtime ;
u_int16_t msg_size ;
u_int32_t orig_addr ;
u_int8_t ttl;
u_int8_t hcount ;
u_int16_t msg_seqno ;
u_int16_t ansn ;
u_int16_t reserved ;
struct neighbor neigh1 ;
struct neighbor neigh2 ;
struct neighbor neigh3 ;
};
```

3) Aggiungere il Pblock identifier alla fine della lista dei vari identificatori presenti nel file libnet-structures.h prestando attenzione all'univocità dell'ID number.

```
#define LIBNET_PBLOCK_AODV_RREQ_H 0x43
```

```
#define LIBNET_PBLOCK_AODV_RREP_H 0x44
#define LIBNET_PBLOCK_OLSR_H 0x45
#define LIBNET_PBLOCK_OLSR_TC 0x46
#define LIBNET_PBLOCK_OLSR_HL 0x47
```

4) Definire le funzioni.

Tipicamente affinché si possano definire le funzioni utili per la creazione di un pacchetto bisogna creare un file in **"root/libnet/src"** ma nel caso specifico le varie funzioni sono state definite e richiamate ogni qualvolta necessitava nei vari attacchi.

Si porta come esempio quella che crea un messaggio di HELLO del protocollo OLSR.

```
/* Funzione che costruisce
 il messaggio di hello */
libnet_ptag_t libnet_build_olsr_hello
(u_int16_t pkt_seqno, u_int8_t vtime,
 u_int32_t orig_addr, u_int16_t msg_seqno,
 u_int8_t linkcode, u_int32_t neigh1,
```

```
u_int32_t neigh2, u_int32_t neigh3,
u_int8_t lq,u_int8_t nlq,
const uint8_t *payload,
uint32_t payload_s, libnet_t *l,
libnet_ptag_t ptag){

   uint32_t n;
   libnet_pblock_t *p;
   struct libnet_olsr_hello hello;

   if (l == NULL)
   {
   return (-1);}

   n = LIBNET_OLSR_H + payload_s;
   p = libnet_pblock_probe(l, ptag, n,
LIBNET_PBLOCK_OLSR_H);
   if (p == NULL){
       return (-1);}

   memset(&hello, 0, sizeof(hello));
```

```
    hello.olsr_packlen = htons(0x0030);
    hello.olsr_seqno = htons(pkt_seqno);
    (hello.msg_h).type = 0xc9 ;
    (hello.msg_h).vtime = vtime;
    (hello.msg_h).size = htons(0x002c);
    (hello.msg_h).orig = orig_addr;
    (hello.msg_h).ttl = 1;
    (hello.msg_h).hops = 0;
    (hello.msg_h).seqno = htons(msg_seqno);
    ((hello.msg_h).hello_h).reserved = 0;
    ((hello.msg_h).hello_h).htime = 134;
    ((hello.msg_h).hello_h).will = 7;
    (((hello.msg_h).hello_h).info_h)
.link_code = linkcode;
    (((hello.msg_h).hello_h).info_h)
.reserved = 0;
    (((hello.msg_h).hello_h).info_h)
.size = htons(0x001c);
    ((((hello.msg_h).hello_h).info_h)
.neigh1).addr = neigh1;
```

```
    ((((hello.msg_h).hello_h).info_h)
.neigh1).lq = lq;
    ((((hello.msg_h).hello_h).info_h)
.neigh1).nlq = nlq;
    ((((hello.msg_h).hello_h).info_h)
.neigh1).pad = 0x0000;

    ((((hello.msg_h).hello_h).info_h)
.neigh2).addr = neigh2;
    ((((hello.msg_h).hello_h).info_h)
.neigh2).lq = 254;
    ((((hello.msg_h).hello_h).info_h)
.neigh2).nlq = 255;
    ((((hello.msg_h).hello_h).info_h)
.neigh2).pad = 0x0000;

    ((((hello.msg_h).hello_h).info_h)
.neigh3).addr = neigh3;
    ((((hello.msg_h).hello_h).info_h)
.neigh3).lq = 222;
    ((((hello.msg_h).hello_h).info_h)
```

```
.neigh3).nlq = 215;
    ((((hello.msg_h).hello_h).info_h)
.neigh3).pad = 0x0000;


    n = libnet_pblock_append(l, p,
(uint8_t *)&hello, LIBNET_OLSR_H);
    if (n == -1)
    {
        goto bad;
    }

    LIBNET_DO_PAYLOAD(l, p);

    return (ptag ptag :
libnet_pblock_update
(l, p, 0, LIBNET_PBLOCK_OLSR_H));
    bad:
    libnet_pblock_delete(l, p);
    return (-1);
};
```

Inizializzazione della sessione.

Libnet si basa su tre concetti:

Context : é un opaque handle usato per mantenere lo stato della sessione durante la costruzione del pacchetto completo. Ci si riferisce al contex tramite una variabile di tipo "libnet_t".

Protocol blocks : sono dati interni di libnet definiti per ogni livello di rete.

Protocol tag : permette di riferirci a un protocol block tramite una variabile di tipo "libnet ptag t".

La prima cosa da fare consiste nel creare un context usando l'apposita funzione **"libnet init()"**, essa prende i seguenti parametri:

injection type : può assumere valori quali, LIBNET LINK o LIBNET RAW4, i quali indicano il livello di partenza della costruzione del pacchetto. Il primo per esempio indica che si sta partendo da un Link

Layer mentre il secondo indica che il livello di partenza è quello IPv4.

device : stringa che contiene il nome dell'interfaccia.

error buffer : buffer a cui mandare i messaggi di errore.

```
#include <libnet.h>
libnet_t *l;
char *device = "wlan0";
char errbuf[LIBNET_ERRBUF_SIZE];
l = libnet_init (LIBNET_RAW4,device,errbuf);
if (l == NULL){
fprintf (stderr, "Error opening context:
 %s", errbuf);
exit (1);
}
```

Costruzione dei protocol blocks.

Dopo aver creato il libnet context si può cominciare a costruire i vari protocol blocks, ricordando che devono essere costruiti in ordine

a partire dal livello più alto. Si susseguono le seguenti operazioni:
- dichiarazione dei protocol tags

```
libnet_ptag_t ipv4=0, udp=0, rreq=0;
```

- si chiamano le funzioni per definire gli headers

```
libnet_build_aodv_rreq();
libnet_build_udp();
libnet_build_ipv4();
```

Invio dei pacchetti e chiusura.

Una volta costruiti i vari protocol blocks attraverso l'utilizzo della funzione "libnet_write()" si assemblano e si inviare il pacchetto in rete.

```
if ((libnet_write (l)) == -1){
fprintf (stderr, "Unable to send
packet: %s",libnet_geterror (l));
exit (1);}
```

Pulitura della memoria.

```
libnet_destroy (l);
```

A.5.3 Accodamento dei pacchetti

I pacchetti accodati vengono trattati attraverso l'utilizzo della libreria **"Libnetfilter_queue-1.0.1"**. Nella fase di inizializzazione tramite la funzione **"nfq_open()"** viene creato un NFQUEUE handler, le due funzioni chiamate successivamente servono per dire al kernel che la coda è gestita dal NFQUEUE handler per il protocollo selezionato.

```
struct nfq_handle *h;
h = nfq_open();
if (!h) {
printf( "error during nfq_open ");

if (nfq_unbind_pf
(h, PF_INET)< 0){
printf( "error during nfq_unbind_pf");
exit(1);}
if (nfq_bind_pf
(h, PF_INET) < 0){
```

```
printf( "error during nfq_bind_pf");
exit(1);
}
```

Quindi si lega il programma a una specifica coda e si imposta quale parte del pacchetto deve essere copiata nello userspace.

```
struct nfq_q_handle *qh;
qh = nfq_create_queue(h,0, &cb, NULL);
if (!qh) {
printf( "error during nfq_create_queue");
exit(1);}
if (nfq_set_mode(qh,NFQNL_COPY_PACKET,
0xffff)< 0){
printf( "can't set packet_copy mode");
exit(1);
}
```

L'accodamento dei pacchetti avviene in maniera iterativa:

```
int rv;
```

```
char buf[4096];
int fd = nfq_fd(h);
while ((rv=recv(fd, buf, sizeof(buf),
 0))>= 0) {
printf("pkt received");
nfq_handle_packet(h, buf, rv);}
```

La funzione **"nfq create queue"** per ogni pacchetto accodato chiama una funzione di **"callback cb()"** nella quale viene presa la decisione sul pacchetto attraverso l'utilizzo della **"nfq_set_verdict()"**. Quest'ultima prende in ingresso

ID : valore univoco assegnato da netfilter ad ogni pacchetto.

Verdict : l'azione (NF DROP, NF ACCEPT o NF QUEUE) che l'utente decide di effettuare sul pacchetto.

```
u_int32_t idpkt (struct nfq_data *tb) {
int id = 0;
struct nfqnl_msg_packet_hdr *ph;
```

```
struct nfqnl_msg_packet_hw *hwph;
ph = nfq_get_msg_packet_hdr(tb);
if (ph)
id = ntohl(ph->packet_id);
return id;}
int cb(struct nfq_q_handle
 *qh,struct nfgenmsg *nfmsg,
struct nfq_data *nfa, void *data) {
u_int32_t id = idpkt(nfa);
return nfq_set_verdict(qh, id,
NF_ACCEPT, 0, NULL);
}
```

Anche in questo caso bisogna liberare la memoria e per far questo di utilizza la **"nfq close(h)"**.

Bibliografia

[1] F. De Rosa, V. Di Martino, L. Paglione, and M. Mecella. (2002) *Mobile Adaptive Information Systems on MANET: What We Need as Basic Layer? In Proceedings of the 1st Workshop on Multichannel and Mobile Information Systems (MMIS'03), IEEE*, Rome.

[2] Krishna Gorantala. (2006) *Routing protocols in mobile ad-hoc networks. Master's thesis* Umea University.

[3] rfc 3626 OLSR
http:// www.ietf.org/rfc/rfc3626.txt

[4] rfc 3561 AODV
http:// www.ietf.org/rfc/rfc3561.txt

[5] Pradip M. Jawandhiya, Mangesh M. Ghonge, DR. M.S.Ali, and Prof. J.S. Deshpande. (2010) *A survey of mobile ad-hoc network attacks. International Journal of Engineering Science and Technology*

[6] Rashid Hafeez Khokhar, Md AsriNgadi and Satria Mandala.(2008) *A review of current routing attacks in mobile ad-hoc networks. International Journal of Computer Science and Security*

[7] S.A.Razak, S.M.Furnell, and P.J.Brooke.(2004) *Attacks against mobile ad-hoc networks routing protocols. In Proceedings of 5th Annual Postgraduate Symposium on the Convergence of Telecomunications, Networking & Broadcasting*

[8] Ali Ghaffari.(2006) *Vulnerability and security of mobile ad-hoc networks.Proceedings*

of the 6th WSEAS International Conference on Simulation, Modelling and Optimization

[9] Po-Wash Yau, Shenglan Hu, and Chris J. Mitchell.(2009) *Malicious attacks on ad-hoc network routing protocols. International Journal of Computer Research*

[10] Jane Zhen and Sampalli Srinivas.(2003) *Preventing replay attacks for secure routing in ad-hoc networks. Lecture Notes in Computer Science*

[11] Wikipedia. *http:// it.wikipedia.org*

[12] Imad Aad, Jean-Pierre Hubaux, and Edward W. Knightly. (2008) *Impact of denial of service attacks on ad-hoc networks. IEEE/ACM TRANSACTIONS ON NETWORKING*

[13] Yin-Chun Hu, and Adrian Perrig. (2006) *Wormhole attacks in wireless networks. IEEE JOURNAL ON SELECTED AREAS IN COMUNICATIONS*

[14] Brahim Bensaou, and Tarik Taleb. (2008) *Detecting and avoiding wormhole attacks in wireless ad hoc networks. Communications Magazine, IEEE*

[15] Abhay Kumar Rai, Rajiv Ranjan Tewari, and Sauranbh Kant Upadhyay. (2010) *Different types of attacks on integrated manet-internet communication.International Journal of Computer Science and Security*

[16] Yin-Chun Hu, David B.Johnson, and Adrian Perrig. (2003) *Rushing attacks and defense in wireless ad-hoc network routing protocols.WiSe '03 Proceeding of the 2nd ACM workshop on Wireless security*

[17] Jabel Ben-Othman and Ali Hamieh.(2009) *Defending method against attack in wireless ad-hoc networks. The 5th IEEE International Workshop on Performance and Management of Wireless and Mobile Networks*

[18] Andreas Tonnesen, Andreas Hafslund,

and Oivind Kure. *The Unik-OLSR plugin library*

[19] Andreas Tonnesen, Andreas Hafslund, Roar Bjorgum Rotvik, Jon Andersson and Oivind Kure. (2004) *Secure Extension to the OLSR protocol*

[20] Cedric Adjih, Thomas Clausen, Philippe Jacquet, Anis Laouiti, Paul Muhlethaler, Daniele Raffo. *Securing the OLSR protocol*

[21] A.M.Hengland, P.Spilling, L. Nilsen and O.Kure. *Hybrid Protection of OLSR*

[22] Fan Hong, Lian Hong, Cai Fu. (2005) *Secure OLSR. AINA '05. IEEE*

[23] Bounpadith Kannhavong, Hidehisa Nakayama, Yoshiaki Nemoto, Nei Kato, Abbas Jamalipour.(2008) *SA-OLSR: Security Aware Optized Link State Routing for Mobile Ad Hoc Networks.IEEE*