



DIPARTIMENTO DI INGEGNERIA INFORMATICA, MODELLISTICA,
ELETTRONICA E SISTEMISTICA

Corso di Laurea Magistrale in Ingegneria Informatica

Relazione per il Progetto del corso Network Security

Analisi di sicurezza per reti SDN e Host Tracking Service con controller ONOS

Professori:

**Floriano De Rango
Mattia Giovanni Spina**

Gruppo:

**Carreri Domenico, 235184
Cavallo Giuseppe, 235328**

ANNO ACCADEMICO 2023/2024

Indice

Indice	1
1 Introduzione	2
1.1 Funzionamento Generale di una SDN	3
1.2 Tecnologie Utilizzate	7
1.2.1 ONOS	7
1.2.2 Mininet	10
2 Attacchi all’Host Tracking Service di una SDN	14
2.1 HTS Poisoning	17
2.2 ARP Poisoning	18
2.3 Man In The Middle su API Rest di ONOS	20
3 Contromisure e Valutazione Costi-Benefici	24
3.1 Detection App 1	25
3.2 Detection App 2	29
3.3 Safe Proxy sull’host nat	31
3.4 Valutazione delle Mitigazioni Proposte	34
4 Conclusione	44

Capitolo 1

Introduzione

Nel contesto delle reti tradizionali, la gestione e il monitoraggio del traffico dati, così come l'identificazione dei dispositivi connessi e la definizione dei percorsi di routing, sono operazioni effettuate mediante algoritmi specifici installati su apparati dedicati. Quando un pacchetto arriva a un dispositivo di routing, viene eseguito un insieme di regole predefinite nel firmware per stabilire la sua destinazione e il percorso da seguire. Nei dispositivi più moderni è possibile configurare tali operazioni anche in base al tipo e al contenuto dei pacchetti, con l'assegnazione di priorità ai diversi flussi di dati.

Tuttavia, questo approccio presenta delle limitazioni significative. La capacità dei dispositivi di rete di gestire grandi volumi di traffico è spesso insufficiente, il che può compromettere le prestazioni generali della rete. Inoltre, gli amministratori sono costretti a configurare manualmente protocolli e regole complesse all'interno di ambienti eterogenei, caratterizzati da una varietà di dispositivi con specifiche e produttori diversi. Un errore critico in questo contesto può richiedere un lungo tempo di risoluzione, incidendo notevolmente sulle prestazioni della rete.

Le reti tradizionali, sviluppatesi con l'espansione di Internet, sono solitamente organizzate in una struttura gerarchica, ereditata dal modello client-server considerato ormai obsoleto. Questa configurazione contribuisce a una staticità che non si adatta alla dinamicità delle reti moderne. Per esempio, una modifica a un singolo elemento della rete può richiedere una riconfigurazione manuale di numerosi altri dispositivi, come router e switch, complicando ulteriormente la gestione della rete.

In risposta a queste limitazioni, sono emersi negli ultimi anni modelli di rete innovativi conosciuti come *Software Defined Networking* (SDN). Le reti SDN offrono una maggiore flessibilità e centralizzazione nella gestione del traffico, permettendo una configurazione più dinamica e una risposta più rapida ai cambiamenti, superando molte delle rigidità e delle inefficienze delle reti tradizionali.

Il lavoro progettuale descritto di seguito è un'analisi di sicurezza delle reti SDN, con proposte di attacchi e di possibili contromisure attuabili, considerando i relativi costi in termini temporali e computazionali che le mitigazioni introducono. L'analisi è stata svolta su una rete SDN avente un unico controller *ONOS*, nonostante sia consentito utilizzarne di più, per semplificare il lavoro svolto. Per simulare le topologie di rete è stato utilizzato invece *Mininet*.

1.1 Funzionamento Generale di una SDN

Con Software Defined Networking (SDN) si indica un modello di rete che porta al superamento di molte delle limitazioni delle reti tradizionali, alcune delle quali descritte anche nel paragrafo precedente.

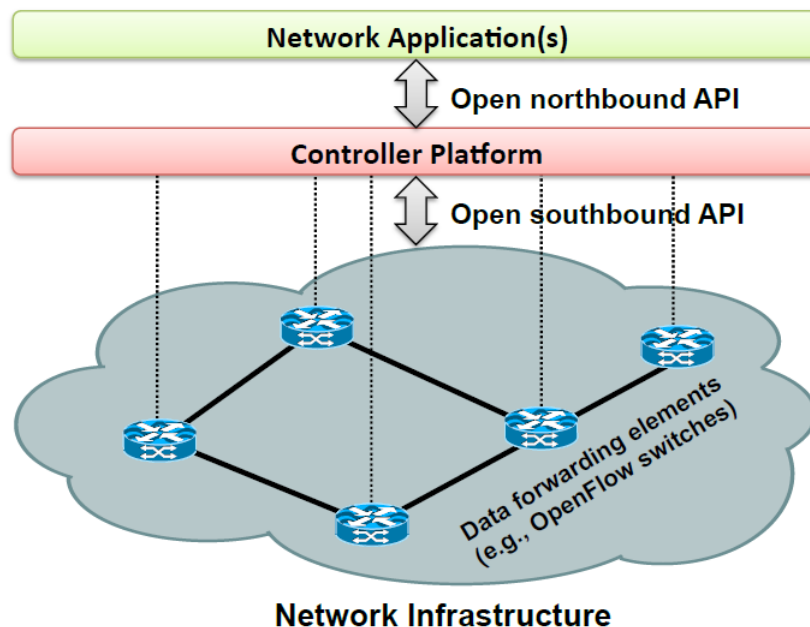


Figura 1.1: Architettura di una SDN.

Come mostrato nell'immagine precedente, in una SDN il piano di controllo (control layer) è distinto dal piano dati (data plane). La differenza principale rispetto alle reti tradizionali è che la gestione del traffico di rete avviene attraverso uno o più computer dedicati nella rete, piuttosto che attraverso dispositivi di rete interconnessi. Questi computer dedicati amministrano dinamicamente la rete come se fosse un'entità logica, utilizzando applicazioni specifiche e altamente programmabili. In questo modello, i dispositivi di commutazione, come router e switch, fungono semplicemente da hardware per l'inoltro dei pacchetti, mentre i computer del piano di controllo hanno una visione complessiva dello stato della rete.

Dato che router e switch utilizzano interfacce di controllo create appositamente dai produttori per consentire la comunicazione tra i controller e gli altri dispositivi di rete, è stato necessario sviluppare un'interfaccia di comunicazione universale. Attualmente, lo standard per le Southbound Interfaces è rappresentato dal protocollo OpenFlow, altamente programmabile. Questo protocollo si basa sul concetto di flusso, che definisce porzioni specifiche di traffico con regole prestabilite e memorizzate in tabelle di flusso (flow table), al fine di eseguire azioni personalizzate.

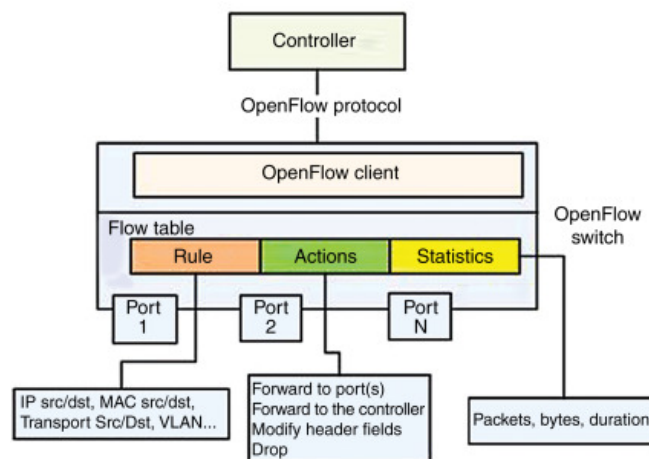


Figura 1.2: Protocollo OpenFlow.

Anche se OpenFlow è di gran lunga il protocollo più comune per la gestione di reti definite dal software, non è l'unico: NETCONF (RFC 6241), BGP (Border Gateway Protocol), XMPP (Extensible Messaging and Presence Protocol), OVSDB (Open vSwitch Database Management Protocol) e MPLS-TP

(MPLS Transport Profile) sono alternative che non sostituiscono completamente il protocollo standard, ma che possono comunque giocare un ruolo decisivo nell'implementazione di reti definite dal software. In alcune architetture vengono utilizzati anche protocolli proprietari di Cisco Systems e Nicira.

Protocollo OpenFlow All'interno di uno switch compatibile con OpenFlow, possiamo individuare questi componenti principali:

- Una o più tabelle di flusso (flow table)
- Una tabella di gruppi (group table)
- Uno o più canali OpenFlow (OpenFlow channel)

Il protocollo OpenFlow svolge un ruolo cruciale consentendo al controller di apportare modifiche, aggiungere nuove voci o eliminare quelle esistenti nelle tabelle degli switch. Questi aggiornamenti e modifiche sono trasmessi attraverso il canale OpenFlow, che facilita lo scambio di messaggi e la gestione dei pacchetti tra il controller e gli switch.

I messaggi OpenFlow possono essere classificati in tre categorie principali:

1. **Messaggi Controller-to-Switch:** Questi messaggi sono inviati dal controller agli switch senza richiedere una risposta da parte dello switch. Sono comunemente utilizzati per aggiornare le configurazioni degli switch, modificare voci specifiche nelle tabelle di flusso e gestire i pacchetti in arrivo (packet-out).
2. **Messaggi Asincroni:** Inviati dagli switch ai controller senza una richiesta preventiva da parte di quest'ultimo. Questa categoria include i messaggi packet-in, che riguardano pacchetti per i quali non esiste una corrispondenza nelle tabelle di flusso. Inoltre, possono includere aggiornamenti riguardanti modifiche alle porte dello switch o alla rimozione di voci dalle tabelle.
3. **Messaggi Simmetrici:** Questi sono messaggi bilaterali utilizzati per stabilire e mantenere la connessione tra il controller e lo switch, o per verificare la connessione attiva.

Quando uno switch OpenFlow riceve un pacchetto, esso viene elaborato tramite una pipeline. Questo processo implica confronti tra i campi del pacchetto e quelli presenti nelle tabelle di flusso. Se ci sono più tabelle di flusso, esse vengono ordinate per dare priorità ai confronti con i pacchetti. Quando viene trovata una corrispondenza, le azioni specificate per quel match vengono eseguite. Se non viene trovata alcuna corrispondenza, il pacchetto viene associato a una voce di table-miss, e si decide se scartarlo o inoltrarlo al controller.


Ogni voce all'interno di una tabella di flusso è composta dai seguenti campi:

- **Match Fields:** Valori utilizzati per verificare la corrispondenza del pacchetto in arrivo, che includono:
 - Porta di ingresso
 - Header del pacchetto
 - Eventuali metadati da altre tabelle (se è presente un meccanismo di pipeline)
- **Counters:** Contatori incrementati solo se c'è una corrispondenza del pacchetto.
- **Priority:** Valore numerico usato per risolvere conflitti tra più corrispondenze di pacchetti.
- **Instructions:** Azioni da eseguire se si verifica un match del pacchetto.
- **Time Out:** Tempo di permanenza della voce nella tabella prima che venga rimossa.
- **Cookie:** Valori utilizzati dal controller per raccogliere informazioni statistiche.
- **Flags:** Valori utilizzati per gestire in modo diverso le voci della tabella.

Per garantire il corretto funzionamento delle comunicazioni, ogni switch OpenFlow dispone di un numero specifico di porte. Ogni pacchetto è caratterizzato da una porta di ingresso e una porta di uscita, quest'ultima determinata attraverso il processo di pipeline.

La figura seguente mostra un esempio di regola installata su uno switch della rete a seguito di un ping tra due host.

Flows for Device of:0000000000000001 (4 Total)



STATE	PACKETS	DURATION	FLOW PRIORITY	TABLE NAME	SELECTOR	TREATMENT	APP NAME
Added	19	60	40000	0	ETH_TYPE:ldp	imm[OUTPUT:CONTROLLER], cleared:true	*core
Criteria: ETH_TYPE:ldp Treatment Instructions: imm[OUTPUT:CONTROLLER], cleared:true							
Added	19	60	40000	0	ETH_TYPE:bdp	imm[OUTPUT:CONTROLLER], cleared:true	*core
Criteria: ETH_TYPE:bdp Treatment Instructions: imm[OUTPUT:CONTROLLER], cleared:true							
Added	34	60	40000	0	ETH_TYPE:arp	imm[OUTPUT:CONTROLLER], cleared:true	*core
Criteria: ETH_TYPE:arp Treatment Instructions: imm[OUTPUT:CONTROLLER], cleared:true							
Added	35	60	5	0	ETH_TYPE:ipv4	imm[OUTPUT:CONTROLLER], cleared:true	*core
Criteria: ETH_TYPE:ipv4 Treatment Instructions: imm[OUTPUT:CONTROLLER], cleared:true							

Figura 1.3: Flows OpenFlow.

1.2 Tecnologie Utilizzate

1.2.1 ONOS

La Open Networking Foundation è una organizzazione no-profit che è stata fondata nel 2011 per promuovere standard e tecnologie relativi alle reti SDN. Essa usa un modello open-source e i suoi prodotti consistono in reti wireless e di telecomunicazioni, datacenters e altre aree del networking.

Questa organizzazione include al suo interno molte compagnie famose tra cui Google, Microsoft AT&T, Cisco, Intel, NVIDIA e molte altre. Nello stesso anno la fondazione ha iniziato a promuovere il disaccoppiamento dei pannelli di controllo e di forwarding introducendo e supportando concetti SDN; nell'anno successivo invece hanno reso open source il Openflow che è il primo standard protocollare che abilita controller di rete remoti alla gestione dei nodi nel pannello di controllo. Nel 2014 la fondazione ha rilasciato la prima versione open source dell'Open Networking Operating System (ONOS), il principale controller open source per SDN. Il codice di ONOS è disponibile sulla piattaforma GitHub.

Esso è principalmente scritto in Java ma all'interno del progetto sono state utilizzate tante altre tecnologie come Python, OSGi, file Bazel, e tanti altri. Per ONOS il termine di controller SDN può essere un po' ambiguo in quanto esso è un vero e proprio sistema operativo e come tale usa un kernel e delle applicazioni installabili. Come livello base ONOS utilizza Apache Karaf che è un application container polimorfico, questo significa che è una tecnologia perché permette la distribuzione di vari servizi e componenti nello stesso ambiente dando loro la possibilità di comunicare e condividere risorse. Esso usa

il concetto di "run anywhere", che significa che può essere utilizzato su diverse tipologie di hardware attraverso Java, immagini Docker oppure istanze cloud. L'immagine 1.4 mostra l'architettura logica di ONOS in cui possiamo vedere i seguenti componenti:

1. **Applicazioni:** Le applicazioni sono bundles OSGi che possono essere attivate o disattivate, sia all'avvio del sistema che a tempo di runtime usando la console di Karaf, e forniscono e implementano servizi che vanno dalla visualizzazione delle topologie di rete attraverso un browser web all'integrazione di prodotti ONOS, come Kubernetes e Apache Kafka, per impostare regole di flusso per un traffico di rete personalizzato.
2. **Northbound API:** La Northbound API è l'interfaccia utilizzata dalle applicazioni e dal Core component per comunicare. Le applicazioni possono ricevere avvisi e notifiche dal Core, ascoltare eventi riguardante uno specifico argomento, sovrascrivere valori nei sottocomponenti (come data stores) del Core e richiedere risorse aggiuntive.
3. **Core:** Il Core è il componente principale, ospita data stores e servizi per gestire i vari sottosistemi come Dispositivi, Collegamenti, Host e molto altro. Esso inoltre è un componente distribuito.
4. **Southbound API:** Similmente alla Northbound, la Southbound API è l'interfaccia che permette al Core di comunicare con i nodi della rete che stanno nel data-plane. Il core può ricevere notifiche e avvisi dalla rete, ascoltare eventi riguardanti specifici argomenti e prendere decisioni sulla base dei messaggi che gli arrivano da data-plane.
5. **Providers:** I Providers sono componenti in grado di riconoscere i protocolli per l'interazione con i dispositivi di rete. Loro possono tradurre i dati che arrivano dalla rete in messaggi interpretabili dal Core.

Quando si parla di servizi in ONOS si fa riferimento a costrutti verticali che attraversano i componenti che descrivono la logica orizzontale descritta nella Figura 1.4. Ad esempio il Device service è un costrutto verticale che attraversa tutti i componenti descritti in precedenza e si occupa della gestione dei dispositivi all'interno della rete.

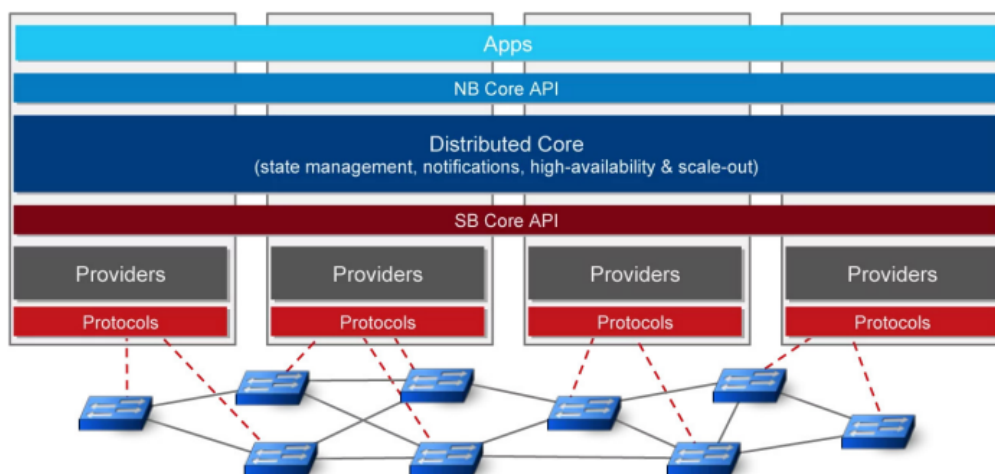


Figura 1.4: Architettura ONOS.

Il concetto di SDN è stato introdotto parlando anche di centralizzazione delle informazioni, tuttavia come si diceva in precedenza ONOS è composto da un Core Distribuito. Questa è una funzione importante introdotta da ONOS in quanto permette di utilizzare per la stessa rete controller differenti in modo da acquisire le proprietà di *Fault Tolerance* e di *Disponibilità dell'Ambiente*.

1.2.2 Mininet

Durante lo sviluppo del progetto il data-plane e le reti utilizzate sono state virtualizzate attraverso Mininet. Mininet è un emulatore di rete open source molto utilizzato per effettuare:

- **Testing**
- **Ricerca**
- **Prototipazione**
- **Debugging**

È possibile trovare il suo codice sulla piattaforma GitHub.

Attualmente mininet conta più di sessanta contributori, distribuiti utilizzando la licenza BSD open source con 6 versioni ufficiali. Supporta inoltre diversi meccanismi di installazione tra i quali il più semplice è quello di utilizzare una macchina virtuale Ubuntu pre-configurata. Sul README della pagina del progetto GitHub gli sviluppatori spiegano Mininet attraverso una definizione molto semplice:

Mininet crea delle reti virtuali utilizzando un processo di virtualizzazione basato sui processi e sul network namespaces, entrambe funzionalità rese disponibili di recente dai kernel Linux. In Mininet gli hosts vengono emulati come processi bash eseguiti nello spazio dei nomi della rete in modo che ogni programma o codice che normalmente può essere eseguito su kernel Linux può essere eseguito anche su un Host in Mininet. Ogni host a sua volta ha una propria interfaccia di rete privata e può vedere solo i suoi processi. Gli switch sono software-based come gli Open vSwitches oppure gli Switches OpenFlow. I links sono coppie di collegamenti Ethernet che vivono direttamente nel kernel e connettono gli switches con gli hosts.

Essendo open source, detiene una community attiva e di un continuo mantenimento per cui gode di molti vantaggi tra cui un'avvio molto più veloce rispetto

alle sue alternative, è poco costoso, sempre disponibile ed è velocemente riconfigurabile e riavviabile. Oltre ai vantaggi però soffre anche di qualche limitazione, ad esempio, le reti virtualizzate non possono superare la capacità della CPU e la larghezza di banda disponibile su un singolo server e non possono eseguire switches OpenFlow non compatibili con Linux. Dalla descrizione fornita dagli sviluppatori si possono trarre le seguenti considerazioni che Mininet ha in comune con le tecnologie basate su Kernel Linux:

- **Bash processes:** I nodi vengono generati eseguendo processi demoni indipendenti.
- **Network namespaces:** Una macchina Linux based condivide un singolo set di risorse di rete, come dispositivi di rete, indirizzi IP e tabelle di routing ARP, protocolli IPv4 e IPv6, regole di firewall e la directory /proc/net. È possibile modificare queste impostazioni che però verranno rese visibili all'interno dell'intero sistema operativo ONOS. Il network namespace fornisce inoltre, metodi di isolamento in cui è possibile definire risorse di rete isolate.
- **Coppie di link Virtual Ethernet (veth):** Per creare collegamenti di rete sono usate coppie di link Ethernet virtuali. Questa tecnologia può essere vista come una sorta di tubo che esce da una estremità ed entra in nell'altra e viceversa.

Tra le varie funzionalità messe a disposizione da mininet le più importanti sono:

1. Un launcher da linea di comando per istanziare la rete (**mn**).
2. Una comoda API Python per la creazione di reti di diverse topologie e dimensioni.
3. Tramite la sottoclasse Topo si possono creare topologie parametrizzate utilizzando oggetti Mininet .

Avviata la topologia Mininet mette a disposizione una Command Line Interface (**Mininet CLI**) utile nel testing di esecuzione della rete. Tramite il

comando *help* ad esempio di possono vedere tutti i comandi disponibili. Il comando *nodes* permette di vedere tutti i nodi (hosts, controllers e switches) presenti nella rete, tramite *links* si possono vedete tutti i collegamenti. Per testare la connettività tra due hosts si può utilizzare il comando di ping. Ad esempio se vogliamo testare la connettività tra l'host 'h1' e l'host 'h2' il comando apparirà in questo modo: *h1 ping -c 1 h2*. A tale scopo per semplificare la connettività tra tutti gli hosts, Mininet mette a disposizione il comando *pingall*.

Oltre alla Mininet CLI, l'emulatore mette anche a disposizione delle **API Python** per interagire con le sue funzionalità chiave. Queste API mettono a disposizione tutto il necessario per la creazione di reti complesse ed utilizzare tutte le funzionalità messe a disposizione dalla Mininet CLI. Esistono tre differenti livelli di APIs Mininet: APIs di basso livello, APIs di livello intermedio e APIs di alto livello. Le APIs di basso livello permettono di definire classi come Host, Switch e Link e sono molto utilizzate quando si vuole avere una chiara idea di ciò che accade nella rete oppure quando si vuole avere un controllo rigoroso sui dispositivi. Le APIs di livello intermedio aggiungono la classe Mininet che possiede le funzioni per gestire in maniera corretta i nodi della rete (fanno parte di questa classe i metodi addHost, addSwitch, addLink...) e per effettuare operazioni di avvio e di chiusura della rete. Le APIs di alto livello invece aggiungono la classe Topo che fornisce per creare topologie riutilizzabili e/o parametrizzate. Insieme alle APIs di livello medio sono quelle più utilizzate nella creazione di grandi topologie di rete.

La Figura 1.5 mostra un esempio di utilizzo delle APIs di basso livello

```

1 h1 = Host( 'h1' )
2 h2 = Host( 'h2' )
3 s1 = OVSSwitch( 's1', inNamespace=False )
4 c0 = Controller( 'c0', inNamespace=False )
5 Link( h1, s1 )
6 Link( h2, s1 )
7 h1.setIP( '10.1/8' )
8 h2.setIP( '10.2/8' )
9 c0.start()
10 s1.start( [ c0 ] )
11 print( h1.cmd( 'ping -c1', h2.IP() ) )
12 s1.stop()
13 c0.stop()

```

Figura 1.5: Uso APIs di basso livello.

La Figura 1.6 mostra un esempio di utilizzo delle APIs di livello medio La

```

1 net = Mininet()
2 h1 = net.addHost( 'h1' )
3 h2 = net.addHost( 'h2' )
4 s1 = net.addSwitch( 's1' )
5 c0 = net.addController( 'c0' )
6 net.addLink( h1, s1 )
7 net.addLink( h2, s1 )
8 net.start()
9 print( h1.cmd( 'ping -c1', h2.IP() ) )
10 CLI( net )
11 net.stop()

```

Figura 1.6: Uso APIs di livello medio.

Figura 1.7 mostra un esempio di utilizzo delle APIs di alto livello

```

1 class SingleSwitchTopo( Topo ):
2     "Single Switch Topology"
3     def build( self, count=1 ):
4         hosts = [ self.addHost( 'h%d' % i ) for i in range( 1, count + 1 ) ]
5         s1 = self.addSwitch( 's1' )
6         for h in hosts:
7             self.addLink( h, s1 )
8
9 net = Mininet( topo=SingleSwitchTopo( 3 ) )
10 net.start()
11 CLI( net )
12 net.stop()

```

Figura 1.7: Uso APIs di alto livello.

Capitolo 2

Attacchi all'Host Tracking

Service di una SDN

Le SDN sono architetture di rete pensate per semplificare e migliorare la gestione delle reti, grazie ad un'elevata flessibilità dovuta alla separazione tra piano di controllo e piano dei dati. Questo nuovo paradigma tuttavia non è privo di preoccupazioni legate alla sicurezza interna dei controller. È stato dimostrato infatti che per degli host malevoli è possibile sfruttare le vulnerabilità interne dei controller e creare danni alla topologia di rete.

In una rete SDN, le *locations*¹ di tutti gli host possono essere monitorate attraverso l'*Host Tracking Service*, un servizio che ispeziona i messaggi di tipo *Packet-In* che il controller riceve dagli switch. Se un host effettua uno spostamento su un'altra location (ad esempio si scollega dallo switch su cui è attualmente connesso per ricollegarsi alla rete su un altro switch), ed invia un pacchetto dalla sua nuova posizione, l'HTS è in grado di riconoscere lo spostamento poiché rileva il *Packet-In* che incorpora il pacchetto inviato dall'host, da cui è in grado di estrarre le informazioni e riconoscere il mittente, se è un host conosciuto.

Questo meccanismo presenta tuttavia un grave **problema**: la totale assenza di meccanismi di *autenticazione* degli hosts. Di conseguenza, un attaccante può facilmente impersonare un host vittima o in generale condurre degli attacchi pericolosi sulla rete. Lo scenario appena descritto prende il nome di *Cross App Poisoning*.

¹Combinazione tra id e porta dello switch a cui un host risulta connesso

Cross App Poisoning Da un punto di vista generale, un Cross App Poisoning (CAP) è un attacco che sfrutta le relazioni esistenti tra due entità aventi diversi livelli di privilegi. La finalità dell'attacco è sfruttare i limitati permessi di un'applicazione per indurne un'altra, avente invece privilegi elevati, a compiere azioni non consentite o non previste. È un tipo di attacco comune in quei sistemi ad elevata complessità e che hanno tanti componenti che necessitano di comunicare tra loro per rendere la rete efficiente, come appunto le SDN.

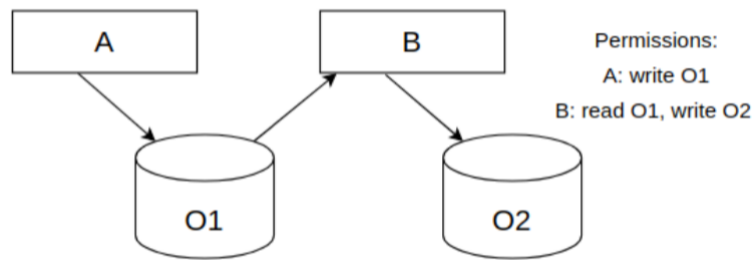


Figura 2.1: Schema generale di un Cross App Poisoning.

Il lavoro progettuale è stato svolto con la finalità di dimostrare la pericolosità dei vettori di attacco di tipo CAP (detti gadgets) per il controller ONOS. Sebbene ne esistano di molteplici, ai fini del progetto ci si è dedicati ai Cross App Poisoning aventi come target l'Host Tracking Service. Si considerino le applicazioni presenti in ONOS di default *Host Location Provider* (l'implementazione offerta da ONOS dell'Host Tracking Service) e *Reactive Forwarding* (applicazione avente il permesso di installare flow rules dinamicamente quando si verifica una 'flow rule miss').

Facendo riferimento all'immagine 2.1:

- A è rappresentato dall'Host Location Provider
- O1 è rappresentato dall'Host Data Store
- B è rappresentato dal Reactive Forwarding
- O2 è rappresentato dalle tabelle di flusso degli switch

Questa coppia di applicazioni rappresenta un CAP gadget; esiste infatti una relazione tra le due, poiché Reactive Forwarding utilizza le informazioni contenute nell'*Host Data Store*² per installare le regole di flusso, mentre l'Host

²Struttura dati contenente le informazioni relative agli hosts connessi

Location Provider invece è il responsabile di ciò che viene scritto nell'Host Data Store. Ne deriva quindi che trovando un modo di avvelenare il contenuto dell'Host Data Store è possibile indurre l'applicazione di Reactive Forwarding (fwd) a installare regole di flusso scorrette, impedendo di effettuare il routing dei pacchetti corretto.

Il modo per fare ciò esiste ed è una conseguenza di quanto detto nell'introduzione di questo capitolo: la totale assenza di meccanismi di autenticazione infatti consente ad un host malintenzionato di generare pacchetti malformati in grado di trarre in inganno l'Host Location Provider, portandolo a inserire informazioni scorrette nell'Host Data Store. Queste informazioni scorrette permetteranno all'attaccante, con l'ausilio involontario del Reactive Forwarding, di installare regole di flusso a suo piacere.

Saranno presentati ora diversi attacchi che sfruttano il meccanismo di Cross App Poisoning appena descritto. Ciascuno degli attacchi presentato considera per semplicità la stessa topologia statica, in particolare:

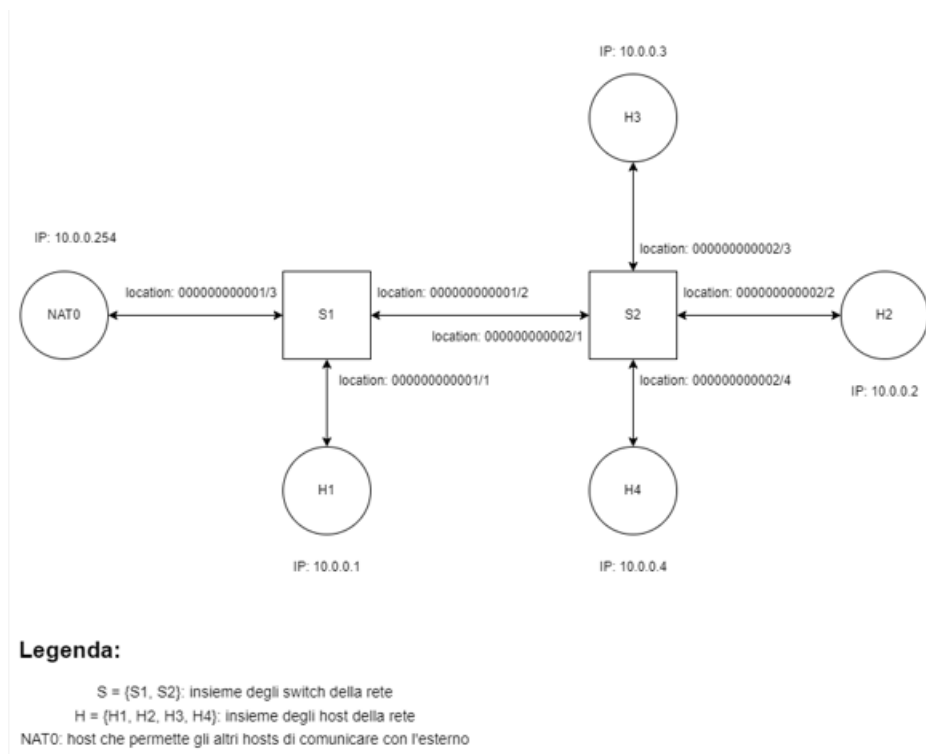


Figura 2.2: Topologia scelta per gli attacchi.

2.1 HTS Poisoning

Come detto anche in precedenza, a causa dell'assenza di validazione e autenticazione, l'Host Tracking Service accetta ogni genere di aggiornamento dell'Host Profile (quello che in ONOS è stato definito Host Data Store). Sfruttando questa debolezza, un attaccante è in grado di modificare la location di un altro host, e a seconda del tipo di modifica, realizzare diverse tipologie di attacchi, come l'**Host Impersonation**, il **Denial of Service** o il **Fake Host Generation**.

Per realizzare gli attacchi appena elencati, è sufficiente per un attaccante creare dei pacchetti fasulli aventi come indirizzi IP e MAC sorgenti quelli della vittima e inviarli ad un altro host nella rete; la differenza tra i tre attacchi consiste nel contenuto del pacchetto che si sta inviando.

Ad esempio, considerando la topologia in 2.2, si supponga l'attaccante sia l'host H2. Utilizzando un semplice script Python realizzato tramite la libreria *Scapy*, H2 è in grado di generare dei pacchetti ICMP personalizzati e realizzare l'attacco inviandoli ad un altro host nella rete, ad esempio H1, o persino al controller stesso.

```
1#!/usr/bin/python3
2import scapy.all as scapy
3
4
5"""----- Sezione costanti e variabili di utilità -----"""
6
7source_ip = "10.0.0.3"
8destination_ip = "10.0.0.1"
9source_mac = "00:00:00:00:00:03"
10
11"""----- Fine sezione costanti e variabili di utilità -----"""
12
13"""----- Sezione metodi di utilità -----"""
14
15
16def invio_ping(src_ip, dst_ip, src_mac):
17    livello_dataLink = scapy.Ether(src=src_mac)
18
19    livello_ip = scapy.IP(src=src_ip, dst=dst_ip)
20
21    livello_icmp = scapy.ICMP()
22
23    packet = livello_dataLink / livello_ip / livello_icmp
24
25    risposta = scapy.srp1(packet, timeout=1, verbose=0)
26
27    if risposta:
28        print(f"Risposta da {dst_ip}: {risposta.summary()}")
29    else:
30        print(f"Nessuna risposta da {dst_ip}")
31
32
33"""----- Fine sezione metodi di utilità -----"""
34
35
36if __name__ == '__main__':
37    invio_ping(source_ip, destination_ip, source_mac)
```

Figura 2.3: Esempio di HTS Poisoning.

Sulla base del contenuto del pacchetto inviato si determina l'effetto dell'attacco. In uno scenario come quello della figura 2.3, si sta realizzando una Host Impersonation / Denial of Service. Avendo inserito IP e MAC sorgenti di H3 nel pacchetto, H2 è in grado di modificare la location di H3 e farla risultare uguale alla propria. La conseguenza di questa azione è che da quel momento in poi qualunque pacchetto inviato ad H3 arriverà invece ad H2, mentre il legittimo destinatario risulterà *irraggiungibile*.

È da tenere a mente tuttavia che nonostante la location logica di H3 sia cambiata, esso continua ad essere collegato fisicamente alla sua reale porta. Questo implica che se H3 dopo aver subito l'attacco iniziasse a inviare pacchetti, l'HTS rileverebbe uno spostamento, ripristinando la sua posizione a quella originaria; è chiaro però che nulla vieta all'attaccante H2 di ripetere l'attacco periodicamente per continuare ad avvelenare la location di H3.

Qualora H2, invece di inserire IP e MAC sorgenti di un host esistente (nel caso precedente H3), avesse inserito dei valori randomici, avrebbe dato vita ad un Fake Host Generation: inviando il pacchetto ICMP malformato, l'HTS avrebbe rilevato erroneamente la presenza di un nuovo host nella rete.

2.2 ARP Poisoning

La seconda tipologia di attacco riguarda il protocollo *ARP*. In generale, anche all'interno delle reti classiche, se un host A volesse contattare un host B, per ottenere il suo indirizzo IP deve prima di tutto individuare l'indirizzo MAC ad esso associato e ciò avviene grazie al protocollo ARP, secondo uno schema request-response. L'host A invia prima di tutto una richiesta broadcast, ovvero una ARP request a tutti i dispositivi della rete, chiedendo l'indirizzo MAC dell'IP dell'host B. Tutti gli host della rete riceveranno il pacchetto inviato dall'host A, che si aspetterà una risposta tramite ARP reply soltanto dall'host B.

L'attacco può avvenire anche nel caso in cui l'host A non abbia inviato alcun pacchetto di tipo ARP request, in quanto il protocollo ARP non controlla che un pacchetto di tipo ARP reply sia collegato ad una particolare richiesta effettuata in precedenza.

Nel contesto di un attacco di tipo ARP poisoning, un attaccante, ad esempio H2, cerca di inviare dei pacchetti di ARP reply contenente informazioni false, e di manipolare così la tabella ARP di un altro host. A seguito dell'avvelenamento, tutti i pacchetti che in realtà erano destinati all'host B, verranno inviati all'host controllato dall'attaccante.

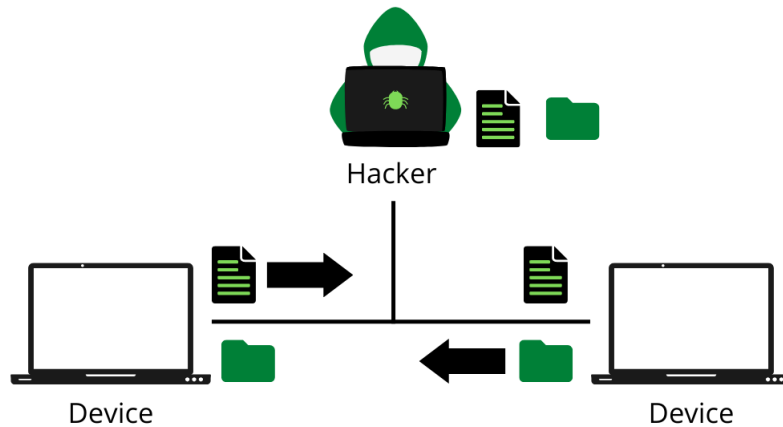


Figura 2.4: Schema di un attacco ARP Poisoning.

L'ARP Poisoning permette di realizzare sia l'Host Impersonation che il Denial of Service visti in 2.1, ma anche un *Man In The Middle*. Considerando la topologia vista in figura 2.2, si supponga sempre H2 attaccante: attraverso l'invio periodico di pacchetti ARP REPLY opportunamente realizzati, H2 è in grado di modificare le tabelle ARP delle vittime scelte, ad esempio H1 e H3. Grazie a questa modifica, nel momento in cui H1 decidesse di inviare pacchetti ad H3, o viceversa, H2 sarebbe in grado di intercettare il traffico di rete e inoltrarlo al legittimo proprietario, senza che nessuno dei due interlocutori si accorga di quanto stia succedendo.

Nella figura 2.5, è possibile osservare il risultato dell'attacco: è riportato il contenuto della tabella ARP dell'host H1 (vittima) in due momenti differenti: inizialmente contiene le corrispondenze IP-MAC corrette a stampata a seguito di un ping tra tutti gli host, mentre la seconda a seguito di un attacco. È possibile osservare come nella prima esecuzione del comando `arp -a` l'IP dell'host H3 sia associato al MAC corretto (00:00:00:00:00:03) mentre nella seconda esecuzione a causa dell'attacco, l'IP di H3 risulta uguale a quello dell'attaccante H2 (00:00:00:00:00:02).

```

root@domenico-VirtualBox:/home/domenico/Scrivania/progettoNetwork# arp -a
? (10.0.0.2) associato a 00:00:00:00:00:02 [ether] su h1-eth0
? (10.0.0.3) associato a 00:00:00:00:00:03 [ether] su h1-eth0
? (10.0.0.254) associato a 00:00:00:00:00:05 [ether] su h1-eth0
? (10.0.0.4) associato a 00:00:00:00:00:04 [ether] su h1-eth0
root@domenico-VirtualBox:/home/domenico/Scrivania/progettoNetwork# arp -a
? (10.0.0.2) associato a 00:00:00:00:00:02 [ether] su h1-eth0
? (10.0.0.3) associato a 00:00:00:00:00:02 [ether] su h1-eth0
? (10.0.0.254) associato a 00:00:00:00:00:05 [ether] su h1-eth0
? (10.0.0.4) associato a 00:00:00:00:00:04 [ether] su h1-eth0

```

Figura 2.5: Tabelle ARP dell'host H1 a seguito di un attacco da parte di H2.

2.3 Man In The Middle su API Rest di ONOS

Durante la sperimentazione dei due attacchi appena discussi sull'Host Tracking Service, si è osservato che nel caso del controller ONOS il servizio può essere compromesso in un ulteriore modo, ossia attraverso le API REST messe a disposizione del controller. Questo è possibile perché le modifiche effettuate secondo questo meccanismo hanno priorità maggiore rispetto a quelle effettuate dall'Host Location Provider, l'implementazione predefinita di ONOS per il servizio di HTS. In particolare, se un host effettua delle richieste utilizzando le API REST, può inviare delle POST in cui il corpo della richiesta contiene delle modifiche alla configurazione di rete illegittime. Effettuando le richieste all'url:

`http://172.17.0.2:8181/onos/v1/network/configuration/hosts`

la richiesta di modifica sarà presa in carico da parte di un componente interno del core di ONOS denominato *Network Config Listener*, avente priorità maggiore rispetto all'Host Location Provider. Questo ha come diretta conseguenza che un attaccante attraverso questo meccanismo può modificare **permanente**mente la location di un host vittima, e l'Host Location Provider non sarà più in grado di ripristinare la corretta posizione nel momento in cui la vittima invierà un pacchetto nella rete, come invece avveniva precedentemente(2.1).

API REST di ONOS. Per effettuare un attacco del genere tuttavia, è bene precisare che l'attaccante oltre a conoscere i dettagli della topologia da attaccare (conoscibili attraverso una fase pre-attacco di reconnaissance) deve anche conoscere le credenziali necessarie per effettuare richieste HTTP, che di default sono "onos" per l'username e "rocks" per la password. ONOS infatti utilizza un meccanismo di tipo HTTPBasicAuth per proteggere le API REST, ma se un amministratore di rete disattento non cambia le credenziali predefinite, chiunque può effettuare richieste e quindi attacchi.

In generale si può ipotizzare anche uno scenario in cui un amministratore di rete abbia realizzato una custom app per ONOS che permetta di effettuare richieste HTTP, ma non contenga controlli adeguati per bloccare le richieste illegittime o potenzialmente dannose. Si intuisce subito da quanto detto finora che un attaccante che può effettuare richieste HTTP detiene un potere immenso: oltre all'esempio appena discusso infatti, tramite API REST è possibile effettuare qualunque tipo di operazione sul controller, dall'aggiunta/rimozione degli host, alla modifica delle tabelle di flusso, o più in generale si ha il completo controllo della rete e dei suoi componenti.

Attacco. Per dimostrare il potenziale dannoso dell'assenza di validazione sul corpo delle richieste HTTP effettuabili al controller, e per rimanere coerenti con gli attacchi visti finora, si è scelto di realizzare un attacco che sfrutta le API REST di ONOS per attuare un Man in The Middle attraverso una compromissione dell'Host Data Store che sia insanabile da parte dell'Host Location Provider a causa dei motivi espressi poc'anzi.

Si supponga che un host malintenzionato, ad esempio H2, voglia spiare le conversazioni tra gli host vittime H1 e H3. Riprendendo quanto detto all'inizio di questa sezione, l'attaccante H2 inizia l'attacco inviando una richiesta HTTP di tipo POST al controller all'url citato poc'anzi³, e modifica la location di H3, ponendola uguale alla propria. Da quel momento il traffico diretto da H1 ad H3 arriverà ad H2 (passi 1-2-3). Quando H2 intercetterà un messaggio proveniente da H1 (ad esempio una ICMP Echo Request), attraverso un'altra richiesta HTTP ripristinerà la location di H3, per potergli inoltrare il messaggio. Prima

³<http://172.17.0.2:8181/onos/v1/network/configuration/hosts>

di inoltrarlo, H2 cambierà la location di H1 allo stesso modo di come fatto in precedenza con H3: questo servirà perché quando H3 risponderà, H2 dovrà intercettare anche la risposta diretta ad H1.

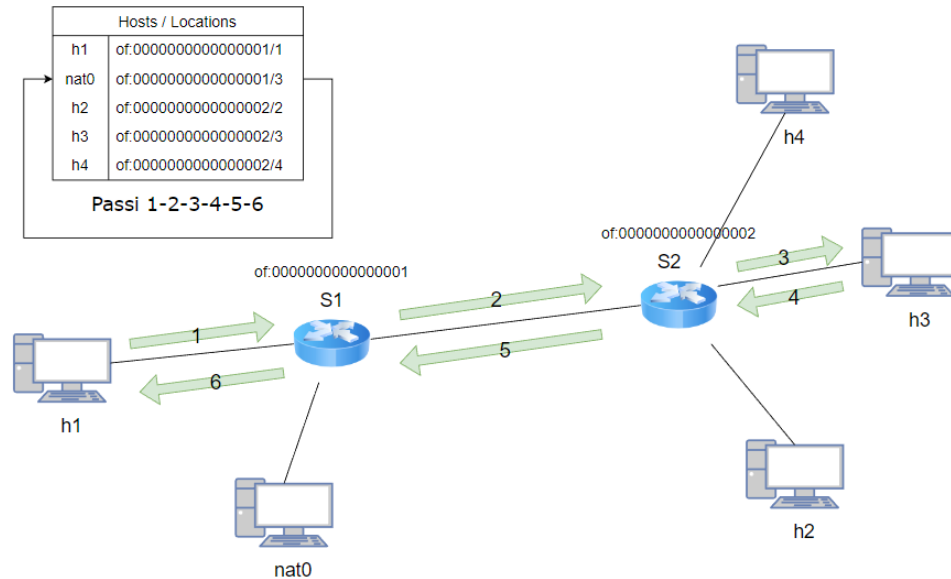


Figura 2.6: Topologia prima dell'attacco.

A quel punto H2 inoltrerà il messaggio ad H3, che risponderà pensando che il messaggio arriverà ad H1, ma sarà invece nuovamente intercettato da H2 (passi 4-5-6-7). H2 provvederà infine a inoltrare la risposta di H3 ad H1 (passi 8-9-10). L'attaccante ha così intercettato conversazione completa (richiesta-risposta) e ripetendo ad oltranza i passaggi appena visti può continuare a spiare il traffico in transito. Quando l'attaccante decide di terminare l'attacco, attraverso ulteriori richieste HTTP è in grado di ripristinare la topologia pre-attacco ed eliminare le configurazioni inserite dall'Host Data Store.

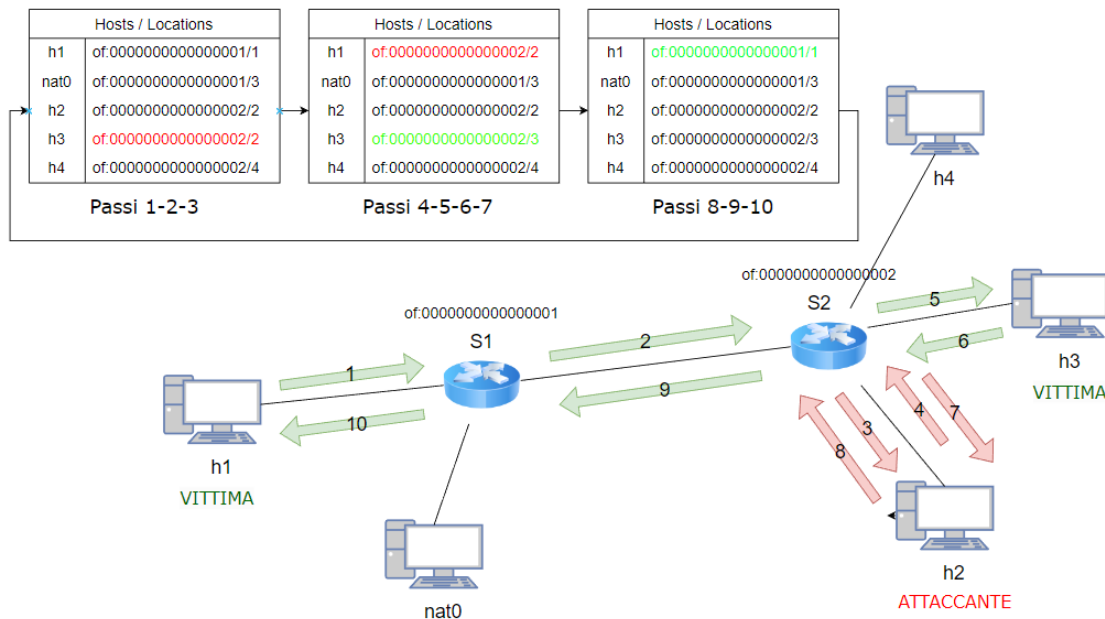


Figura 2.7: Topologia durante l'attacco.

In conclusione, si mostra una cattura effettuata con wireshark sui tre host coinvolti in questo scenario (H1, H2, H3) durante l'esecuzione dell'attacco:

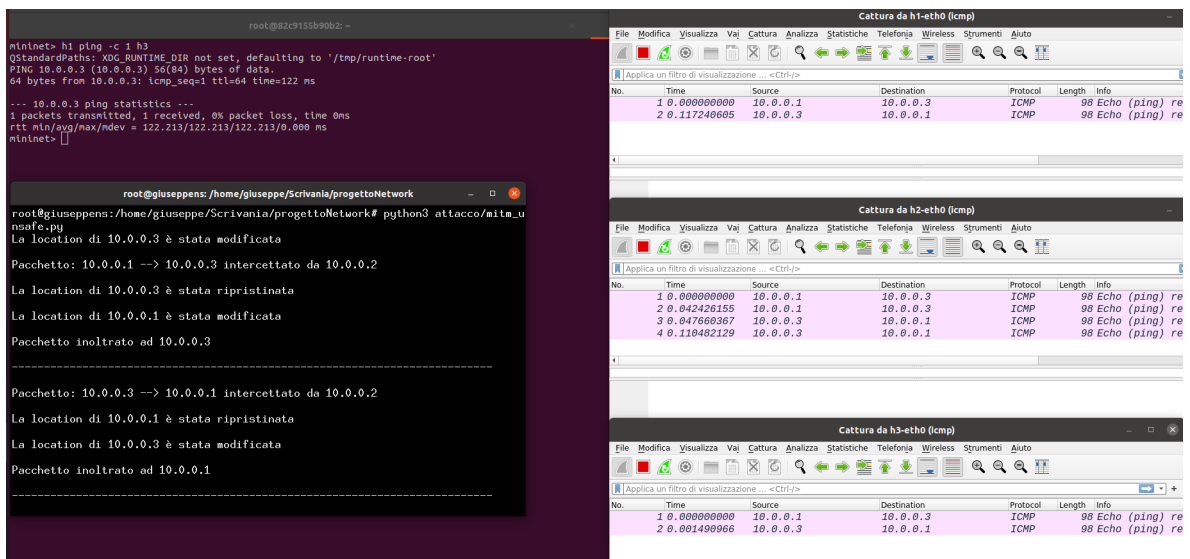


Figura 2.8: Cattura con Wireshark effettuata durante l'attacco.

Capitolo 3

Contromisure e Valutazione Costi-Benefici

I principali pericoli a livello hosts in una rete SDN si verificano a causa della mancanza di meccanismi di autenticazione nei componenti interni dei controllers. Gli attacchi di ARP Poisoning e di HTS poisoning descritti nel capitolo precedente sfruttano proprio la mancanza di autenticazione per trasmettere all'interno della rete messaggi di Packet-In che contengono dati non veritieri i quali vengono intercettati dall'Host Tracking Service, ente che in ONOS viene implementato attraverso l'applicazione di *Host Location Provider*, la quale setta sulla base di questi pacchetti informazioni importanti per il corretto funzionamento della rete, come ad esempio le locations degli switch alle quali sono connessi gli hosts. Nasce dunque la necessità di implementare dei meccanismi che permettono sia al controller che ai dispositivi della rete di identificare in primis in maniera corretta chi sta inviando un determinato pacchetto e una volta identificato il mittente per evitare anche attacchi da hosts interni alla rete implementare dei meccanismi di controllo sul contenuto del payload.

Per mettere in atto quanto detto finora si è sfruttata la proprietà che il controller ONOS è open source e quindi mette a disposizione il codice sorgente e si è sfruttato inoltre la possibilità di realizzare applicazioni custom, per cui è stato preso il codice sorgente dell'Host Location Provider ed è stato esteso per realizzare due applicazioni di difesa che rilevano e bloccano gli attacchi, espellendo l'attaccante dalla rete. Le due difese differiscono l'una dall'altra dai meccanismi crittografici utilizzati. Seguirà nei paragrafi successivi una mag-

giore descrizione delle due applicazioni.

Nonostante la realizzazione di questi due meccanismi di difesa, si è scoperto che l'Host Tracking Service può essere ugualmente aggirato se viene effettuato l'attacco descritto nel paragrafo 2.3. In particolare, questo attacco sfrutta la maggiore priorità del componente *Network Config Listener* di ONOS rendendo permanente anche in presenza dell'Host Location Provider lo scambio illegale di locations.

Per risolvere questo problema è stato implementato un meccanismo di proxy sicuro che utilizza la firma digitale per garantire l'autenticità e l'integrità del pacchetto.

Per semplicità nei test crittografici sono state utilizzate delle chiavi a 256 bit, tuttavia per una maggiore sicurezza in casi di applicazioni reali è consigliabile utilizzare chiavi crittografiche più lunghe.

Ovviamente l'estensione dell'Host Tracking Service, l'implementazione del meccanismo di proxy e l'introduzione della crittografia hanno comportato dei costi riguardanti l'utilizzo della larghezza di banda, dei tempi di comunicazione della rete e dell'utilizzo delle risorse hardware per cui è stata effettuata una analisi delle prestazioni mostrando attraverso dei grafici le differenze.

3.1 Detection App 1

Questo meccanismo di difesa è stato implementato estendendo e realizzando una versione custom della classe ONOS dell'Host Location Provider per evitare attacchi di Cross App Poisoning nei suoi confronti. Per fare ciò questa applicazione rileva e blocca immediatamente i tentativi di attacco bannando dalla rete l'host che ha cercato di effettuarli. Per tenere sotto controllo lo stato della rete è stato realizzato uno snapshot che viene aggiornato automaticamente ogni qual volta vengono rilevati cambiamenti all'interno della topologia. È stato inoltre considerato il seguente scenario:

«Se un host perde la connessione o si disconnette volontariamente dalla rete, è possibile che un altro host si accorga di questa caduta prima del controller. Questo host potrebbe sfruttare la caduta per inviare pacchetti e convincere sia

gli altri hosts che il controller che l'host è ancora attivo, impersonificandolo. Questo scenario è stato gestito introducendo la crittografia ed un meccanismo di probing dove ogni 20 secondi il controller invia un pacchetto cifrato a cui solo l'host corretto potrà rispondere».

Per mitigare gli attacchi di tipo ARP Poisoning e di HTS Poisoning viene utilizzato il servizio di *Packet Service* di ONOS che tramite la sua funzionalità di *Packet Processor* permette di intercettare i pacchetti che arrivano al controller e trattarli in maniera differente a seconda del tipo di attacco rilevato. Ovviamente non vengono trattati i pacchetti riferiti al nat perché abbiamo assunto l'ipotesi che il nat è un host fidato.

Se arriva un pacchetto ARP si va a controllare la presenza di un ARP Poisoning: viene intercettato il pacchetto e da esso vengono estratti l'indirizzo IP del mittente e la sua location e si va a controllare se la location è valida e dunque presente nello snapshot. Se il controllo viene superato allora si va a controllare che la lista degli indirizzi IP di quella location non sia vuota e/o se l'indirizzo IP del pacchetto non sia associato a quella location; se questo controllo viene superato allora si sta rilevando un tentativo di attacco per cui il pacchetto viene bloccato e, viene bannato il mittente della richiesta utilizzando il suo indirizzo MAC. La rimozione dell'attaccante consiste nella sua rimozione dall'Host Store, l'aggiornamento dello snapshot, e l'installazione di una regola di flusso in modo tale che finché la rete non viene riavviata questi non può più rientrarvi.

Se arriva un pacchetto di tipo IPV4 potrebbero verificarsi due scenari: se è un pacchetto di tipo ECHO REPLY si tratta di un pacchetto di probing e discuteremo in seguito come verrà gestito, se invece è un pacchetto ICMP classico si va a controllare la presenza di un HTS Poisoning: viene intercettato il pacchetto dal quale si estrae l'indirizzo MAC del mittente e location da cui arriva. Anche qui si va a verificare che la location sia valida e quindi contenuta nello snapshot e se il controllo viene superato si va a controllare che l'indirizzo MAC del mittente sia effettivamente quello associato alla location. Se questo controllo non viene superato, allora vuol dire che il pacchetto è malformato e contiene un tentativo di attacco per cui sempre attraverso il MAC viene bannato dalla rete l'attaccante installando una regola di flusso come nel caso

precedente e rimuovendolo dallo snapshot e dall'Host Store.

Per risolvere il problema della disconnessione di un host descritta in precedenza è stato implementato un meccanismo di probing in cui il controller invia dei pacchetti ICMP di tipo ECHO REPLY agli hosts come heartbeat per sapere se sono ancora attivi. Questo meccanismo parte con un delay iniziale all'avvio dell'applicazione di circa 10 secondi per poi ripetersi con una frequenza di 20 secondi. Il probing è il meccanismo con cui viene utilizzato l'Authenticated Encryption with Associated Data. Utilizzando in questo caso un algoritmo di cifratura simmetrico è stato necessario implementare un meccanismo di scambio delle chiavi, inoltre per evitare **Reflection Attack** per ogni hosts vengono utilizzate due chiavi segrete: una per gestire la comunicazione $HOST \rightarrow CONTROLLER(K_{h \rightarrow c})$, l'altra per gestire la comunicazione $CONTROLLER \rightarrow HOST(K_{c \rightarrow h})$. Per lo scambio delle chiavi è stato utilizzato l'**Algoritmo di Diffie-Hellman**. Lo scenario si svolge in questo modo: il controller avvia la procedura di scambio delle chiavi verso gli hosts, in caso di mancata risposta la procedura viene ritentata. Se per tre volte consecutive il controller non ottiene risposta assume che l'host sia inattivo nella rete e lo espelle; se invece ottiene risposta si conclude lo scambio delle chiavi per poi iniziare l'invio del probing cifrato. Il pacchetto di probing è composto dalla scritta *PROBING* per riconoscere che è un messaggio di probing, poi vengono inseriti 12 bytes di *nonce* generati casualmente per evitare **Replay Attack** e poi dal *payload ICMP* e viene cifrato utilizzando l'algoritmo di cifratura AES/GCM e la chiave $K_{h \rightarrow c}$ per poi essere inviato agli hosts.

Lato hosts quando arriva il pacchetto viene decifrato utilizzando la stessa chiave usata per la cifratura $K_{h \rightarrow c}$, se la decifratura va a buon fine l'host è in grado di autenticare il controller in quanto solo lui (oltre a questo host) conosce questa chiave. A questo punto l'host calcola casualmente un nuovo nonce e costruisce il pacchetto di risposta al probing cifrandolo con la chiave $K_{c \rightarrow h}$ e lo invia al controller. Il controller avvia la procedura di decifratura del pacchetto: se il processo restituisce esito positivo allora il controller è in grado di autenticare l'host in quanto solo lui (oltre che al controller) possiede la chiave $K_{c \rightarrow h}$. Se il processo restituisce esito negativo il controller solleva un messaggio di warning in cui comunica la mancata corrispondenza nel payload di

probing. Se l'host non risponde al tentativo di probing per tre volte di fila o se si ottiene una mancata corrispondenza di payload per tre volte di fila l'host viene considerato inattivo o malevolo e viene rimosso.

La Figura 3.1 mostra il corretto funzionamento dell'applicazione quando nella rete non è presente alcun attacco.

```
16:40:33.999 INFO [MltnDetection] Configured. Using ARP is enabled
16:40:33.999 INFO [MltnDetection] Configured. Using IPv6 NDP Neighbor Solicitation and Advertisement is disabled
16:40:34.000 INFO [MltnDetection] Configured. Using IPv6 NDP Router Solicitation and Advertisement is disabled
16:40:34.000 INFO [MltnDetection] Configured. Using DHCP is disabled
16:40:34.000 INFO [MltnDetection] Configured. Using DHCP6 is disabled
16:40:34.001 INFO [MltnDetection] Configured. Request intercepts is enabled
16:40:34.001 INFO [MltnDetection] Configured. Multihoming is disabled
16:40:43.772 INFO [MltnDetection] Avviata procedura di scambio delle chiavi con l'host: 00:00:00:00:03/None
16:40:43.776 INFO [MltnDetection] Avviata procedura di scambio delle chiavi con l'host: 00:00:00:00:01/None
16:40:43.779 INFO [MltnDetection] Avviata procedura di scambio delle chiavi con l'host: 00:00:00:00:02/None
16:40:43.782 INFO [MltnDetection] Avviata procedura di scambio delle chiavi con l'host: 00:00:00:00:04/None
16:41:03.764 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:03/None
16:41:03.765 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:01/None
16:41:03.766 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:02/None
16:41:03.767 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:04/None
16:41:03.786 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:01/None
16:41:03.786 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:02/None
16:41:03.786 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:03/None
16:41:03.787 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:04/None
16:41:23.765 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:03/None
16:41:23.767 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:01/None
16:41:23.769 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:02/None
16:41:23.771 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:04/None
16:41:23.797 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:03/None
16:41:23.799 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:01/None
16:41:23.803 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:04/None
16:41:23.804 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:02/None
16:41:43.764 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:03/None
16:41:43.766 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:01/None
16:41:43.767 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:02/None
16:41:43.768 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:04/None
16:41:43.791 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:03/None
16:41:43.792 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:02/None
16:41:43.802 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:01/None
16:41:43.806 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:04/None
16:42:03.764 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:03/None
16:42:03.766 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:01/None
16:42:03.768 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:02/None
16:42:03.770 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:04/None
16:42:03.794 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:02/None
16:42:03.794 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:01/None
16:42:03.796 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:03/None
16:42:03.800 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:04/None
```

Figura 3.1: Scambio delle chiavi e probing.

La Figura 3.2 mostra come l'applicazione rileva e blocca un attacco di ARP Poisoning e rimuove l'attaccante.

```
16:47:41.537 INFO [MltnDetection] Avviata procedura di scambio delle chiavi con l'host: 00:00:00:00:03/None
16:47:41.543 INFO [MltnDetection] Avviata procedura di scambio delle chiavi con l'host: 00:00:00:00:01/None
16:47:41.547 INFO [MltnDetection] Avviata procedura di scambio delle chiavi con l'host: 00:00:00:00:02/None
16:47:41.551 INFO [MltnDetection] Avviata procedura di scambio delle chiavi con l'host: 00:00:00:00:04/None
16:48:01.526 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:03/None
16:48:01.527 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:01/None
16:48:01.527 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:02/None
16:48:01.529 INFO [MltnDetection] Inviato Pacchetto di Probing all'host: 00:00:00:00:04/None
16:48:01.555 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:02/None
16:48:01.555 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:01/None
16:48:01.563 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:04/None
16:48:01.564 INFO [MltnDetection] Ricevuto Pacchetto di Probing da: 00:00:00:00:03/None
16:48:03.119 WARN [MltnDetection] Rilevato Pacchetto ARP Malformato. Pacchetto Scartato.
16:48:03.125 WARN [MltnDetection] Espulso Host Malevolo avente Indirizzo MAC: 00:00:00:00:00:02 sul Device: of:0000000000000002
```

Figura 3.2: Mitigazione ARP Poisoning.

La Figura 3.3 mostra come l'applicazione rileva e blocca un attacco di HTS Poisoning e rimuove l'attaccante.

```

14:16:52 INFO [DistributedAppare] Group AUDIT: Setting device of:0000000000000001 initial AUDIT completed
14:17:93 INFO [TopologyManager] Topology DefaultTopology(time=43894184058205, creationTime=1721322857936, computeCost=136472, clusters=1, device
14:21:81 WARN [MitnDetection] Rilevato Pacchetto IPv4 Malformato. Pacchetto Scartato.
14:26:55 INFO [MitnDetection] Espulso Host Malevolo avente indirizzo MAC: 00:00:00:00:00:02 sul Device: of:0000000000000002
14:26:55 INFO [MitnDetection] Avviata procedura di scambio delle chiavi con l'host: 00:00:00:00:00:03/None
14:26:56 INFO [MitnDetection] Avviata procedura di scambio delle chiavi con l'host: 00:00:00:00:00:04/None

```

Figura 3.3: Mitigazione HTS Poisoning.

3.2 Detection App 2

Anche questa applicazione è un'implementazione sicura dell'Host Location Provider come la precedente e allo stesso modo gestisce il rilevamento di un ARP Poisoning e di un HTP Poisoning.

La differenza sostanziale tra le due applicazioni è il meccanismo utilizzato per fare probing e quindi rilevare gli hosts attivi all'interno della rete evitando che qualcuno sfrutti la loro caduta per impersonarli. In questo caso si utilizza *Hybrid Public Key Encryption*, si usa quindi una combinazione tra la crittografia asimmetrica e la crittografia simmetrica. In questo caso come nel caso del Safe Proxy descritto nella sezione successiva le chiavi pubbliche e private sono state generate una sola volta utilizzando la crittografia a curve ellittiche per poi essere utilizzate negli esperimenti come costanti.

Ovviamente in casi reali è fortemente consigliato la generazione periodica di nuove chiavi

Le chiavi private sono state utilizzate per definire una chiave di sessione: il controller prende la chiave pubblica di un host (si suppone che dietro le quinte ci sia una Certification Authority che certifica le chiavi pubbliche degli hosts e del controller) e in coppia con la sua (del controller) chiave privata calcola la chiave di sessione tramite Diffie-Hellman a curve ellittiche. Allo stesso modo l'host prende la chiave pubblica del controller e la sua (dell'host) chiave privata, applica Diffie-Hellman a curve ellittiche ed ottiene la stessa chiave di sessione segreta generata dal controller. A questo punto il controller genera un *IV* casuale per evitare **Reflection Attack** e lo aggiunge all'inizio del testo in chiaro.

Questa concatenazione viene cifrata con la chiave di sessione e viene inviata all'host come pacchetto di probing. L'host avvia la procedura di decifratura e se il risultato è positivo riesce ad autenticare il controller. Successivamente prepara la risposta generando un nuovo **IV**, lo concatena al payload, e lo cifra usando la stessa chiave di sessione tramite l'algoritmo AES/GCM. Invia poi il messaggio cifrato al controller il quale avvierà la procedura di decifratura. Se riuscirà a decifrare allora autenticcherà l'host e lo considererà il suo stato come attivo, altrimenti scriverà nei logs un messaggio di warning in cui specifica la mancata autenticazione. Anche in questo caso se un host non risponde a tre probing di fila o non riesce ad autenticarsi per tre volte viene considerato inattivo o malevolo e verrà espulso dalla rete. La Figura 3.4 mostra il corretto funzionamento dell'applicazione quando nella rete non è presente alcun attacco.

```

43:08.633 INFO [TopologyManager] Topology DefaultTopology[time=42024278277404, creationTime=1721
43:17.106 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:03/None
43:17.107 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:01/None
43:17.108 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:02/None
43:17.109 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:04/None
43:17.134 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:04/None
43:17.134 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:01/None
43:17.135 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:03/None
43:17.135 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:02/None
43:36.891 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:03/None
43:36.893 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:01/None
43:36.894 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:02/None
43:36.896 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:04/None
43:36.899 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:02/None
43:36.918 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:04/None
43:36.918 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:01/None
43:36.923 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:03/None
43:56.891 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:03/None
43:56.893 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:01/None
43:56.894 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:02/None
43:56.896 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:04/None
43:56.905 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:02/None
43:56.905 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:01/None
43:56.927 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:03/None
43:56.935 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:04/None
44:16.891 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:03/None
44:16.893 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:01/None
44:16.894 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:02/None
44:16.925 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:02/None
44:16.928 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:03/None
44:16.938 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:01/None
44:16.939 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:04/None

```

Figura 3.4: Probing hpke.

La Figura 3.5 mostra come l'applicazione rileva e blocca un attacco di ARP Poisoning e rimuove l'attaccante.

```

7:02:44.469 INFO [MitnDetection2] Configured. Multihoming is disabled
7:02:45.434 INFO [TopologyManager] Topology DefaultTopology[time=43201678038774, creationTime=1721322165430, computeCost=216191, clusters=1,
7:02:54.460 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:03/None
7:02:54.463 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:01/None
7:02:54.463 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:02/None
7:02:54.463 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:04/None
7:02:54.486 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:02/None
7:02:54.486 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:03/None
7:02:54.486 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:01/None
7:02:54.499 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:04/None
7:03:01.276 WARN [MitnDetection2] Rilevato Pacchetto ARP Malformato. Pacchetto Scartato.
7:03:01.282 WARN [MitnDetection2] Espulso Host Malevolo avente Indirizzo MAC: 00:00:00:00:00:02 sul Device: of:0000000000000002

```

Figura 3.5: Mitigazione ARP Poisoning.

La Figura 3.6 mostra come l'applicazione rileva e blocca un attacco di HTS Poisoning e rimuove l'attaccante.

```
7:12:25.485 INFO [MitnDetection2] Configured. Using DHCP6 is disabled
7:12:25.486 INFO [MitnDetection2] Configured. Request intercepts is enabled
7:12:25.486 INFO [MitnDetection2] Configured. Multihoming is disabled
7:12:30.702 WARN [MitnDetection2] Rilevato Pacchetto IPv4 Malformato. Pacchetto Scartato.
7:12:30.706 WARN [MitnDetection2] Espulso Host Malevolo avente indirizzo MAC: 00:00:00:00:00:02 sul Device: of:0000000000000002
7:12:35.246 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:00:03/None
7:12:35.247 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:00:01/None
7:12:35.248 INFO [MitnDetection2] Inviato Pacchetto di Probing all'host: 00:00:00:00:00:04/None
7:12:35.274 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:00:03/None
7:12:35.274 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:00:01/None
7:12:35.282 INFO [MitnDetection2] Ricevuto Pacchetto di Probing da: 00:00:00:00:00:04/None
```

Figura 3.6: Mitigazione HTS Poisoning.

3.3 Safe Proxy sull'host nat

Come già detto in precedenza l'implementazione di un proxy di sicurezza è stato necessario per risolvere il Man In The Middle su API REST. Affinché gli hosts possano effettuare questo attacco è stato necessario introdurre all'interno della rete un host nat il quale reindirizza tutti i messaggi che dall'interno della rete devono raggiungere l'esterno. Per realizzare questo meccanismo di sicurezza si assume l'ipotesi che l'host nat sia un host fidato all'interno della topologia. La logica del meccanismo si basa sul modello client-server in cui l'host nat è un server che rimane in ascolto sulla porta 8080 come mostrato in Figura 3.7 in attesa che i client, ovvero gli altri hosts all'interno della topologia, abbiano la necessità di comunicare con l'esterno.

```
197 def avvio(server_class=http.server.HTTPServer, handler_class=GestoreRichiesteHttp):
198     ip_server = (IP_HOST, PORTA_HOST)
199     httpd = server_class(ip_server, handler_class)
200     print(f'Avviato MITM Detection su {IP_HOST}:{PORTA_HOST}')
201     try:
202         httpd.serve_forever()
203     except KeyboardInterrupt:
204         print("\nTerminazione di MITM Detection.")
205         httpd.server_close()
```

Figura 3.7: Codice di avvio del safe proxy.

Quando un host ha questa necessità, costruisce e firma una richiesta con la sua chiave privata e la inoltra al nat. Potrebbe succedere che questo host sia malevolo e che quindi stia cercando di impersonare un altro host oppure che

stia cercando di modificare la location di un altro host per poter ascoltare le sue conversazioni, per cui il nat per evitare ciò effettua i seguenti passi:

1. Estrae l'indirizzo IP dalla richiesta.
2. Prova a decifrare il pacchetto, e quindi validare la firma, sulla base della chiave pubblica associata all'host avente quell'indirizzo IP, come mostrato in Figura 3.8.
 - (a) Se il processo di decifratura va a buon fine allora il mittente è realmente chi dice di essere e si prosegue con il passo 3.
 - (b) Se il processo di decifratura non va a buon fine il nat capisce che il mittente non è chi dice realmente di essere e scarta il pacchetto.
3. Se la validazione della firma è verificata si valuta la presenza di tentativi di attacco controllando il contenuto del payload della richiesta, come mostrato in Figura 3.9.
 - (a) Se il payload è pulito si inoltra la richiesta al controller e, quando riceve la risposta, il nat la inoltra all'host.
 - (b) Se il payload risulta essere pericoloso il pacchetto viene scartato mentre l'host viene bannato dalla rete.

L'immagine seguente mostra il codice con cui viene validata la firma

```
101  def valida_firma(self, payload):
102      payload = payload.split(b"separatore")
103      firma, messaggio = payload[0], payload[1]
104
105      try:
106
107          public_key = hpke.converti_chiave_pubblica(chiavi[self.client_address[0]])
108          public_key.verify(firma, messaggio, ec.ECDSA(hashes.SHA256()))
109          result = True
110          return messaggio, result
111      except Exception as e:
112          print(e)
113          raise ValueError('Errore nella validazione della firma')
```

Figura 3.8: Codice di autenticazione hosts.

L'immagine seguente mostra il codice eseguito per validare la bontà della richiesta

```

127  def mitm_detection(corpo_richiesta, ip_mittente):
128      topologia = acquisisci_snapshot()
129      ips = None
130      locations = None
131      for dato in corpo_richiesta:
132          ips = corpo_richiesta[dato]["basic"]["ips"]
133          locations = corpo_richiesta[dato]["basic"]["locations"]
134
135      if ip_mittente != ips[0]:
136          return True
137
138      for location in locations:
139          for host in topologia['hosts']:
140              for l in host['locations']:
141                  verifica = f"{l['elementId']}/{l['port']}"
142                  if location == verifica:
143                      if ip_mittente == ips[0] and ips[0] == host['ipAddresses'][0]:
144                          continue
145                      return True
146      return False

```

Figura 3.9: Codice di rilevamento attacchi.

Nota: per semplicità le coppie di chiavi pubbliche e private sono state generate una sola volta utilizzando le curve ellittiche per poi essere state usate come costanti negli esperimenti. Tuttavia, in uno scenario di applicazione reale è fortemente consigliato aggiornare molto spesso le chiavi.

3.4 Valutazione delle Mitigazioni Proposte

Dopo aver presentato diversi meccanismi di difesa per mitigare gli attacchi proposti, sono state effettuate delle analisi per valutare l'efficienza delle soluzioni difensive adottate e i costi introdotti in termini computazionali e temporali.

Tempo di scambio dei pacchetti in diversi scenari. Una prima analisi è stata svolta sui tempi di scambio dei pacchetti tra gli host nella rete. L'obiettivo era valutare in termini temporali quanto tempo in più fosse necessario al controller a gestire i pacchetti ARP e IPv4 a causa dei controlli di sicurezza effettuati su di essi da parte delle due versioni di Mitm Detection App.

Gli scenari considerati sono quattro e in per ciascuno di essi è stata effettuata una media sui tempi relativa allo scambio di 10 pacchetti ICMP tra due host nella rete:

- S1: Nessun attacco in corso e nessun sistema di difesa attivo
- S2: Man In The Middle in corso e nessuna difesa attiva
- S3: Nessun attacco in corso e difesa attiva
- S4: Difesa attiva e Man In The Middle tentato e sventato

Gli esperimenti sono stati ripetuti anche su host appartenenti a switch differenti ma non si è rilevato alcun cambiamento apprezzabile nei tempi di scambio dei pacchetti. Osservando il grafico 3.10 è possibile osservare come nello scenario S3 l'introduzione dei controlli di sicurezza abbia triplicato il tempo necessario a effettuare lo scambio dei pacchetti rispetto allo scenario S1. Al tempo stesso si vede come, in caso di attacco che viene sventato (S4), il tempo medio triplichi ulteriormente, mentre nel caso peggiore si arrivi ad un tempo 35 volte superiore al caso peggiore rispetto allo scenario S1. Per poter confrontare gli scenari S1, S3 ed S4, si è dovuto escludere dal grafico lo scenario S2, il caso estremo in cui è in corso un Man In The Middle realizzato tramite l'attacco di cui si è parlato in 2.3. Nello scenario S2 infatti i tempi di scambio dei pacchetti sono così elevati da rendere illeggibili i valori degli altri tre scenari. La figura 3.11 rappresenta il grafico contenente anche lo scenario S2.

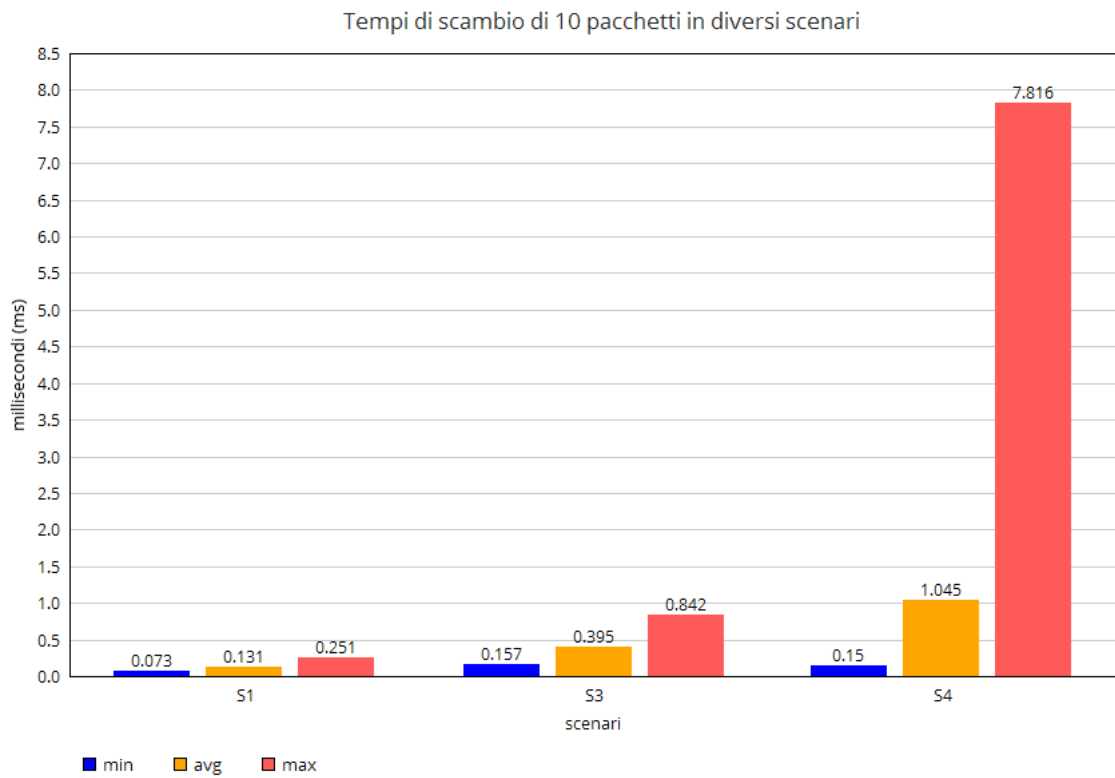


Figura 3.10: Scambio dei pacchetti in tre scenari.

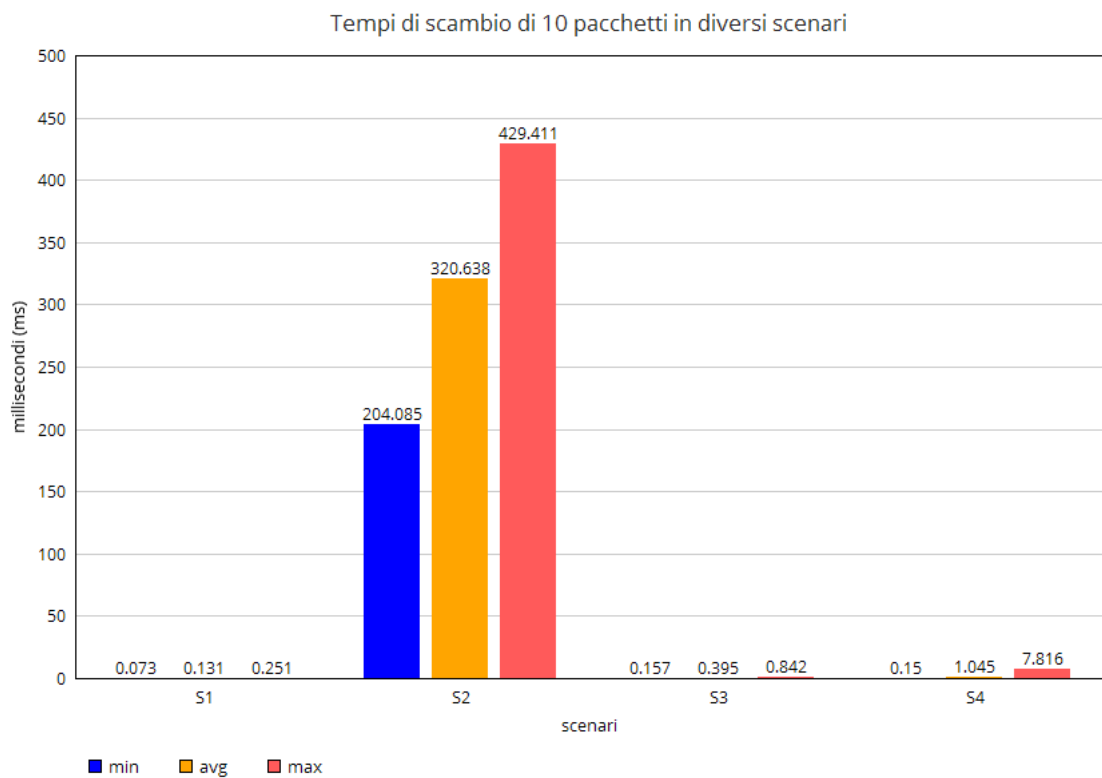


Figura 3.11: Scambio dei pacchetti in quattro scenari.

Tempo di esecuzione delle richieste HTTP. Quest'analisi è volta a valutare il tempo, espresso in millisecondi, impiegato dagli host ad effettuare richieste HTTP al controller in due diversi scenari. Il primo è uno scenario in cui le richieste HTTP arrivano direttamente al controller e non è attiva la difesa che utilizza l'host nat come proxy di difesa. Il secondo scenario prevede la presenza del nat utilizzato come proxy per validare le richieste. La validazione, come detto anche in precedenza, comprende la verifica della firma digitale e un check di legittimità del contenuto del body della richiesta. Lo scenario che considera il proxy attivo prevede che venga inviata una richiesta HTTP legittima e correttamente firmata. In casi diversi da quello appena detto la richiesta viene scartata, rendendo il tempo trascurabile. Dalla figura 3.12 risulta che il tempo medio di risposta al mittente che ha effettuato la richiesta HTTP sia quasi triplicato rispetto al caso standard in cui non è presente un proxy.

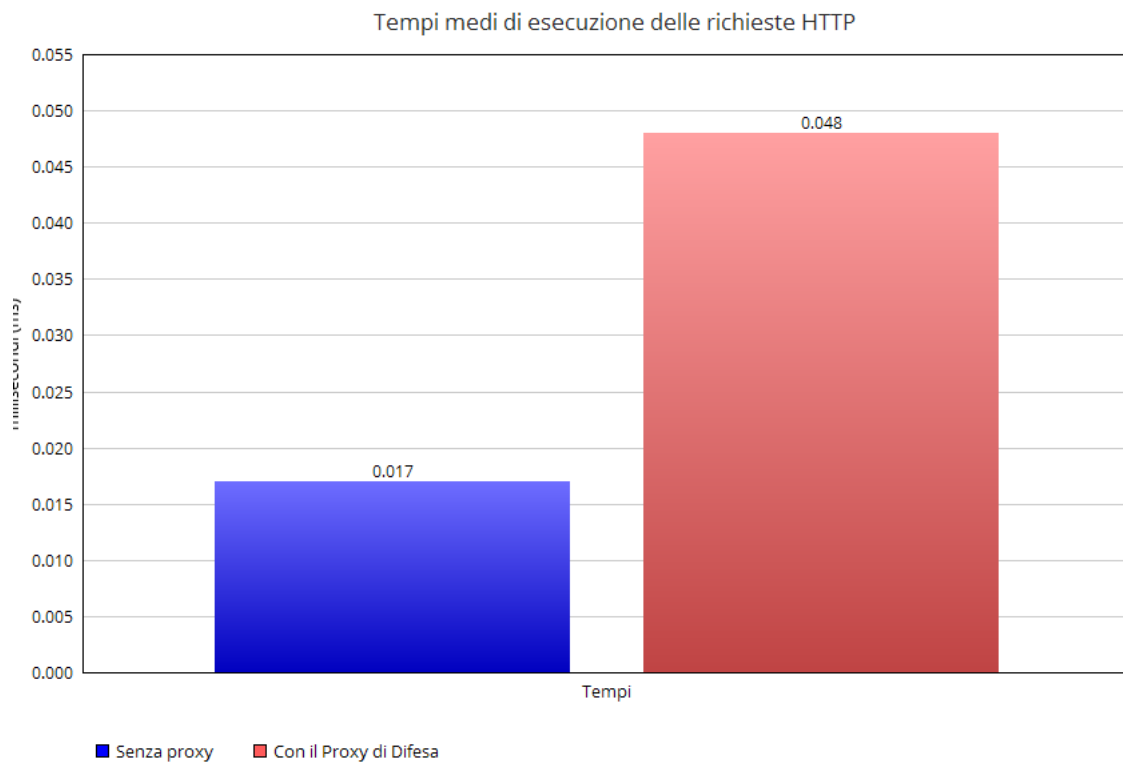


Figura 3.12: Confronto nei tempi medi per effettuare una richiesta HTTP con Proxy e senza Proxy.

Tempo di Discovery in diversi Scenari. Per testare le performance della rete e la sua robustezza al variare delle dimensioni della topologia di riferimen-

to, si è considerato lo scenario in cui è attiva la difesa Mitm Detection App, che utilizza il meccanismo di probing basato su scambio delle chiavi tramite il protocollo Diffie-Hellman e come schema di cifratura AEAD. Il tempo di Discovery in questo scenario è definito come la somma tra il tempo necessario a completare un ping tra tutti gli host e il tempo necessario a completare lo scambio delle chiavi con DH tra il controller ONOS e tutti gli hosts. Si fa presente che un ping iniziale tra tutti gli host è necessario affinché il controller possa rilevarne la presenza e avviare in seguito lo scambio delle chiavi. Il ping iniziale è eseguito in maniera sequenziale, ogni host invia un ping al suo successivo $H1 \rightarrow H2 \rightarrow \dots \rightarrow Hn$. Dalla figura 3.13 si evince che utilizzando topologie che comprendono più di venti hosts e dieci switch il tempo di discovery sia dell'ordine dei secondi, fino ad arrivare a 14.054 s nel caso di cento hosts e cinquanta switch.

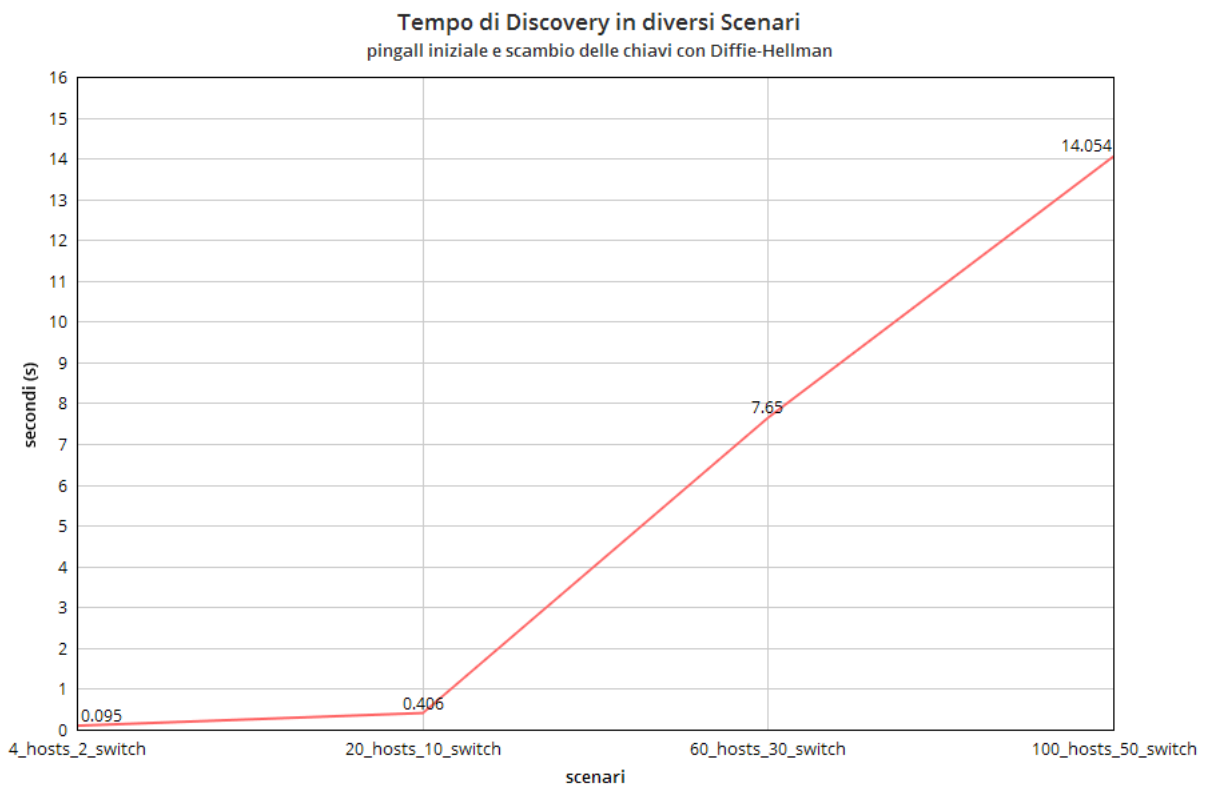


Figura 3.13: Confronto nei tempi medi di Discovery.

Larghezza di Banda utilizzata. L'implementazione dei meccanismi di sicurezza previsti da Mitm Detection App e Mitm Detection App2 ha comportato un incremento nella larghezza di banda utilizzata. Per effettuare la

valutazione sono stati considerati diversi scenari su cui fare le misurazioni. Per avere un parametro di riferimento comune, in ciascuno di questi scenari è stato generato del traffico tramite l'invio di dieci pacchetti ICMP tra due hosts casuali interni alla rete. Ciascuno degli scenari di riferimento utilizza la topologia che prevede 4 hosts e 2 switch (Il nat non è considerato in quanto host fidato). Gli scenari considerati sono:

- S1: Nessun attacco presente, nessuna difesa attiva.
- S2: Man In The Middle 2.3 in corso e nessuna difesa attiva
- S3: Mitm Detection App attiva (Diffie Hellman - AEAD con AES-GCM), nessun attacco in corso
- S4: Mitm Detection App attiva (Diffie Hellman - AEAD con AES-GCM), con tentativo di attacco realizzato tramite HTS Poisoning 2.1 e sventato
- S5: Mitm Detection App2 attiva (HPKE - AEAD con AES-GCM), nessun attacco in corso
- S6: Mitm Detection App2 attiva (HPKE - AEAD con AES-GCM), con tentativo di attacco realizzato tramite HTS Poisoning 2.1 e sventato

Dalla figura 3.14 si evince, come era prevedibile, che il caso di maggiore consumo di larghezza di banda sia lo scenario S2, in cui è in corso un Man In The Middle. È possibile notare invece una leggera differenza nella banda utilizzata tra gli scenari S3 ed S5. La differenza è dovuta all'assenza dei pacchetti utilizzati dallo scenario S3 per effettuare lo scambio delle chiavi con il protocollo Diffie-Hellman. In generale, confrontando lo scenario S1 con gli scenari S3, S4, S5 ed S6, si nota come l'overhead introdotto dalle misure di sicurezza sia dell'ordine di un centinario di Byte in media. La presenza di attacchi sventati invece non comporta variazioni apprezzabili in termini di banda utilizzata.

Più in dettaglio, sono da considerare ulteriori parametri:

- Probing da Controller verso Host: **77 Byte**
- Probing da Host verso Controller: **84 Byte**. La differenza è dovuta all'aggiunta di un identificatore per distinguere i pacchetti di probing da

altri generici pacchetti ICMP Echo Reply. Nei pacchetti di probing da Controller verso Host invece non si creava ambiguità

- Larghezza di Banda utilizzata per effettuare un doppio scambio delle chiavi tra un host e il controller: **765 Byte**

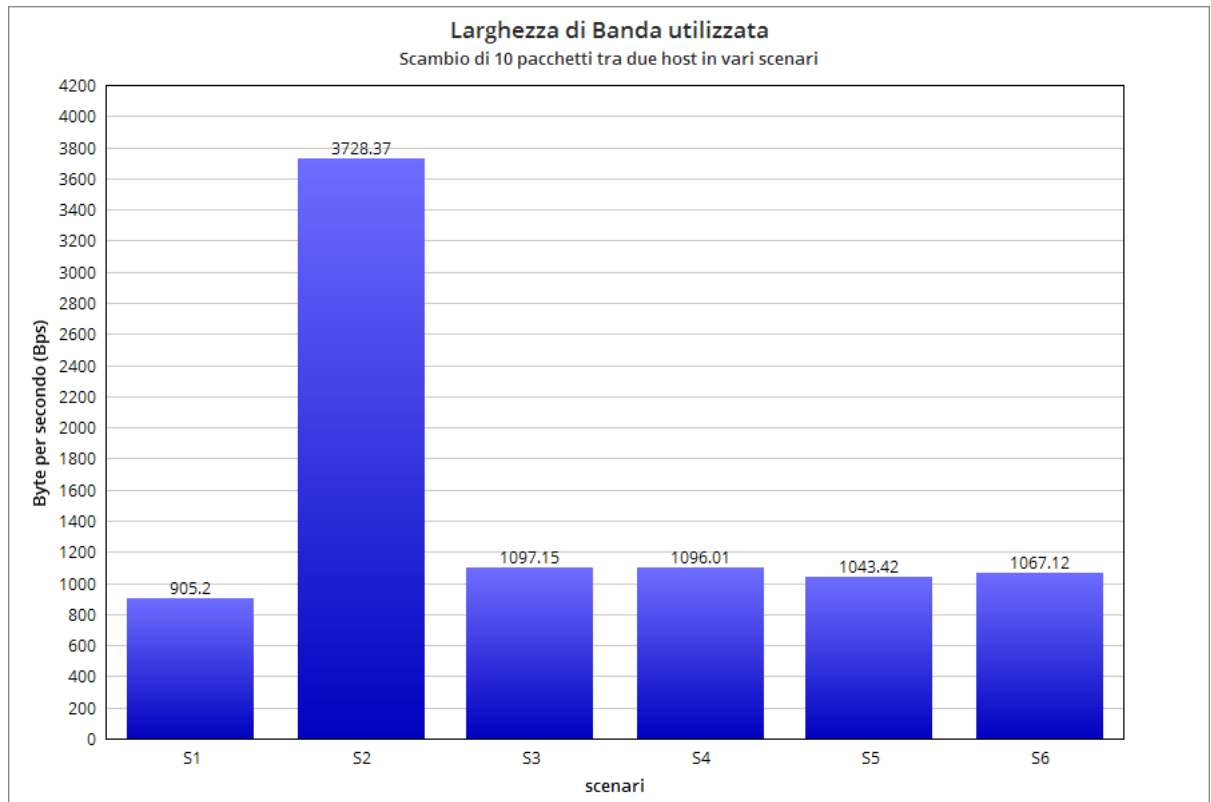


Figura 3.14: Confronto nella larghezza di banda utilizzata in diversi scenari.

Consumo delle risorse hardware (CPU e Ram). Infine, sono presentati dei grafici contenenti delle rilevazioni riguardo il consumo delle risorse hardware in diversi scenari. Le misurazioni sono state effettuate per un tempo totale di 80 secondi in ogni scenario considerando, rilevando il consumo di CPU e Ram ogni 0.5s. Gli scenari considerati sono:

- S1: Nessun attacco e nessuna difesa attiva
- S2: Nessun attacco e difesa Mitm Detection App attiva (Diffie Hellman - AEAD con AES-GCM)
- S3: Nessun attacco e difesa Mitm Detection App2 attiva (HPKE - AEAD con AES-GCM)

- S4: Attacco Man In The Middle in corso realizzato con MITM su API Rest e nessuna difesa attiva
- S5: Tentativo di attacco sventato con HTS Poisoning / ARP Poisoning e difesa Mitm Detection App attiva (Diffie Hellman - AEAD con AES-GCM)
- S6: Tentativo di attacco sventato con HTS Poisoning / ARP Poisoning e difesa Mitm Detection App2 attiva (HPKE - AEAD con AES-GCM)

Dai grafici riportati in seguito si evince come il consumo della Ram rimanga pressoché invariato nei vari scenari. Il consumo della CPU invece in tutti i casi vede dei picchi intorno al 50% dovuti all'avvio iniziale della rete, per poi assestarsi in un range compreso tra il 2% e il 10%. Negli scenari S3 ed S6 in cui la difesa HPKE è attiva è possibile notare la presenza di leggeri picchi dovuti alla presenza dei pacchetti di probing, inviati a partire dal decimo secondo di esecuzione e ripetuti ogni venti secondi da quel momento in poi. Negli scenari S2 ed S5 in cui la difesa DH - AEAD è attiva, al decimo secondo di esecuzione viene avviata la procedura di scambio delle chiavi. Da quel momento in poi ogni venti secondi viene effettuato il controllo dello stato di attività degli hosts tramite l'invio di pacchetti di probing. Da notare infine che nello scenario S4 in cui è attivo l'attacco di Man In The Middle e non è presente alcuna difesa, il consumo di CPU non segnali particolari variazioni rispetto allo scenario S1 in cui non è presente alcun attacco e non è attiva alcuna difesa.

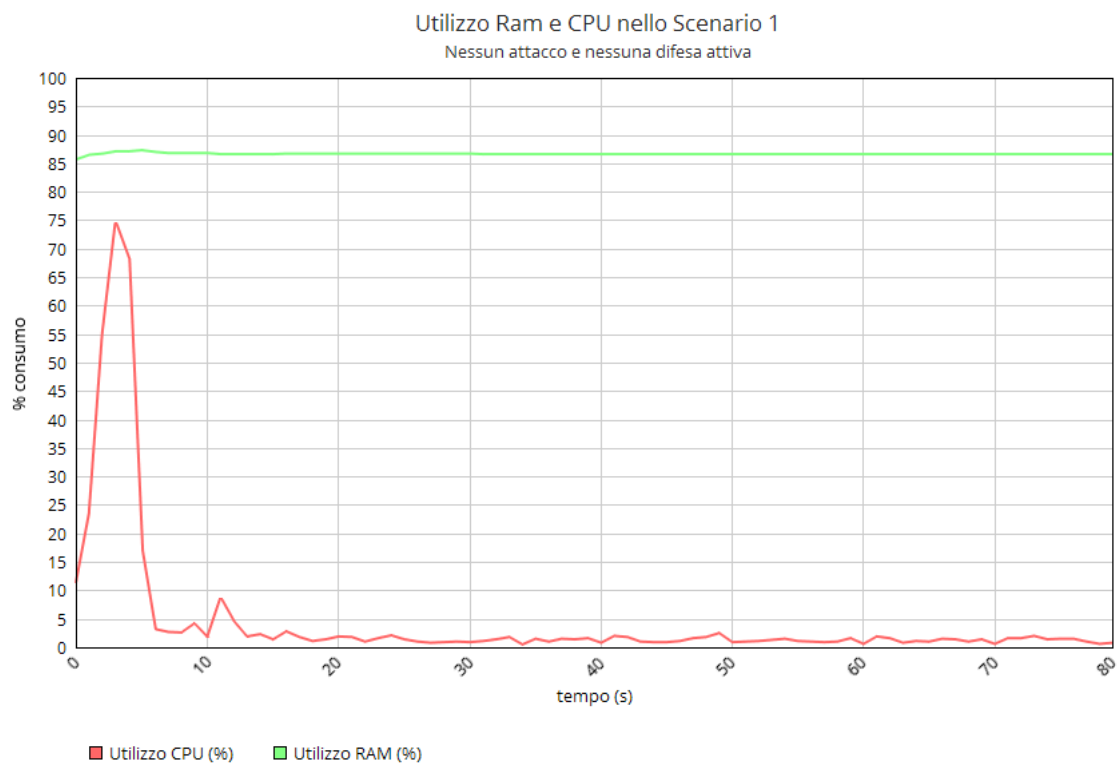


Figura 3.15: Consumo delle risorse hardware nello scenario S1.

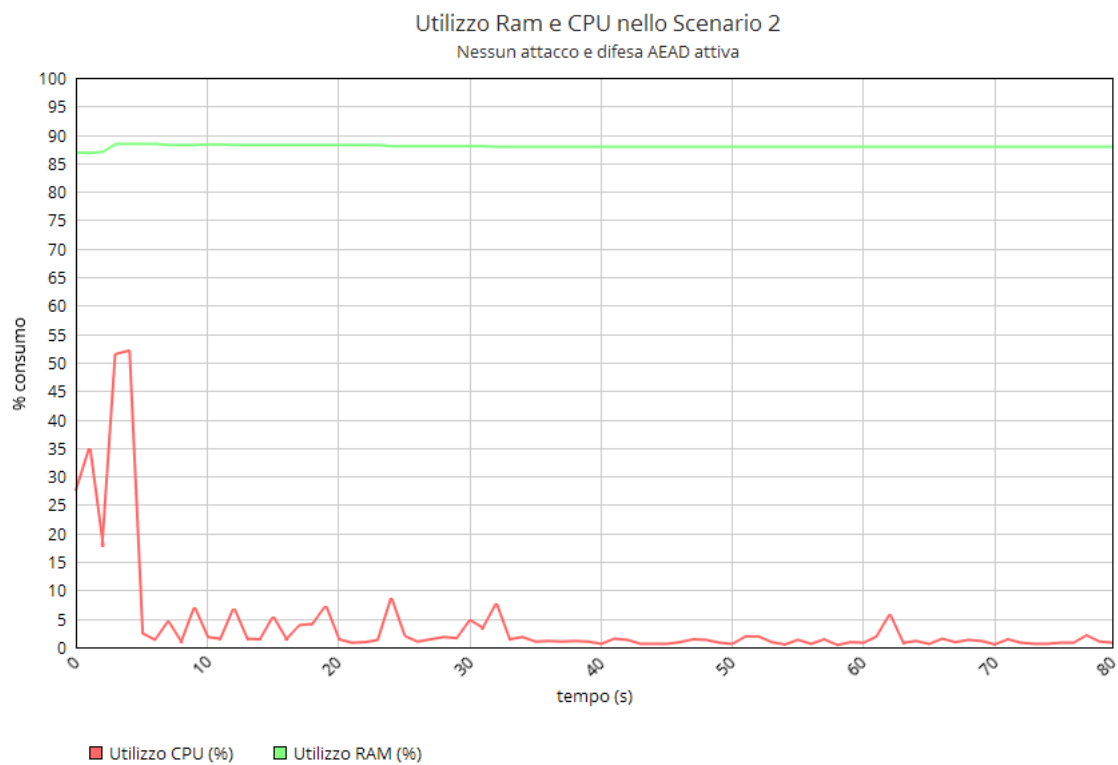


Figura 3.16: Consumo delle risorse hardware nello scenario S2.

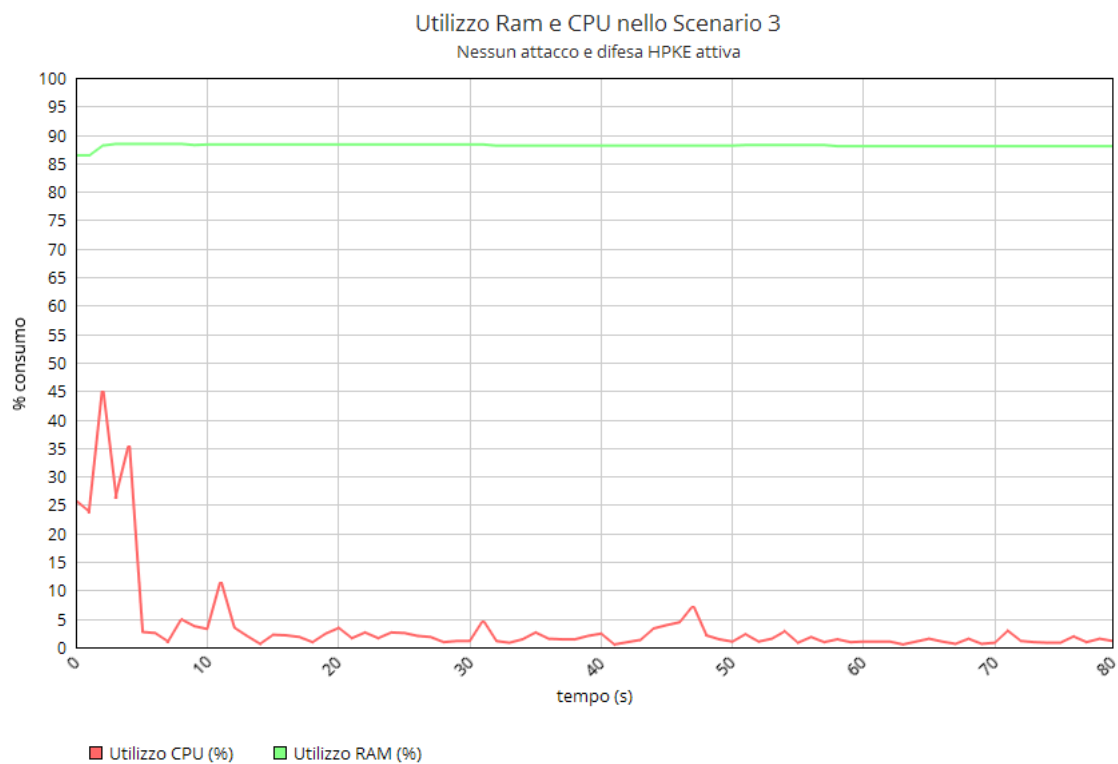


Figura 3.17: Consumo delle risorse hardware nello scenario S3.

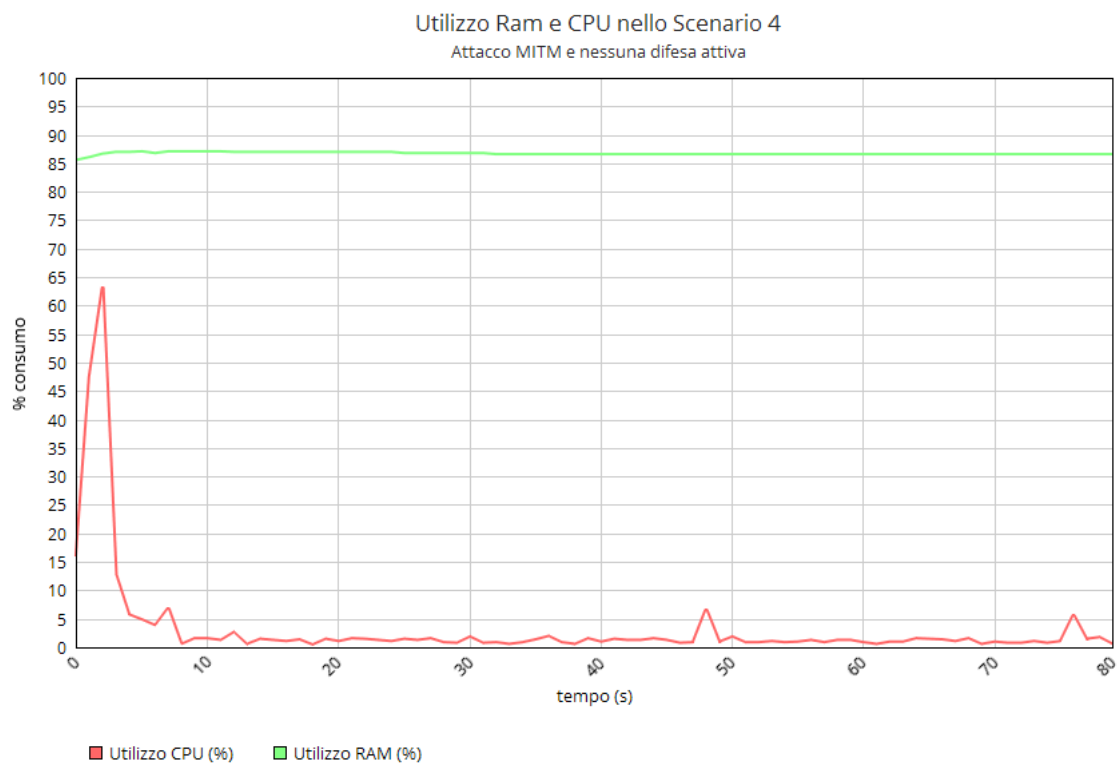


Figura 3.18: Consumo delle risorse hardware nello scenario S4.

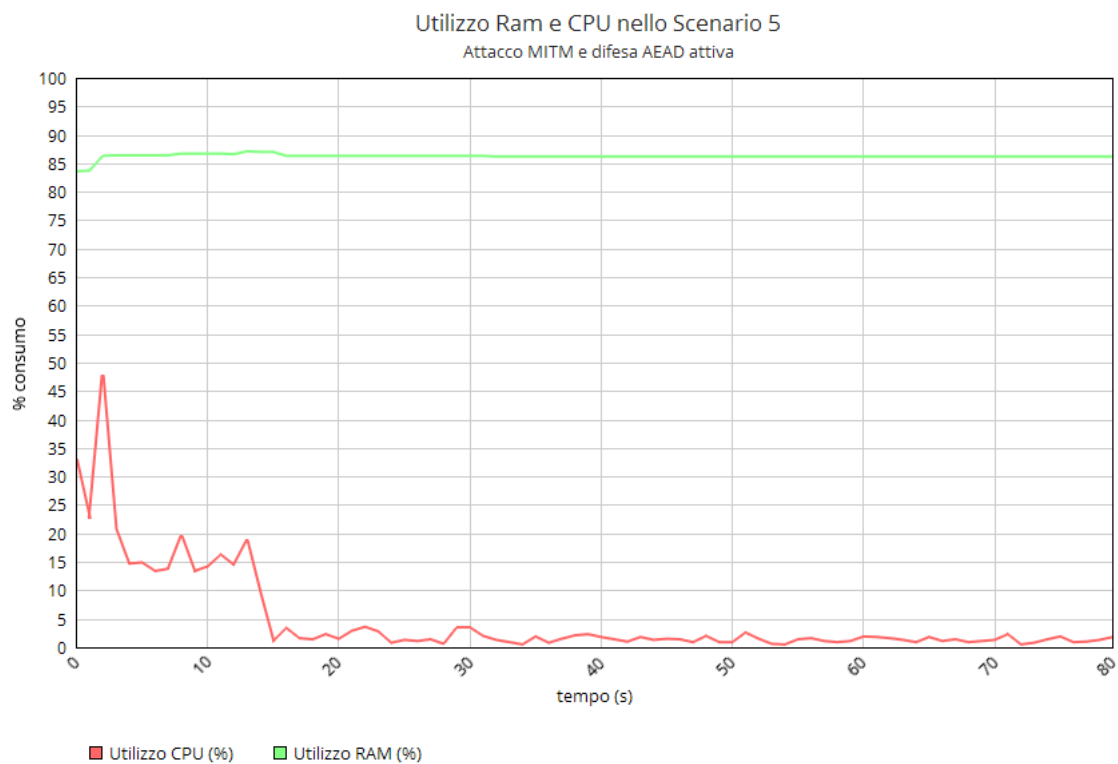


Figura 3.19: Consumo delle risorse hardware nello scenario S5.

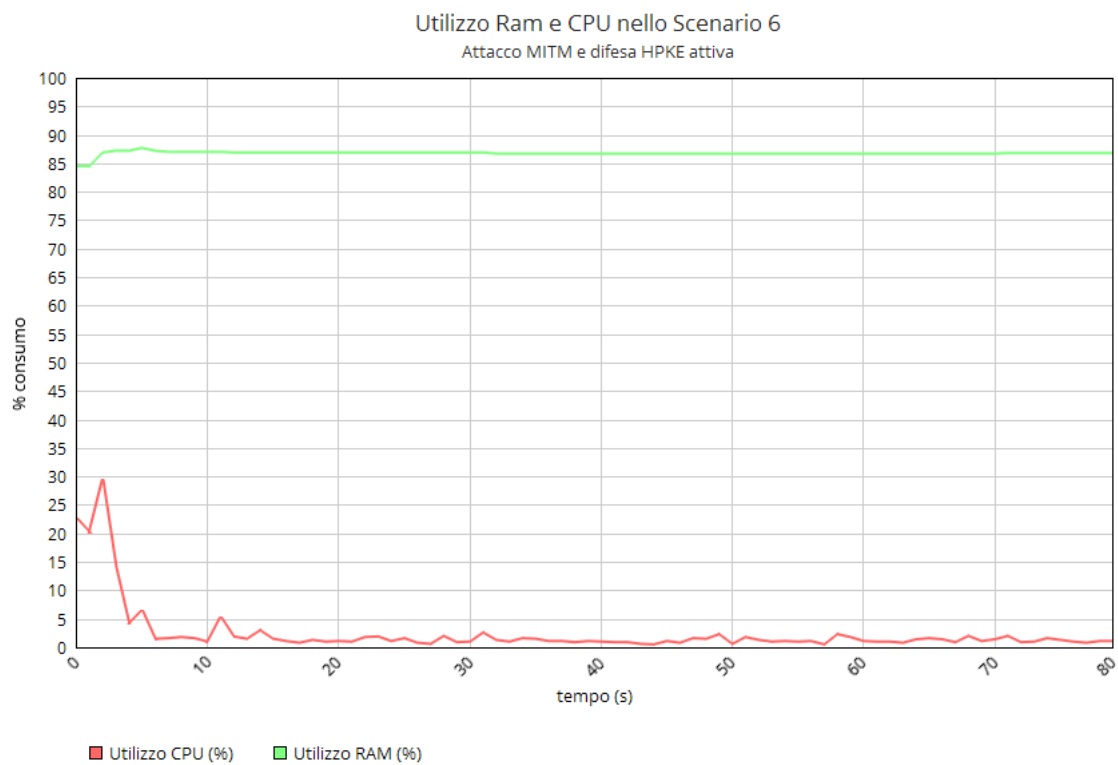


Figura 3.20: Consumo delle risorse hardware nello scenario S6.

Capitolo 4

Conclusione

Poter disporre di un Controller Centralizzato che permetta di avere una visione globale di una rete è una delle principali innovazioni introdotte dalle SDN. L'*Host Tracking Service* è uno dei servizi fondamentali per il funzionamento di una rete con un controller SDN, ma può essere facilmente compromesso. In questo progetto, sono stati analizzati diversi modi di eseguire attacchi all'HTS simulando la presenza di host malevoli, e dimostrato che tali attacchi in molti casi portano conseguenze gravi e pericolose per la rete. Inoltre, sono state proposte diverse contromisure efficaci per prevenire gli attacchi all'Host Tracking Service ed espellere gli host malevoli. Sono stati infine valutati i costi introdotti dalle misure di sicurezza e discussi i punti di forza e di debolezza delle soluzioni proposte.