ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA TRIENNALE IN INGEGNERIA INFORMATICA

Tesi di Laurea

In

Laboratorio di Amministrazione di Sistemi T

Caratterizzazione della sicurezza dei controller SDN

CANDIDATO:

Giacobbe Lorenzo

RELATORE:

Prof. Marco Prandini

CORELATORE:

Dott. Andrea Melis

Sessione II

Anno Accademico 2018/2019

Sommario

1 Introduzione

Per inoltrare i dati (sotto forma di pacchetti) i router nelle reti tradizionali usano delle regole implementate nel loro firmware, il che comporta una forte rigidità della rete, dovuta in parte alle grosse differenze tra dispositivi di rete di rivenditori diversi.
SDN è un paradigma di networking che ha ricevuto molta attenzione negli ultimi anni, in quanto risolverebbe proprio questo problema. Riesce a farlo disaccoppiando il processo di instradamento dei pacchetti (Data Plane) dalla logica di controllo (Control Plane), la quale viene trasferita dentro ad un componente software centrale, rendendola facilmente programmabile e modificabile. Mentre il Data Plane non mostra grosse differenze dallo stesso livello nelle reti tradizionali, la grossa novità viene fornita dal Control Plane, il quale è stato trasferito all'interno di un controller il quale gestisce la rete sottostante. Per permettere l'uso di dispositivi di rete anche di riveditori differenti serve però un protocollo che definisce una comunicazione standard tra i due livelli. Pur essendo presenti anche altre soluzioni, come Open Network Enviroment di Cisco e Network Virtualization Software di Nicira, in questa tesi ho deciso di usare OpenFlow, protocollo gestito dalla Open Networking Foundation, in quanto è quello più ampiamente usato.

Pur risolvendo la rigidità, questa nuova architettura offre nuovi punti d'attacco, come il canale di comunicazione tra control plane e data plane oppure il control plane stesso, il quale offre un single-point-of-failure per l'intera rete. Particolarmente le vulnerabilità del controller sono di alto interesso per un attaccante in quanto permettono sfruttandole, di influenzare l'intera rete. Un attacco DoS per esempio, che mette fuori uso il controller rende impossibile inoltrare dati sull'intera rete da lui gestita in quanto i dispositivi di rete non sono in grado di prendere delle decisioni autonome. Visto che al giorno d'oggi la maggior parte delle informazioni viaggiano attraverso la rete, se si vuole arrivare a sostituire i network attualmente in uso con dei network SDN bisogna renderli il più sicuro possibile.

Per questo motivo nasce l'idea di creare SATO (Sdn Attacking TOol) per testare la sicurezza tramite penetration testing. Ovvero di creare un tool che permette di lanciare degli attacchi simulati contro una rete SDN per controllarne la robustezza. In particolare in questa tesi creo un primo modulo di fingerprinting della rete, puntando a ricavare informazioni riguardanti il controller che gestisce la rete. I due compiti di

questo modulo sono come prima cosa individuare la macchina che ospita il controller all'interno della rete sotto analisi e successivamente scoprire l'implementazione specifica di controller che gestisce la rete. Per verificare l'efficacia di questo modulo simulo un network tramite Mininet, che permette di creare un network virtuale facilmente configurabile, rendendo quindi possibile creare qualunque topologia di rete risulti necessaria. Come controller su cui eseguire il fingerprinting ho scelto Floodlight e Ryu per avere una differenza maggiore tra i controller considerati, in quanto implementati in linguaggi diversi. Questa scelta è stata fatta anche per mostrare i limiti che ha l'implementazione attuale di questo modulo, che verranno in seguito discussi e ai quali propongo delle soluzioni.

2 SDN nel dettaglio

2.1 Reti tradizionali

Le reti di comunicazioni consistono in generale di […] hosts interconnessi tra di loro dall'infrastruttura di rete. Questa infrastruttura viene condivisa da tutti i nodi ed utilizza dispositivi di rete come router e switch […] per trasportare (pacchetti di) *dati tra un host e l'altro [1].* Per effettuare il trasporto dei dati, i dispositivi di rete devono avere delle tabelle contenenti delle regole in base alle quali decidono la destinazione dei singoli pacchetti. In particolare *le politiche [...] vengono definite dal management plane, imposte dal control plane ed eseguite dal data plane [2]*. Nelle reti tradizionali control plane e data plane sono integrati in un unico dispositivo, che una volta ricevuto un pacchetto decide, con l'aiuto di *regole inserite nel suo firmware [3]*, come instradarlo nella rete. Questo forte accoppiamento tra data plane e control plane negli switch comporta importanti differenze tra diversi rivenditori e di conseguenza una grande rigidità della rete. Inoltre rende molto difficile l'aggiunta di nuove funzionalità. Per alleviare queste problematiche sono stati messi in commercio numerosi "middlebox", componenti il cui compito è di aggiungere agli switch nuovi servizi, come firewall e load balancing. Questi però ne aumentano notevolmente la complessità.
Nasce dunque l'idea di separare questi due livelli e di passare da numerose regole distribuite su altrettanti dispositivi ad un unico componente che prenda le decisioni per tutta la rete.

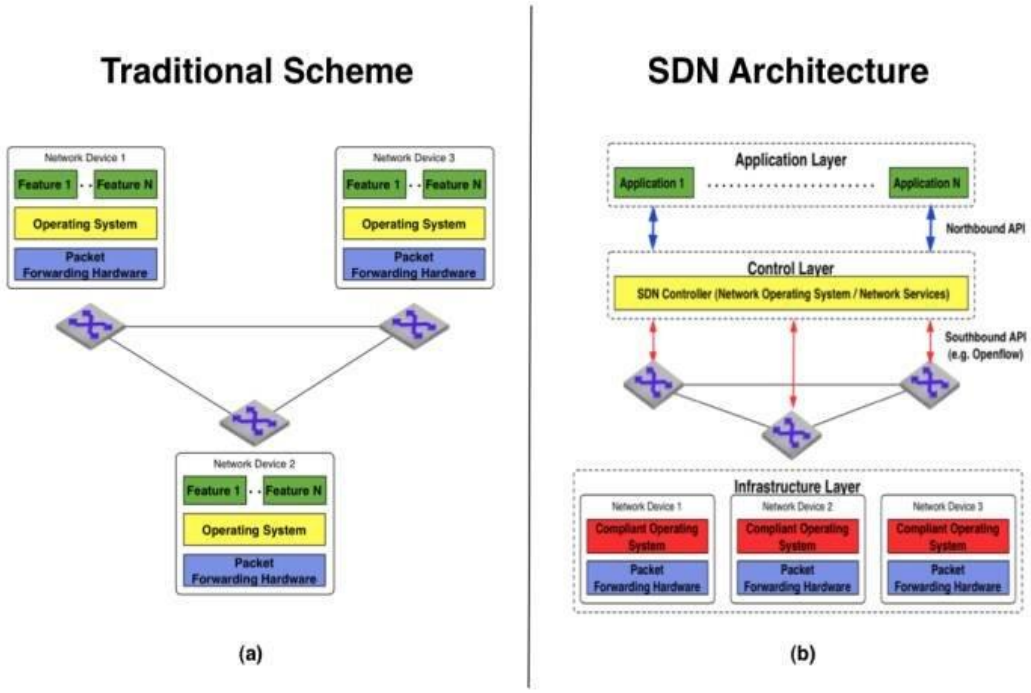


FIGURA 1: DIFFERENZA TRA ARCHITETTURE SDN E RETI TRADIZIONALI

2.2 Sofware defined Networking (SDN)

SDN riduce il compito degli switch al semplice ed efficiente instradamento dei pacchetti in ingresso, trasferendo la parte complessa e dinamica di controllo della rete ad applicazioni software […] chiamate controller [4]. Il controller, avendo una visuale globale di tutta la rete, riesce a fornirne una visione astratta, il che rende possibile aggiungere comportamenti e servizi nuovi tramite applicazioni che lavorano al di sopra del controller. Per via di questo comportamento simile ad un sistema operativo il controller viene spesso anche chiamato Network Operating System (NOS). Avere una visuale dell'intera rete permette inoltre di usare delle regole di forwarding non più basate sulla destinazione di un singolo pacchetto ma su un intero flusso di pacchetti, fornendo così gli stessi servizi che una rete tradizionale può offrire soltanto usando dei middlebox (che però rendono la rete molto complicata). Non essendo le regole incorporate nel firmware degli switch ma gestite da un software, questo riesce a reagire automaticamente ad un cambiamento dello stato della rete mantenendone intatto il funzionamento. SDN non è un paradigma nuovo, ne esistono dunque molte

implementazioni. In questa dissertazione considero lo standard OpenFlow proposto dalla Open Networking Foundation, essendo quello più usato.

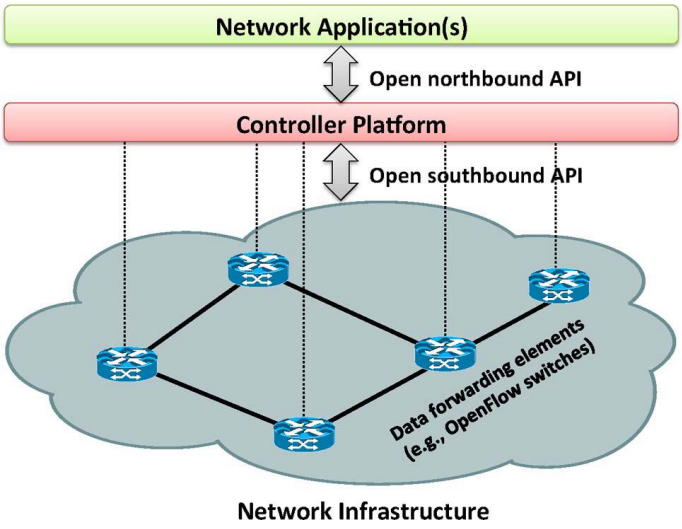


FIGURA 2: ARCHITETTURA SEMPLIFICATA DI UNA RETE SDN

2.3 Struttura

Un'architettura SDN può essere suddivisa in diversi livelli con dei compiti specifici, che analizzo in seguito con un approccio bottom-up.

2.3.1 Infrastructure

Come nelle reti tradizionali, questo livello è composto da diversi dispositivi fisici di rete chiamati switch. A differenza di quelli tradizionali tuttavia gli switch SDN sono caratterizzati da una parte dalla mancanza di logica embedded o software per prendere decisioni autonome; dall'altra da delle interfacce standard che permettono

ai controller di comunicare anche con un vasto numero di dispositivi eterogenei. Le decisioni sulle azioni da eseguire sui pacchetti in ingresso vengono prese grazie alle regole (flow entry) organizzate sotto forma di flow table all'interno degli switch.

2.3.2 Southbound Interfaces (Southbound API)

Visto che nelle SDN la comunicazione tra controller e switch è cruciale per il funzionamento, ne è particolarmente importante la sicurezza e il funzionamento corretto. *Le Southbound Interfaces sono i ponti connettori tra il control plane […] [2]* e gli switch.

2.3.3 Controller/NOS

Il compito principale del controller è quello di offrire un'astrazione della rete sottostante, il che permette ai programmatori di non doversi più preoccupare della raccolta di dettagli di basso livello. Le Northbound API fornite dal NOS allo sviluppatore, gli permettono di eseguire delle funzioni che servono frequentemente, come device discovery o la raccolta di informazioni sulla rete, senza doverle inventare ogni volta da zero. Questo ha l'effetto di favorire e rendere più veloce l'innovazione e la creazione di nuovi protocolli e applicazioni di rete [2]. Essendoci una molteplicità di controller con design e architetture diverse un modo per categorizzarli è vedere se sono distribuiti o no.

Centralized Controller:

In una rete con un singolo controller centralizzato quest'ultimo ha una visuale sull'intero network che deve gestire. Questo lo rende da una parte facilmente implementabile, dall'altra però anche un single point of failure dell'intera rete. Inoltre un unico controller soffre di una scarsa scalabilità, in quanto non riesce a gestire reti di grandi dimensioni.

Distributed Controller:

In un design distribuito i controller possono essere organizzati come un cluster di nodi centralizzato, permettendo un throughput maggiore; oppure come un set di elementi fisicamente distribuiti [2], offrendo invece una maggiore resistenza a diversi attacchi. Sono possibili anche versioni ibride. Mentre in questo modo la scalabilità non rappresenta più un problema, se ne presenta uno nuovo: la consistenza dei dati. Come in ogni sistema distribuito, in caso di cambiamento di un qualsiasi dato questo deve essere notificato a tutti gli altri nodi; compito che può richiedere anche un certo periodo di tempo, nel quale gli altri controller non hanno la garanzia di leggere il dato aggiornato. La consistenza delle informazioni ha un forte impatto sulle prestazioni e viene offerta solo da pochi controller (Onix, ONOS e SmaRtLight [2]). Un ulteriore vantaggio dei NOS distribuiti è la resistenza al guasto, in quanto nel caso di malfunzionamento di un nodo un'altro può sostituirlo, garantendo così la continuità dei servizi. Le comunicazioni tra i vari controller avvengono tramite delle API chiamate westbound/eastbound interfaces.

Alternativamente i controller possono essere suddivisi in controller reattivi o proattivi:

Controller Reattivo:

In un modello di controllo reattivo gli switch devono consultare il controller ogniqualvolta c'è da prendere una decisione [1]. Questo crea un collo di bottiglia che riduce la scalabilità delle reti gestite da questo tipo di controller.

Controller Proattivo:

DIFANE è un esempio di controller proattivo, dove le regole vengono installate preventivamente sui vari switch sottostanti, che in questo modo riescono a prendere delle decisioni autonome [7].

2.3.4 Northbound Interfaces (Northbound API)

Mentre per le interfacce southbound il protocollo OpenFlow è stato ampiamente

accettato come standard, per le interfacce northbound non ne è stato ancora trovato uno, visto che non ne sono ancora del tutto chiari i diversi casi d'uso [4]. Queste API servono alle applicazioni esterne, nel caso in cui vogliano estrarre dal controller delle informazioni sul network sottostante oppure ne vogliano gestire alcuni aspetti.

2.3.5 Application Layer

Le applicazioni possono essere viste come il "cervello della rete". Implementano la logica di controllo che viene tradotta in regole da installare nel data plane, dettandone il comportamento [4]. Le applicazioni comunicano con il NOS tramite le Nortbound API.

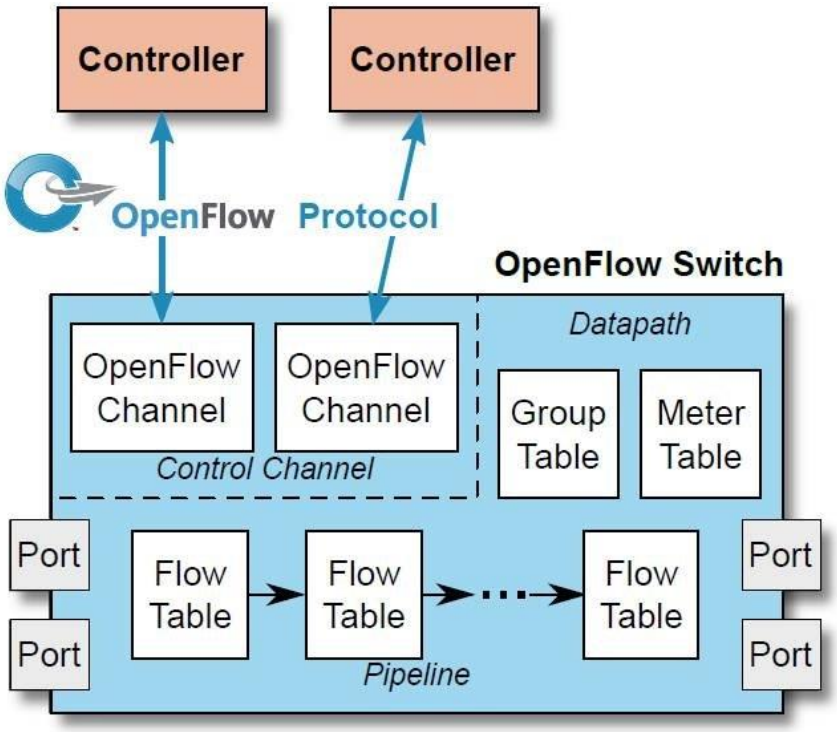


FIGURA 3: STRUTTURA SWITCH OPENFLOW

2.4 Struttura e Funzionamento switch Openflow

Nell'architettura OpenFlow lo switch deve contenere una o più flow table (numerate partendo dallo zero in ordine di priorità) ed un livello di astrazione che comunica in modo sicuro con il controller attraverso il protocollo OpenFlow [1]. Attraverso questo

canale di comunicazione il controller può aggiungere, aggiornare o eliminare le regole contenute all'interno delle flow table (flow entries). Le flow entries sono composte da diversi campi di cui i più importanti sono:

> Match fields: usate per decidere se una entry va applicata ad un pacchetto. Contengono spesso informazioni che vanno confrontate con gli header dei pacchetti.
>
> Counters: contano il numero di pacchetti processati dalla flow entry.
>
> Set of instructions/actions: un action set è un insieme, inizialmente vuoto, di azioni […] che vengono accumulate finché il pacchetto viene elaborato da ogni table e che vengono eseguite nell'ordine specificato […] [10]. Una volta che il pacchetto è passato da tutte le tabelle essi indicano cosa deve essere fatto con il pacchetto.

La comunicazione con il resto della rete, come ricevere ed inoltrare pacchetti, avviene tramite le porte dello switch.

All'arrivo di un nuovo pacchetto lo switch inizia a controllare le flow table, seguendone la priorità, cercando per ogni tabella la prima entry che combaci con il pacchetto. Una volta trovata la entry giusta viene eseguita l'azione specificata nel campo action set della flow entry. Nel caso in cui in una tabella non venga trovato nessun riscontro, viene adottata la table-miss entry, obbligatoriamente presente in ogni tabella [1] (se non è presente il pacchetto viene eliminato). Dipendentemente dalla configurazione esso può eliminare il pacchetto, passarlo alla prossima tabella o inoltrarlo al controller.

Per permettere la comunicazione con il controller, il protocollo OpenFlow impone la presenza di un control channel (connessione TCP potenzialmente criptata con TLS) tramite il quale possono essere inviati messaggi ben definiti, di cui ne approfondisco alcuni nel capitolo quattro.

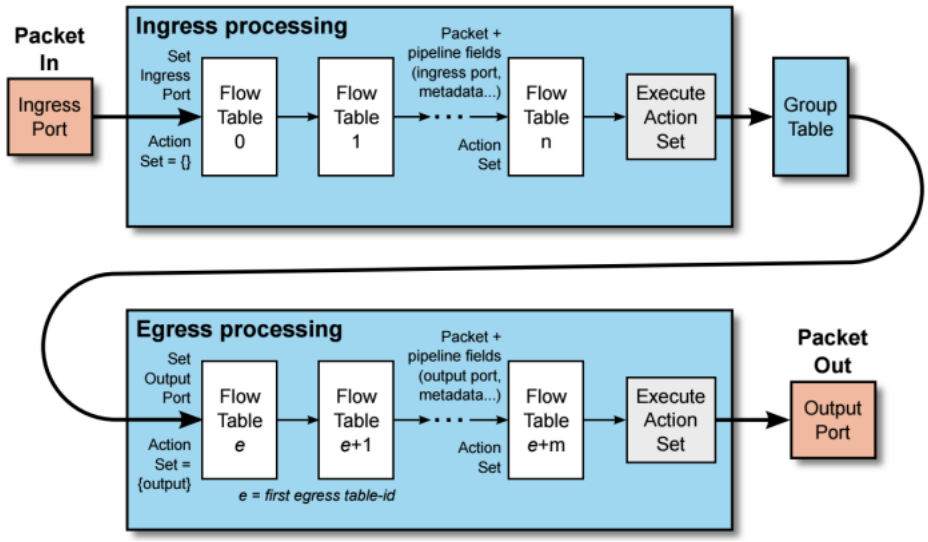


FIGURA 4: PROCESSAMENTO FLOW TABLE

3 Security SDN

Sebbene SDN porti con sé molti vantaggi dal punto di vista della sicurezza rispetto alle reti tradizionali, offre anche diversi nuovi punti di attacco. Mentre per esempio il controllo centralizzato e la visione completa della rete sottostante rendono possibile la generazione di regole più complesse per la gestione del traffico [12], esse semplificano anche un attacco DoS, offrendo un single point of failure ed influenzando così l'intera rete. Il controller tuttavia non è l'unico strato delle SDN ad offrire nuove vulnerabilità.

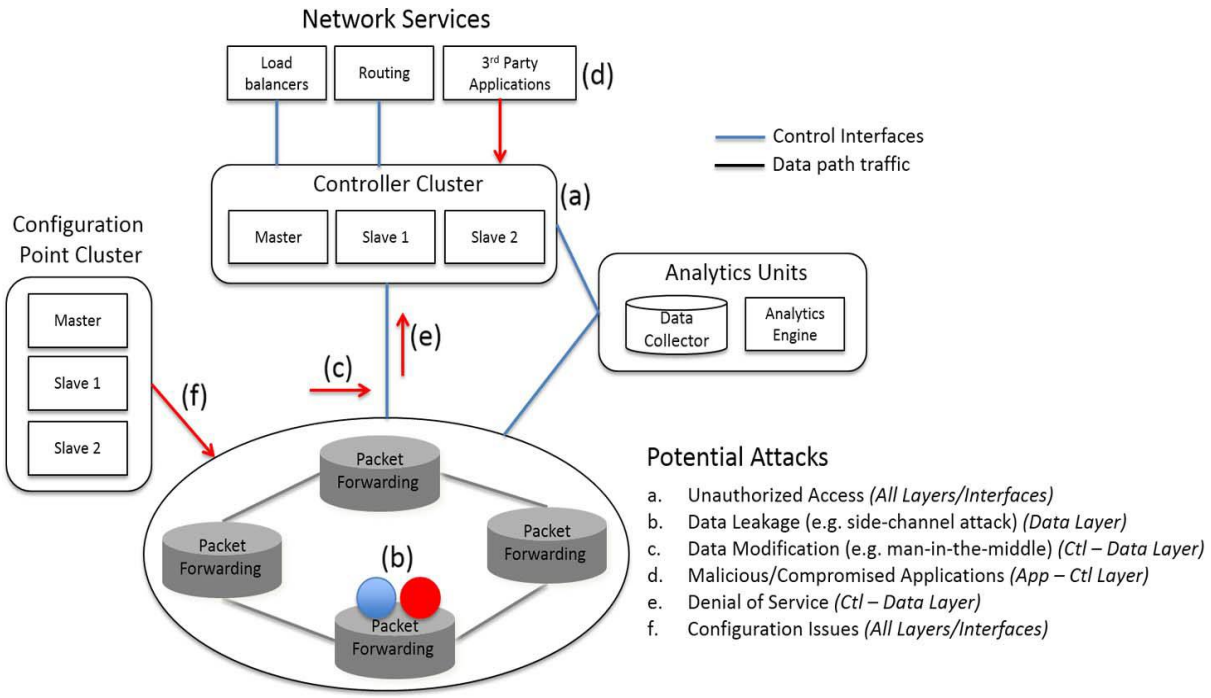


FIGURA 5: POTENZIALI ATTACCHI SU RETI SDN

Elenco in seguito alcune vulnerabilità per ognuno dei nuovi vettori d'attacco introdotti da SDN, ovvero control plane, data plane e control channel.

3.1 Control Plane

3.1.1 Unauthorized Access/ Data Modification

Nelle reti SDN l'accesso fraudolento al controller abilita l'attaccante a controllare l'intero sistema, *permettendogli per esempio di inserire o modificare flow entries negli switch, pilotando così i pacchetti a proprio vantaggio [11]* (portando per esempio ad un attacco Man-in-the-middle). Questo tipo di attacco viene semplificato dal fatto che, pur essendo presenti, le password di protezione sono spesso lasciate ai valori di default.

3.1.2 Data Leakage

Tramite i diversi tempi di processamento dei pacchetti da parte del controller, facilmente calcolabili da parte dell'attaccante, è possibile farsi un'idea della configurazione del controller. I dati ottenuti possono essere usati tra le altre cose a fare reconnaissance della rete e in particolare del controller, come mostro nel

prossimo capitolo.

3.1.3 DoS

Avendo il controller il compito di gestire l'intera rete, un malfunzionamento la metterebbe in ginocchio completamente. Per questo motivo gli attacchi DoS sui controller sono particolarmente comuni. Un possibile attacco DoS contro un controller è il Packet-in Flooding. Uno dei dati deducibile dal tempo impiegato dal controller a processare i pacchetti è l'esistenza o meno di una entry valida per un dato pacchetto. Infatti, in caso di mancanza di una tale entry il pacchetto viene mandato al controller il quale decide come trattarlo; ciò aumenta il tempo necessario al pacchetto per arrivare a destinazione. Un attaccante, una volta capito per quali pacchetti non esistono flow entry, ne genera un grande numero, il che costringe lo switch a inviarli al controller. Il processamento di questi pacchetti rende il controller indisponibile per richieste legittime.

In [17] viene mostrato un'altro attacco DoS che sfrutta il fatto che lo switch, una volta che il suo buffer è pieno di pacchetti ancora da processare, all'arrivo di un nuovo pacchetto lo invia interamente al controller (mentre normalmente viene inviato solo l'header del pacchetto [10]) che poi lo restituisce direttamente allo switch. In questo modo inviando un grande numero di pacchetti è possibile esaurire la banda del controller, o almeno abbassarne le prestazioni.

3.2 Control Channel

3.2.1 Data Modification (Man-in-the-Middle attack)

Dalla versione 1.4.0 il protocollo OpenFlow consiglia l'uso di TLS per criptare il control channel, senza tuttavia imporlo [6]. Questo comporta che di default nella maggior parte dei controller e degli switch viene usata una semplice connessione TCP; in alcuni casi TLS non viene neanche supportato [6]. Ciò permette ad un attaccante di posizionarsi tra il controller e gli switch e per esempio impersonare il controller. In [16] Michael Brooks e Baijian Yang descrivono un attacco man-in-the-middle ottenuto con ARP poisoning contro un controller OpendayLight.

3.3 Data Plane

3.3.1 DoS

Anche a questo livello risulta possibile effettuare attacchi DoS. In particolare in [17] gli autori espongono un'attacco che si avvale della dimensione limitata della memoria

all’interno degli switch per memorizzare nuove flow entry al fine di metterli fuori uso. È da notare che attacchi DoS a questo livello di solito influenzano solo il funzionamento dei singoli switch e non quello dell’intera rete.

4 Fingerprinting SDN controller

Con fingerprinting (o reconnaissance) si definisce il processo di raccolta di informazioni su un dispositivo remoto e sulla sua configurazione. Poiché alcuni attacchi sfruttano delle vulnerabilità non dovute al protocollo SDN ma ad un'implementazione specifica del controller, scoprire quale controller gestisce la rete può essere di grande rilevanza per un'attaccante, e ancora prima scoprire quale macchina della rete è il controller.

Nella prima parte mostro un metodo di reconnaissance della rete tramite *packet-based analysis*, nella seconda invece due metodi che lavorano con la *time-based analysis*.

4.1 Packet based analysis

Nella packet based analysis si cerca di trarre delle informazioni sul network, analizzando i pacchetti che vengono inviati nella rete da analizzare. In questo metodo vengono usati i pacchetti OFTP_HELLO e OFTP_FEATURE_REQUEST brevemente spiegati in seguito.

1. OFTP_HELLO

   Subito dopo la creazione della connessione viene inviato questo messaggio, sia da parte dello switch che da parte del controller, che contiene solo un header e viene usato per la trattazione della versione di OpenFlow da usare.

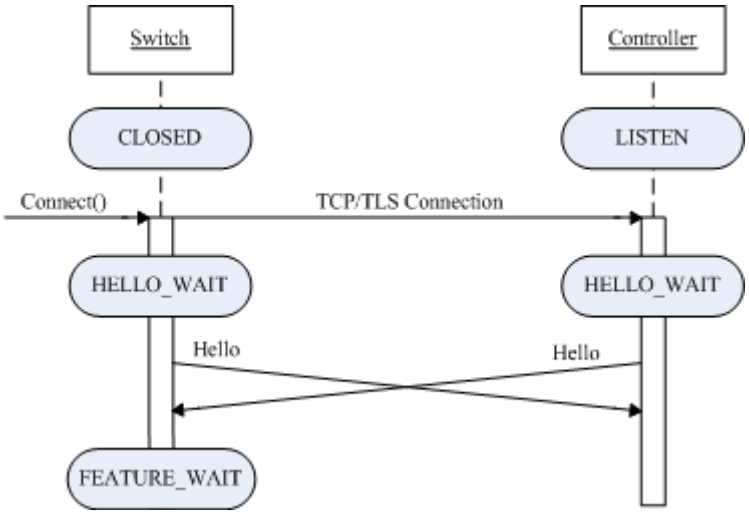
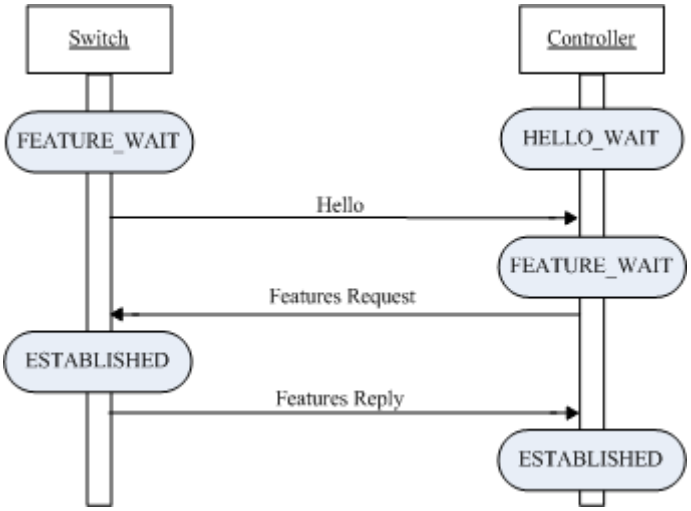


FIGURA 6: SEQUENZA INVIO MESSAGGI OFTP_HELLO

2. OFTP_FEATURES_REQUEST

Dopo che tramite l'invio del messaggio OFTP_HELLO è stata stabilita la versione di OpenFlow da usare il controller invia la OFTP_FEATURE_REQUEST. Lo switch risponde con una OFTP_FEATURES_REPLY nella quale elenca al controller le sue caratteristiche. È importante notare che, al contrario del messaggio OFTP_HELLO, OFTP_FEATURE_REQUEST è un messaggio asimmetrico, ovvero un messaggio che manda solo il controller e al quale lo switch risponde con un messaggio diverso.

FIGURA 7: SEQUENZA OFTP_FEATURE_REQUEST E OFT_FEATURE_REPLY



4.1.1 Pickett

Associati alla sua presentazione “Abusing Software Defined Networks” al DEFCON 22 del 2014 [6], Gregory Pickett fornisce due moduli (of-check.py e of-enum.py) che sfruttano i messaggi sopra nominati per ricavare informazioni sulla rete. Il compito di questi moduli è di scoprire la topologia della rete che si vuole attaccare, anche se si limita ad elencare i ruoli delle macchine della rete senza nemmeno cercare le connessioni tra di loro. L’obbiettivo finale infatti è identificare il controller. I due moduli funzionano in questo modo:

of-check: In questo caso si suppone che l’attaccante sia in possesso di una lista di IP appartenenti a dispositivi della rete, che verranno dati in ingresso a of-check. Facendo un ciclo tra questi indirizzi of-check cerca di stabilire una connessione con ognuno di essi sulla porta 6633, porta standard del protocollo Openflow. Da notare tuttavia che of-check accetta anche porte alternative come parametro. Se risulta possibile creare la connessione significa che la macchina con cui ci si è collegati usa il protocollo OpenFlow essendo in ascolto sulla porta 6633. Una volta stabilita la connessione esso manda un messaggio OFTP_HELLO e aspetta una risposta, dalla quale ricaverà la versione di OpenFlow che viene usata dal dispositivo connesso. Nel caso in cui la connessione fallisca il dispositivo con quell’IP non usa il protocollo OpenFlow. Viene dunque stampata a video una lista di indirizzi che usano il protocollo unitamente alla versione usata, mentre viene salvata in un file la sola lista di indirizzi che può poi essere direttamente usata per il prossimo passo.

of-enum: Una volta identificate le macchine che usano OpenFlow, è necessario capire che ruolo esse occupano nella rete. Of-enum, usando la lista trovata in precedenza con of-check, stabilisce nuovamente una connessione con i vari indirizzi. Dopo aver mandato il messaggio obbligatorio OFTP_HELLO ed aver atteso la risposta invia una OFTP_FEATURES_REQUEST e attende la risposta. Anche qui si presentano due casi: se la risposta è un'altra OFTP_FEATURES_REQUEST, la quale come detto in precedenza può essere inviata solo da un controller, ne risulta che la macchina connessa è un controller. Se invece la risposta consiste di una OFTP_FEATURES_REPLY, essa viene da uno switch.

4.2 Time based analysis

La Time based analysis esegue delle misurazioni temporali per trovare informazione sul dispositivo d'interesse. In [5] viene spiegato come sfruttare questo tipo di analisi per capire il tipo di controller che gestisce la rete, ovvero la Timeout Values Inference e la Processing Time Inference, il cui funzionamento viene spiegato in seguito:

4.2.1 Timeout Values Inference

Per evitare che le memorie all'interno degli switch si saturino, le flow entry delle flow table vengono eliminate dopo un periodo di tempo predeterminato. Tale periodo si chiama *idle-timeout* quando viene calcolato a partire dall'ultimo utilizzo; si chiama *hard-timeout* se calcolato a prescindere dall'ultimo utilizzo. Questi valori sono associati alle entry e hanno valori di default che cambiano al variare del tipo di controller.

Nella Timeout Values Inference vengono calcolati e infine usati questi valori per identificare la tipologia di controller analizzata, supponendo che non siano stati cambiati dagli amministratori di rete.

Algorithm 1 idle_timeout measurement

1: Send first ping to install flow entry;
2: Send n pings and calculate the average ping time RTT_avg;
3: Wait $wait$ seconds;
4: Send one ping and calculate ping time T_{ping}
5: **if** $T_{ping} \approx RTT\_avg$ **then** //the flow entry still exists
6: $wait \leftarrow wait + step$;
7: Go to 3;
8: **else**//idle_timeout expired and the flow entry removed
9: $idle\_timeout = wait$
10: **end if**

FIGURA 8: PSEUDOCODICE PER LA MISURAZIONE DELL'IDLE_TIMEOUT

Algorithm 2 hard_timeout calculation

1: $hard\_timeout \leftarrow 0$ seconds;
2: Calculate RTT_avg as in algorithm 1;
3: Calculate $idle\_timeout$ as in algorithm 1;
4: Send one ping to make the controller install flow entry;
5: Wait $wait$ seconds, $wait$ must be less then $idle\_timeout$;
6: Send one ping and calculate ping time T_{ping};
7: **if** $T_{ping} \approx RTT\_avg$ **then** //the flow entry still exists
8: $hard\_timeout \leftarrow hard\_timeout + wait$
9: Go to 5;
10: **else**//hard_timeout expired and the flow entry removed
11: print $hard\_timeout$
12: **end if**

FIGURA 9: PSEUDOCODICE PER LA MISURAZIONE DELL'HARD_TIMEOUT

Come prima cosa viene calcolato idle-timeout (Figura 8), il che viene effettuato in due passaggi: innanzitutto viene calcolato il Round-Trip Time medio (*RTT_avg*) inviando un numero n di ping ad un'altra macchina della rete (il valore di n, come quello delle altre costanti usate, può essere cambiato dal file fingerprinting.config). Il Round-Trip Time (*RTT*) è la somma tra il tempo che ci mette un pacchetto ad arrivare al destinatario e il tempo che ci impiega la risposta a tornare al mittente. Dopodiché (Figura 9) viene misurato il *RTT* ogni *wait* secondi, valore che viene aumentato ad ogni ciclo di *step* secondi, finché non viene riscontrata una differenza significativa tra *RTT* e *RTT_avg*. *Idle-timeout* corrisponde al valore finale di *wait*. *Una versione più accurata dell'algoritmo può essere ottenuta effettuando una ricerca binaria attorno al valore finale […] [5].* Per calcolare l'*hard-timeout* vengono determinati *RTT_avg* e *idle-timeout* come nell'algoritmo precedente. Infine viene inviato un ping, del quale viene misurato *RTT*, ogni *wait* secondi, con *wait* minore di *idle-timeout.* Nel momento in cui *RTT* è più grande di *RTT_avg* in modo significativo *hard-timeout* equivale al valore di *wait*.

4.2.2 Processing Time Inference

Ci sono diversi controller in commercio, tutti programmati in modo diverso, usando linguaggi, librerie e framework differenti e di conseguenza caratterizzati da tempi di esecuzione diversi [5]. La Processing Time Inference calcola il tempo (*processing_time*) che ci mette il controller per processare un pacchetto nuovo inviatogli da uno switch, e usa questo valore per stimare il controller in uso.

Algorithm 3 Building the processing-time database

1: Calculate RTT_avg as in algorithm 1;
2: Calculate $idle\_timeout$ as in algorithm 1;
3: **for** $i \leftarrow 1..n$ **do**
4: Wait $period$ seconds, $period$ must be greater than $idle\_timeout$;
5: Send a ping and save ping time;
6: **end for**
7: Calculate the average of saved ping time values T_{pavg} and calculate controller processing time $T_p = T_{pavg} - RTT\_avg$;
8: Insert $(controller, T_p)$ in the processing-time database;

FIGURA 10: PSEUDOCODICE PER LA CREAZIONE DEL PROCESSING-TIME DATABASE

Algorithm 4 Fingerprinting $controller$

1: Calculate RTT_avg as in algorithm 1;
2: Calculate $idle\_timeout$ as in algorithm 1;
3: **for** $i \leftarrow 1..m$ **do** //m = 20 for example
4: Wait $period$ seconds, $period$ must be greater then $idle\_timeout$;
5: Send a ping and save ping time;
6: **end for**
7: Calculate the average of saved ping-time values RTT' and compare $RTT' - RTT\_avg$ to the processing-time entries;

FIGURA 11: PSEUDOCODICE PER IL FINGERPRINTING DEL CONTROLLER

Dopo aver calcolato *RTT_avg* e *idle-timeout* come nell'algoritmo precedente, vengono

inviati *n* ping a intervalli di *period* secondi e viene fatta la media dei loro *RTT* (*Tpavg*). *Period* deve essere maggiore di *ilde-timeout* in modo che dopo ogni ping la entry venga rimossa dallo switch. La differenza tra *Tpavg* e *RTT_avg dà* il tempo di processamento del controller. Infatti aspettando ogni volta che la regola sia eliminata dallo switch con *Tpavg* viene calcolato il tempo in cui il pacchetto è in transito più il tempo in cui viene processato dal controller. Sottraendo il tempo di transito medio (*RTT_avg*) avanza il tempo di processamento.

Basandosi entrambi i metodi sulla misurazione del RTT di pacchetti, questi valori possono essere fortemente condizionati da un forte uso della rete e risultare più alti del caso preso in considerazione durante gli esperimenti. Per via dell'aleatorietà di questi valori è consigliato l'uso di entrambi metodi per aumentare le probabilità di un risultato esatto.

4.3 Penetration testing e SATO

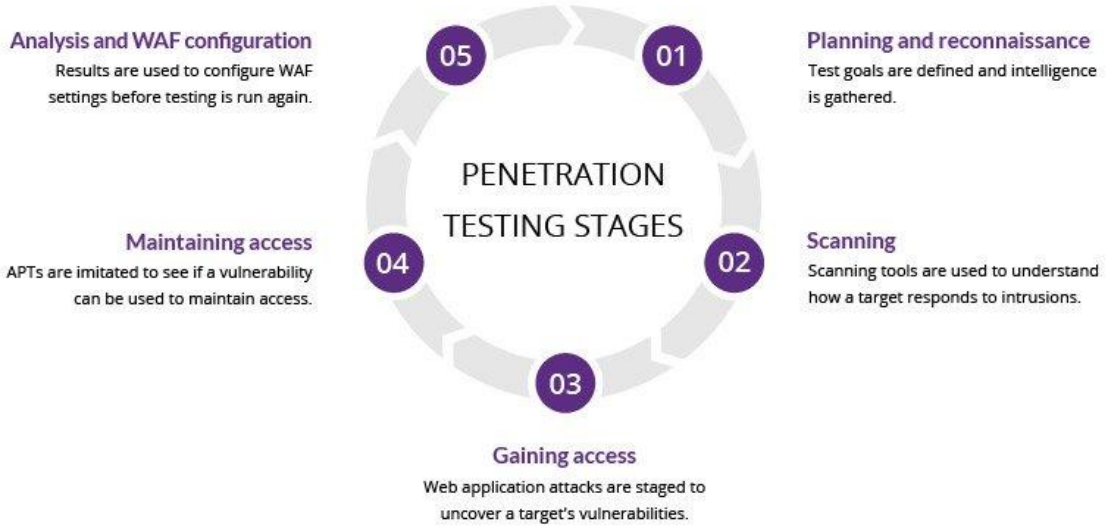


FIGURA 12: 5 FASI DEL PENETRATION TESTING

Un penetration test […] è un cyber attacco simulato contro un proprio sistema per cercare delle vulnerabilità sfruttabili da un potenziale attaccante. [21]

Questi test si svolgono in 5, fasi brevemente riassunte in seguito:

Reconnaissance: In questa fase vengono raccolte delle informazioni preliminari sul sistema da attaccare in modo da pianificare al meglio i possibili attacchi da eseguire.

Scanning: Nella fase di scanning si analizza il comportamento del sistema in risposta a dei tentativi di intrusione e può essere svolta in due modi:

1. Static analysis
   Nell'analisi statica viene analizzato il codice del sistema cercando di capirne il comportamento una volta avviato.
2. Dynamic analysis
   Nell'analisi dinamica il sistema viene esaminato in funzione, mostrandone il modo di rispondere ad un intrusione in real-time.

Gaining access: A questo punto l'attaccante cerca di sfruttare le informazioni raccolte finora per accedere ad alcuni dispositivi della rete e prenderne il controllo.

Maintaining access: Una volta ottenuto l'accesso del dispositivo desiderato l'attaccante cerca di mantenerne il controllo della risorsa permanentemente, per poter raccogliere più informazioni possibili. Tutto questo deve essere eseguito furtivamente in modo da non venire scoperti dal sistema attaccato.

Analysis: Nell'ultima fase vengono analizzati i risultati degli attacchi andati a buon fine per permettere di eliminare le vulnerabilità sfruttate e rendere il sistema più sicuro.

L'obbiettivo dietro alla creazione di SATO è quello di facilitare questi passaggi, facilitando in questo modo la messa in sicurezza delle reti SDN.

In questa tesi mi focalizzo sulla creazione di un modulo per eseguire la prima fase del penetration testing: la reconnaissance. Nelle reti SDN un'informazione particolarmente utile è la conoscenza del controller che gestisce la rete, in quanto permette ad un attaccante di sfruttare non solo le vulnerabilità di SDN o del protocollo OpenFlow, ma anche dell'implementazione specifica del controller usato. Per creare questo modulo ho iniziato implementando gli pseudocodici della Timeout Value Inference e della Processing Time Inference (Figure 8-11) forniti in [5] creando *fp-timeout.py* e *fp_processing_time.py*. Infine ho scritto un'interfaccia (*reconnaissance.py*) per gli utenti che permette di scegliere gli attacchi da lanciare (*of-check.py, of-enum.py, fp-timeout.py* e *fp-processing_time.py*) ed inserire i dati richiesti dai vari attacchi. Per potere inserire meglio *of-check.py* e *of-enum.py* nell'interfaccia ho apportato alcune modifiche, che puntano anche a semplificarne l'uso.

5 Ambiente di testing

5.1 Mininet

Ho usato Mininet 2.2.2 [18] per simulare una rete su cui potere testare il modulo creato. Mininet è un simulatore di rete che permette di far partire un intero network, composto da un numero flessibile di controller, switch e host, su una singola macchina Linux. Questa flessibilità nel potere scegliere la topologia della rete rende possibile eseguire i test su tante configurazioni diverse in tempi ridotti, in quanto ci mette solo pochi secondi a fare partire la rete desiderata.

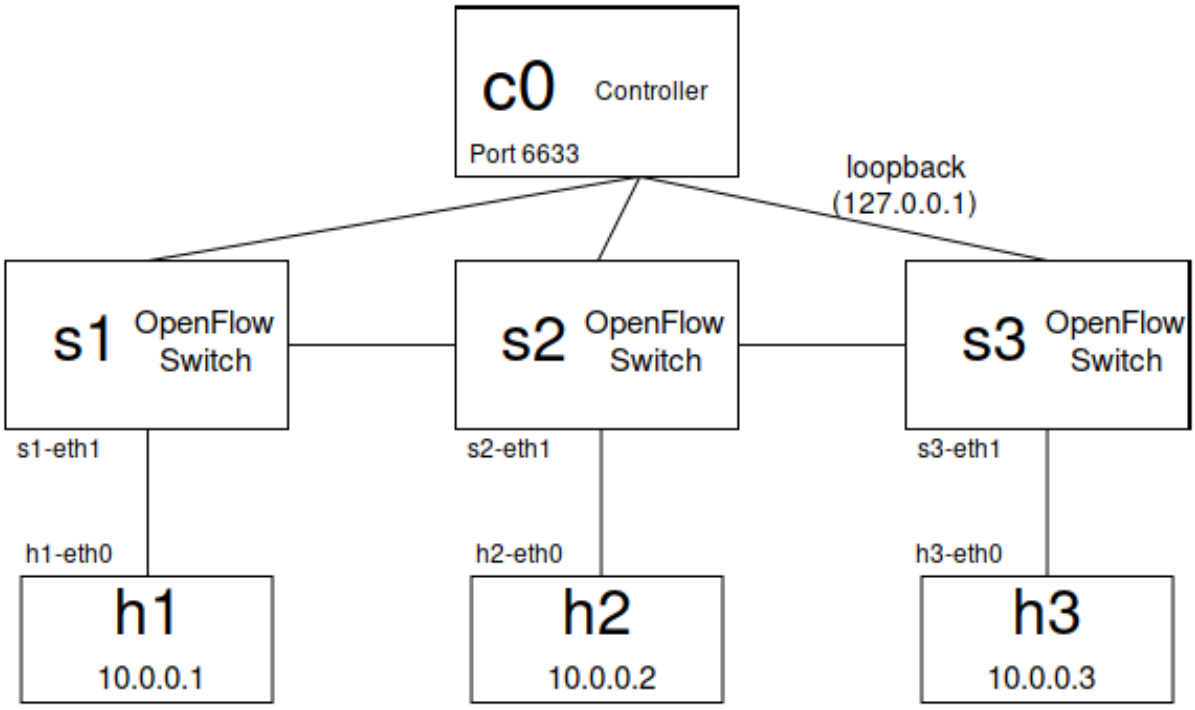


FIGURA 13: TOPOLOGIA MININET

In questa tesi la topologia usata (figura 13) viene creata tramite il comando seguente:

```
sudo mn --topo linear,3 --mac --switch ovsk --controller remote,ip=192.168.56.20x
```

dove:

- --topo linear,3 crea una topologia formata da 3 switch in parallelo, collegati ad un controller e ad ognuno di essi collegato un host
- --switch ovsk indica il tipo di implementazione dello switch da usare. In questo caso si tratta di OVSwitchKernel.

5.2 Macchine virtuali

Per eseguire i vari test ho usato due macchine virtuali (VM) con rispettivamente sistema operativo Ubuntu 14.04.4 e Ubuntu 14.04.3. Mentre ho usato la prima (indirizzo IP 192.168.56.205) per simulare la rete sulla quale eseguire fingerprinting, ho utilizzato la seconda (indirizzo IP 192.168.56.205) per lanciare i controller in modo

da simulare meglio la separazione del control plane dal data plane.

I due controller usati sono Floodlight (versione) [19], supportato dalla Big Switch Networks, il quale è un controller con licenza Apache e basato sul linguaggio Java e Ryu [20] che invece è implementato con il linguaggio Python.

5.3 Pickett

Come già spiegato nel capitolo precedente, questo modulo serve ad individuare l'indirizzo IP del controller all'interno di un pool di indirizzi già disponibili dall'attaccante. Questo attacco deve avvenire da una macchina non situata all'interno del network da analizzare, in quanto non è permesso ad un host contattare il controller e rende perciò impossibile inviarli i pacchetti necessari. Dopo avere quindi creato il network con Mininet sulla prima VM e aver fatto partire prima uno poi l'altro controller sulla seconda VM, ho lanciato *reconnaissance.py*. Prima ho eseguito i test con il controller Floodlight, inserendo nel file con gl'indirizzi da analizzare, oltre all'IP del controller (192.268.56.205), anche altri indirizzi non corrispondenti ad alcun controller. Successivamente ho eseguito gli stessi test usando il controller Ryu.

```
try:
        client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        client_socket.connect((ip, arguments.port))

        # Build
        type = OFPT_HELLO
        length = OFP_HEADER_LENGTH
        xid = 0
        header = struct.pack(OFP_HEADER_FORMAT, OFP_VERSION_1_0_0, type, length, xid)
        client_socket.send(header)

        # Listen for response ...
        data = client_socket.recv(512)

        # Next, Check to make sure data was returned before processing
        if len(data) !=0:

                #
                version, msg_type, msg_length, xid = struct.unpack(OFP_HEADER_FORMAT, data[:8])

                #
                if msg_type == OFPT_HELLO:
                        print('Openflow service (Version: %i) found at %s' % (version,ip))
                        with open(arguments.result, 'a') as res:
                                res.write(ip)

        client_socket.close()

except socket.error:
        pass
```

FIGURA 14: CODICE OF-CHECK

```
if msg_type == OFPT_HELLO:

        # Make Feature Request
        type = OFPT_FEATURES_REQUEST
        length = OFP_HEADER_LENGTH
        xid = 0
        header = struct.pack(OFP_HEADER_FORMAT, OFP_VERSION_1_0_0, type, length, xid)
        client_socket.send(header)

        # Listen for response ...
        data = client_socket.recv(512)

        #
        version, msg_type, msg_length, xid = struct.unpack(OFP_HEADER_FORMAT, data[:8])

        #
        if msg_type == OFPT_FEATURES_REPLY:

                # Acknowledge enumeration
                print('Openflow switch found at %s!' % ip)

        elif msg_type == OFPT_FEATURES_REQUEST:

                # Acknowledge enumeration
                print('Openflow controller found at %s!' % ip)

        else:

                # Only interested in Openflow services
                pass

else:

        # Only interested in Openflow services
        pass
```

FIGURA 15: CODICE OF-ENUM

Mentre *of-check.py* (figura 14) nei test eseguiti è sempre riuscito ad individuare entrambi i controller indicando correttamente nel file dei risultati sia l'indirizzo che la versione di OpenFlow usata dal controller, *of-enum.py* (figura 15) risulta funzionare solo per Floodlight. Questo è dovuto al fatto che il controller Ryu alla ricezione della OFTP_FEATURE_REQUEST cerca di eseguire il parsing del messaggio. Non avendo nessun comportamento programmatto per la ricezione di questo messaggio interrompe l'handshake iniziato.

```
1. Check targets
2. Enumerate targets
3. Timeout Inference
4. Processing Time Inference
5. Help
6. Exit
Please Select: 1
Insert name of file containing the targets (or h for help): targets
Where do you want to save the result? result
Openflow service (Version: 5) found at 192.168.56.206

Openflow service (Version: 1) found at 192.168.56.205

Finished checks!
Results saved in result
```

FIGURA 16: STAMPA RISULTATI OF-CHECK

```
Results saved in result
1. Check targets
2. Enumerate targets
3. Timeout Inference
4. Processing Time Inference
5. Help
6. Exit
Please Select: 2
Insert name of file containing the targets (or h for help): result
Openflow controller found at 192.168.56.206
!
Openflow controller found at 192.168.56.205
!
Finished enumeration!
```

FIGURA 17: STAMPA OF-ENUM

5.4 Timeout Value Inference

Per potere distinguere i controller in base ai risultati forniti da *fp-timeout.py* (figura 18) è necessario come prima cosa trovare i valori di *idle_timeout* e *hard_timeout* di default definiti nei file di configurazione dei due controller analizzati.

```
def idleTimeout(host, config):
        #send pings to install flow rule
        start_time = ping(host, 4)

        #send pings to calculate avg ping time

        avg_time = ping(host, config["pings"])

        # while ping time equals avg_time add step to wait
        wait = 2

        count = 0
        ping_time = ping(host, 1)

        while count < 2:
                wait = wait + config["step"]
                if wait > 15:
                        # idle timeout is infinite
                        return 0
                time.sleep(wait)
                ping_time = ping(host, 1)
                if ping_time > (avg_time + config["accuracy"]):
                        count = count + 1

        idle_timeout = wait
        return idle_timeout

def hardTimeout(host, idle_timeout, config):
        #send pings to install flow rule
        start_time = ping(host, 4)

        #send pings to calculate avg ping time
        avg_time = ping(host, config["pings"])

        hard_timeout = 0
        while ping(host, 1) < (avg_time + config["accuracy"]):
                hard_timeout = hard_timeout + config["hard_wait"]
                if hard_timeout > 40:
                        return 0
                time.sleep(config["hard_wait"])

        return hard_timeout
```

FIGURA 18: CODICE FP-TIMEOUT

```
net.floodlightcontroller.forwarding.Forwarding.idletimeout = 5
net.floodlightcontroller.forwarding.Forwarding.hardtimeout = 0
```

FIGURA 19: IDLE E HARD TIMEOUT DI DEFAULT DEL CONTROLLER FLOODLIGHT

```
def send_flow_del(self, rule, cookie, out_port=None):
    self.send_flow_mod(rule=rule, cookie=cookie,
                       command=self.ofproto.OFPFC_DELETE,
                       idle_timeout=0, hard_timeout=0, priority=0,
                       out_port=out_port)
```

FIGURA 10: IDLE E HARD TIMEOUT DI DEFAULT DEL CONTROLLER RYU

Come si può vedere nella figura 19 Floodlight ha un *idle_timeout* pari a cinque secondi e nessun *hard_timeout.* Le flow entry non vengono dunque mai eliminate finché vengono usate almeno una volta ogni *idle_timeout* secondi. Lo stesso comportamento lo mostra il controller Ryu sia per le flow entry usate che per quelle non usate di recente (figura 20). Questo controller infatti non le elimina finché è disponibile posto nelle flow table degli switch.

Al contrario di *of-check.py* e *of-enum.py* in questi test i programmi vanno eseguiti all'interno del network da analizzare in quanto è necessario inviare dei ping ad un altro host della rete. Per entrambi i controller *fp-timeout.py* è stato lanciato dieci volte, fornendo sempre un risultato esatto per il controller Ryu (figura 22) e risultati con variazioni massime del valore di *step* impostato, per il controller Floodlight (figura 21).

```
1. Check targets
2. Enumerate targets
3. Timeout Inference
4. Processing Time Inference
5. Help
6. Exit
Please Select: 3
IP address of host to ping: 10.0.0.2
Done!
The idle_timeout of the cotroller is: 0
The hard_timeout of the cotroller is: 0
-------------------------------------------------------------
|  Controller    |  idle_timeout (s)  |  hard_timeout (s)  |
-----------------+--------------------+---------------------
|  OpenDaylight  |                  0 |                  0 |
|  Floodlight    |                  5 |                  0 |
|  POX           |                 10 |                 30 |
|  Ryu           |                  0 |                  0 |
|  Beacon        |                  5 |                  0 |
-------------------------------------------------------------
```

FIGURA 22: STAMPA FP-TIMEOUT PER CONTROLLER RYU

```
1. Check targets
2. Enumerate targets
3. Timeout Inference
4. Processing Time Inference
5. Help
6. Exit
Please Select: 3
IP address of host to ping: 10.0.0.3
Done!
The idle_timeout of the cotroller is: 5.0
The hard_timeout of the cotroller is: 0
------------------------------------------------------------
|  Controller    |  idle_timeout (s)  |  hard_timeout (s)  |
-----------------+--------------------+---------------------
|  OpenDaylight  |                  0 |                  0 |
|  Floodlight    |                  5 |                  0 |
|  POX           |                 10 |                 30 |
|  Ryu           |                  0 |                  0 |
|  Beacon        |                  5 |                  0 |
------------------------------------------------------------
```

FIGURA 21: STAMPA OF-TIMEOUT PER CONTROLLER FLOODLIGHT

Essendo questo modulo basato sulla misurazione di RTT dei ping, è fortemente influenzato dall'utilizzo della rete, il che rende necessarie più esecuzioni per ottenere

dei risultati più affidabili.

5.5 Processing Time Inference

Anche in questo caso per eseguire la reconnaissance bisogna trovarsi all'interno della rete SDN. I tempi di processamento non sono definiti all'interno dei file di configurazione dei controller ma dipendono dall'implementazione del controller. Per questo motivo è stato eseguito *fp-processing_time.py* (figura 23) venti volte e infine fatta la media dei risultati ottenuti, in modo da ricavare il tempo di processamento che distingue i controller.

```
def times(host, config):
        #send pings to install flow rule
        start_time = ping(host, 4)

        #send pings to calculate avg ping time
        avg_time = ping(host, config["pings"])

        # while ping time equals avg_time add step to wait
        wait = 0
        while ping(host, 1) < (avg_time + config["accuracy"]):
                wait = wait + config["step"]
                if wait > 15:
                        # idle timeout is infinite
                        return {"idle_timeout":0, "avg_time":avg_time}
                time.sleep(wait)

        idle_timeout = wait
        return {"idle_timeout":idle_timeout, "avg_time":avg_time}

def processingTime(host, idle_timeout, avg_time, config):
        avg_ping_time = 0
        if idle_timeout == 0:
                return 0
        for x in range(config["pings"]):
                time.sleep(idle_timeout + 3)
                avg_ping_time += ping(host, 1)

        avg_ping_time = avg_ping_time / config["pings"]

        # avg_ping_time = ping_RTT + flow entry processing time
        # avg_time = ping_RTT
        processing_time = avg_ping_time - avg_time
        return processing_time
```

FIGURA 23: CODICE FP-PROCESSING_TIME

5.5.1 Test e risultati per il controller Floodlight

Usando il procedimento descritto sopra ho trovato un tempo di processamento per il controller Floodlight pari a 13 ms (figura 24). Questo metodo per il fingerprinting è particolarmente influenzato da altro traffico sulla rete, in quanto i tempi analizzati sono molto ridotti e quindi anche piccole oscillazioni nella quantità di traffico possono facilmente cambiare il risultato; è dunque anche qui consigliato l'attacco più volte e in combinazione con altri moduli.

```
1. Check targets
2. Enumerate targets
3. Timeout Inference
4. Processing Time Inference
5. Help
6. Exit
Please Select: 4
IP address of host to ping: 10.0.0.2
Done!
The processing time of the Controller is: 13.8376
-------------------------------------
|  Controller    |  processing_time |
-----------------+------------------
|  Floodlight    |            13.0  |
|  Ryu           |               -  |
-------------------------------------
```

FIGURA 11: STAMPA FP-PROCESSING_TIME PER CONTROLLER FLOODLIGHT

5.5.2 Test e risultati per il controller Ryu

Allo stato attuale questo modulo non risulta efficace per individuare un controller Ryu (figura 25) in quanto la distinzione tra un controller e l'altro avviene tramite le differenze dei tempi con i quali i controller inseriscono delle nuove regole dentro agli switch. In particolare questo viene fatto trovando prima il RTT dei ping, una volta inserita la flow entry e successivamente sottraendo il tempo trovato al RTT dei ping nel caso in cui la flow entry non fosse presente all'interno dello switch. Avendo i controller Ryu idle e hard timeout infiniti le flow entry una volta inserite non vengono più cancellate. Per questo motivo è necessario calcolare il RTT senza flow entry inserita inviando un solo ping, il che rende il dato trovato poco attendibile.

Per risolvere questo problema, in una versione futura, si dovrebbe rendere possibile inserire un range di destinatari ai quali inviare i ping, anzi che uno solo, permettendo così di avere più dati a disposizione per calcolare un tempo di processamento più attendibile.

```
1. Check targets
2. Enumerate targets
3. Timeout Inference
4. Processing Time Inference
5. Help
6. Exit
Please Select: 4
IP address of host to ping: 10.0.0.2
Done!
Not possible to calculate processing time because idle-timeout is 0
-------------------------------------
|  Controller    |  processing_time |
-----------------+------------------
|  Floodlight    |             13.0  |
|  Ryu           |                -  |
-------------------------------------
```

FIGURA 2512: STAMPA FP-TIMEOUT PER CONTROLLER RYU

6 Conclusione

SDN offre grandi potenzialità, soprattutto per quanto riguarda l'innovazione delle reti e la velocità con la quale queste possono avvenire. Tuttavia prima che possano sostituire completamente le reti tradizioni ne occorre trovare un rimedio al suo maggior punto debole: la sicurezza.

In questa tesi ho analizzato come sfruttare questo punto debole per ottenere informazioni sul controller SDN senza averne accesso diretto. Successivamente ho eseguito questi attacchi su una rete simulata e gestita prima da un controller Opendaylight e dopo da un controller Floodlight per mostrarne la fattibilità e per verificare l'affidabilità dai risultati forniti da questi. I test che ho eseguito si focalizzano solo su due controller tra i tanti in commercio, usando una sola disposizione di rete relativamente semplice. In futuro andrà dunque espanso il numero di NOS sui quali eseguire gli esperimenti e provate configurazioni di rete più vicine ad una rete reale, tenendo per esempio conto del traffico generato dagli altri hosts oppure usando una rete di dimensioni maggiori. Inoltre è necessaria l'implementazione di ulteriori metodi di reconnaissance per garantire un'attendibilità maggiore del risultato finale, in quanto quelli mostrati fanno delle assunzioni sulla configurazione del controller non sempre avverate. Essendo l'obiettivo finale quello di creare un tool di penetration testing dovranno essere sviluppati anche altri attacchi in modo da ottenere tool più completo possibile.

7 Bibliografia:

- A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks [1]
- Software-Defined Networking: A Comprehensive Survey [2]
- A Survey on Software-Defined Network and OpenFlow: From Concept to Implementation [3]
- Flow Wars: Systemizing the Attack Surface and Defenses in Software-Defined Networks [4]
- Fingerprinting OpenFlow controllers: The first step to attack an SDN control plane [5]
- Abusing Software Defined Networks [6]
- Scalable Flow-Based Networking with DIFANE at https://www.cs.princeton.edu/~jrex/papers/difane10.pdf [7]
- Paul Göransson, Timothy Culver, in Software Defined Networks (Second Edition), 2017 at https://www.sciencedirect.com/topics/computer-science/openflow-protocol[8]
- Gary Lee, in Cloud Networking, 2014 at https://www.sciencedirect.com/topics/computer-science/openflow [9]
- OpenFlow Switch Specification at https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf [10]
- A Survey of Security in Software Defined Networks [11]
- Securing the Software-Defined Network Control Layer [12]
- https://sdn.cioreview.com/cxoinsight/security-advantages-of-software-defined-networking-sdn-nid-23290-cid-147.html (Vantaggi delle reti SDN) [13]
- https://www.networkworld.com/article/2840273/sdn-security-attack-vectors-and-sdn-hardening.html (punti di attacco di una rete SDN) [14]
- A denial of service attack against the Open Floodlight SDN controller [15]
- A Man-in-the-Middle Attack Against OpenDayLight SDN Controller [16]
- Denial-of-Service Attacks in OpenFlow SDN Networks [17]
- mininet at http://mininet.org/ [18]
- Foloodlight controller v0.9.0 at http://www.projectfloodlight.org/ [19]
- Codice Tesi at https://github.com/LorenzoGiacobbe/CodiceTesiSDN [20]