



**UNIVERSITÀ DEGLI STUDI DI MILANO**  
**DIPARTIMENTO DI INFORMATICA**

*Corso di Laurea in*  
*Sicurezza dei Sistemi e delle Reti Informatiche*

**Studio ed implementazione di una  
architettura avanzata basata su VPN per  
Security Assessment**

RELATORE

Prof. Marco Anisetti

TESI DI LAUREA DI

Nicola Bena

CORRELATORE

Prof. Claudio A. Ardagna

Matr. 870103

Anno Accademico 2017/2018



# Indice

<b>1</b>	<b>MoonCloud</b>	<b>7</b>
1.1	Overview . . . . .	7
1.1.1	Proprietà, meccanismi, certificati . . . . .	8
1.2	Architettura . . . . .	8
1.3	Non solo cloud . . . . .	10
<b>2</b>	<b>VPN</b>	<b>13</b>
2.1	Introduzione . . . . .	13
2.2	Anatomia . . . . .	14
2.3	Topologie . . . . .	17
2.4	Introduzione alle tecnologie . . . . .	20
2.5	OpenVPN . . . . .	22
2.5.1	Overview . . . . .	22
2.5.2	OpenVPN e MoonCloud . . . . .	23
2.5.3	Conclusioni . . . . .	24
2.6	SoftEther . . . . .	24
2.6.1	Overview . . . . .	24
2.6.2	SoftEther e MoonCloud . . . . .	30
2.6.3	Conclusioni . . . . .	31
2.7	IPsec . . . . .	31
2.7.1	Overview . . . . .	31
2.7.2	IPsec e MoonCloud . . . . .	33
2.7.3	Conclusioni . . . . .	33
2.8	Altre tecnologie . . . . .	34
2.8.1	SSTP . . . . .	34
2.8.2	OpenConnect . . . . .	34
2.8.3	OpenSSH . . . . .	34
2.8.4	WireGuard . . . . .	34
2.8.5	IPsec/L2TPv3 e EtherIP/IPsec . . . . .	35
2.8.6	PPTP . . . . .	36
2.9	Conclusioni . . . . .	36
<b>3</b>	<b>Configurazioni di OpenVPN</b>	<b>39</b>
3.1	SoftEther . . . . .	39
3.2	OpenVPN . . . . .	40
3.2.1	Introduzione . . . . .	40
3.2.2	Configurazione base . . . . .	43

3.3	Problematiche di configurazione . . . . .	50
3.3.1	Impossibilità di aggiungere configurazioni alla rete target . . . . .	50
3.3.2	La soluzione . . . . .	52
3.3.3	Dinamicità della creazione dei client . . . . .	53
3.3.4	Il problema . . . . .	53
3.3.5	La soluzione . . . . .	53
3.3.6	Dinamicità delle rotte server-side . . . . .	54
3.4	IP mapping . . . . .	57
3.4.1	Il problema . . . . .	58
3.4.2	La soluzione . . . . .	62
3.5	Recap . . . . .	67
3.6	Configurazione finale . . . . .	70
3.6.1	Configurazione del server . . . . .	71
3.6.2	Configurazione del client . . . . .	73
3.7	Testing . . . . .	75
3.7.1	Test 1 . . . . .	75
3.7.2	Test 2 . . . . .	81
<b>4</b>	<b>nftables</b>	<b>85</b>
4.1	Prerequisiti . . . . .	85
4.2	Criticità . . . . .	88
4.3	nftables . . . . .	89
4.3.1	Set e altre strutture dati . . . . .	90
4.4	Come è stato usato . . . . .	95
<b>5</b>	<b>Sicurezza della VPN</b>	<b>97</b>
5.1	Sicurezza di OpenVPN . . . . .	97
5.1.1	Protocollo . . . . .	97
5.1.2	Configurazioni . . . . .	99
5.1.3	Opzioni non ancora utilizzate . . . . .	101
5.2	Protezione di MoonCloud . . . . .	103
5.2.1	Politica di default . . . . .	104
5.2.2	Traffico consentito . . . . .	105
5.2.3	Configurazione applicata . . . . .	106
5.2.4	Protezione con nftables . . . . .	107
5.3	Scenari di attacco . . . . .	109
5.3.1	Scenario 1 . . . . .	109
5.3.2	Scenario 2 . . . . .	110
5.3.3	Scenario 3 . . . . .	110
5.3.4	Scenario 4 . . . . .	111
<b>6</b>	<b>MoonCloud_VPN</b>	<b>113</b>
6.1	Requisiti . . . . .	113
6.2	Architettura . . . . .	114
6.3	Dettagli . . . . .	116
6.3.1	Certificate Management . . . . .	116
6.3.2	IP Mapping . . . . .	121
6.3.3	Trasferimento file . . . . .	126

6.3.4	Registrazione server . . . . .	131
6.3.5	Registrazione client . . . . .	135
6.4	API . . . . .	139



# Capitolo 1

## MoonCloud

Questo primo capitolo descrive il funzionamento di MoonCloud, unitamente alle motivazioni per cui la soluzione proposta basata su VPN era necessaria.

### 1.1 Overview

MoonCloud è un framework per la valutazione ed il monitoraggio continuo di servizi cloud. Molto sinteticamente, si definiscono delle proprietà (es: *cifratura dei dati salvati su storage*) e si verifica in maniera continua che tale proprietà sia sempre rispettata. Il rispetto della proprietà porta al rilascio di un certificato che ne attesta il rispetto. Nel momento in cui tale rispetto viene meno il certificato diventa revocato, e, si presume che vengano al più presto applicate azioni correttive per ripristinare la proprietà.

Il framework prevede il coinvolgimento di diversi soggetti:

- *CA – Certification Authority* responsabile del rilascio dei certificato
- *accredited lab* che si occupa dell'analisi e della verifica delle proprietà (può essere il framework MoonCloud stesso)
- *service provider* ovvero lo sviluppatore di di applicazioni basate sul cloud
- *cloud provider*, il fornitore della piattaforma/infrastruttura cloud. Può essere esso stesso a volersi certificare, oppure semplicemente supportare le applicazioni del *service provider*.

Si definisce il concetto di *ToC – Target of Certification* come il contesto di applicazione ed il perimetro del cloud entro il quale si vuole effettuare una valutazione. Il processo di certificazione si svolge in due fasi Nella prima fase avviene la verifica del rispetto della proprietà in un ambiente di laboratorio controllato (*pre-deployment evaluation*), allo scopo di collezionare evidenze. Nel caso in cui le evidenze diano esito positivo, la CA rilascia un certificato per il ToC.

Nella seconda fase, detta *production evaluation*, l'*accredited lab*, mediante test continui, verifica che il certificato sia o meno valido.

### 1.1.1 Proprietà, meccanismi, certificati

Fondamentale è la separazione tra *proprietà* e *meccanismi*.

Una *proprietà non funzionale* è definita come “una proprietà astratta presa da vocabolari comuni o domain-specific, ed un set di attributi che la raffinano” [1].

Un *meccanismo non funzionale* viene definito come “un tipo di meccanismo (es: cifratura) ed un set di attributi che specificano la configurazione del meccanismo (es: lunghezza chiave), unitamente a configurazioni cloud-specific che interessano il comportamento del meccanismo” [1].

Questi due concetti si riflettono su due *modelli* necessari per l’attività di certificazione.

**CM Template – Certificate Model Template  $\mathcal{T}$**  E’ un modello dichiarativo che descrive ad alto livello che cosa deve essere fatto per verificare delle proprietà. Viene definito dalla CA.

**CM Instance – Certificate Model Instance  $\mathcal{I}$**  Si tratta di un modello eseguibile generato istanziando un  $\mathcal{T}$  su un vero ToC.

Il certificato viene rilasciato quando le attività indicate da  $\mathcal{I}$  danno un riscontro positivo, esso contiene una descrizione della proprietà certificata, un riferimento alle evidenze collezionate, ed è legato al ToC.

Affinché le evidenze raccolte ed i risultati derivati da esse siano conformi alla realtà è fondamentale che tutti i modelli usati siano conformi a quelli reali. Ad esempio i modelli di evidenze che ci si aspetta siano corretti devono corrispondere alle evidenze effettive che provino la correttezza. Lo stesso vale per l’intera modellazione del sistema [2].

Oltre al test di proprietà funzionali, recentemente ([2]), MoonCloud ha introdotto anche verifici di vincoli basati sul tempo (es: garantire che un tempo di risposta di una API inferiore ad  $x$  ms), sulla probabilità che certi eventi si verifichino, su particolari configurazioni che devono essere adottati all’interno di un modello più ampio, ed anche alla verifica di attacchi noti.

## 1.2 Architettura

Logicamente, è possibile dividere l’architettura MoonCloud in due macrocomponenti, così come mostrato in figura ???. La comunicazione tra di essi avviene mediante due code:

**Agent Queue** Su di essa vengono inviate richieste di esecuzione di test.

**Evidence Queue** Tale coda è responsabile del passaggio delle evidenze tra il componente di esecuzione e quello di gestione.

I due macrocomponenti sono:

**Certification Manager** Componente responsabile dell’orchestrazione dell’intero processo di certificazione. Ne fanno parte i seguenti moduli:



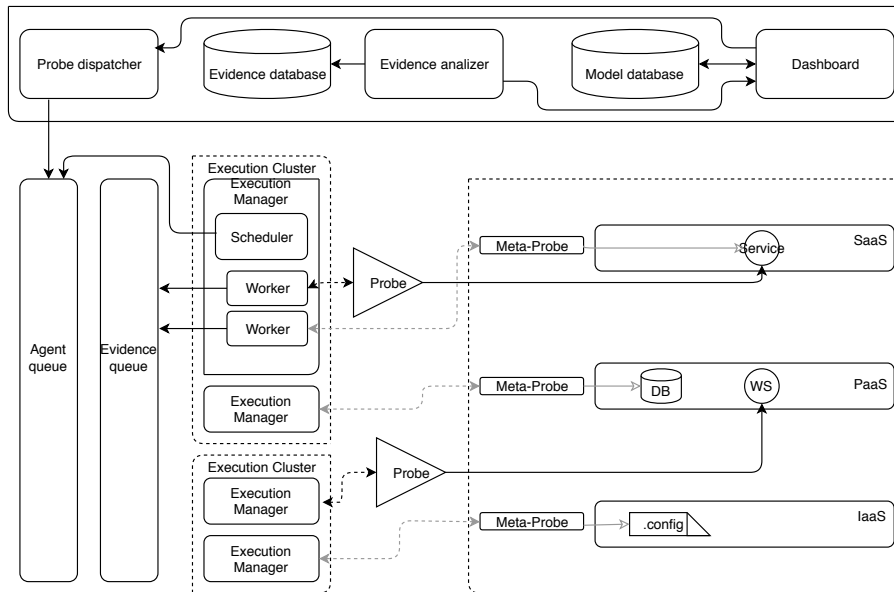


Figura 1.1: L'architettura MoonCloud

- **Dashboard**: una web GUI utilizzata dai clienti per gestire l'intero processo di certificazione
- **Model Database**: una base di dati in cui sono memorizzate tutte le regole relative al processo di certificazione (e di verifica dei modelli)
- **Probe Dispatcher** il quale si occupa della comunicazione tra **Certification Manager** ed i cluster di esecuzione.
- **Result Database** ovvero una base di dati **InfluxDB** in cui vengono salvate le evidenze collezionate durante i test
- **Evidence Analyzer** è un componente responsabile di analizzare le evidenze raccolte nella specifica base di dati.

**Execution Cluster** Un set di VM dedicate all'esecuzione dei test così come orchestrato dal **Certification Manager**. In ciascun cluster vi sono uno o più **Execution Manager**, responsabili della gestione degli agent per l'analisi. Ognuno di questi **Manager** è composto da:

- **Scheduler**: aspetta richieste dal **Certification Manager** e le spaccia al successivo componente.
- **Worker**: esso è connesso direttamente agli agenti che analizzano, manda ad essi le configurazioni necessarie, ne raccoglie le evidenze e le invia al **Certification Manager**.

Gli agent di esecuzione a loro volta si dividono in due categorie:

**Probe** Sono responsabili dell'esecuzione dei test e del monitoraggio in un certo ToC.

**Meta-probe** Si occupano di verificare eventuali violazioni ai vincoli di tempo, probabilità, configurazioni.

Si può dire che mentre il primo tipo sia *invasivo*, il secondo invece ha un minore overhead in quanto si limita a verificare una certa proprietà senza interferire con il normale funzionamento del sistema. Poiché prevedono una forma di analisi meno intrusiva, il minor overhead comporta anche una minore qualità delle evidenze raccolte. Ad esempio, un probe potrebbe utilizzare `nmap` per verificare che una certa comunicazione sia cifrata con TLS, mentre un meta-probe si potrebbe limitare a leggere un file di configurazione alla ricerca dell'opzione che abiliti la cifratura TLS.

Entrambi sono eseguiti come dei container *Docker* su delle *Docker machine* (ci si riferirà ad esse anche come *Docker host*), e si dispone anche di un repository interno di ricette per essi. L'attività di test è eseguita da script Python all'interno di questi container. L'utilizzo di Docker ha un grande vantaggio: mediante i `Dockerfile` si possono specificare tutte le dipendenze necessarie ad un certo test, le quali saranno installate automaticamente in fase di creazione dell'immagine. Non è quindi necessario preoccuparsi di avere una piattaforma di esecuzione sui cui siano disponibili *tutte* le dipendenze che servono.

D'ora in poi ci si riferirà al termine “rete MoonCloud” come a quella delle costituita dalle Docker machine e dal relativo `Execution Cluster`.

## 1.3 Non solo cloud

MoonCloud è stata progettata per poter essere eseguita sul cloud, e, più di tutto, la si vuole *as-a-service*<sup>1</sup>, ciò significa che i clienti che vogliono certificare parte della propria infrastruttura cloud, non devono installare alcun software, semplicemente usano la `Dashboard` per regolare i controlli che vogliono fare.

La verifica del fatto che, ad esempio, si utilizzi uno storage cifrato, può essere effettuata mediante hook messi a disposizione dal cloud provider.

Tuttavia, si è voluto ampliare la platea di possibili target per MoonCloud: si supponga che anziché voler analizzare la configurazione di OpenStack di un deployment pubblico si vogliano verificare che certe policy di `Active Directory` all'interno di una rete aziendale. Anziché utilizzare gli hook messi a disposizione da un cloud provider, occorrerebbe utilizzare le API di `Active Directory`, e come veicolo di accesso ad esse non più (ad esempio) HTTP, ma bisognerebbe aver accesso alla rete interna in cui Windows Server è in esecuzione. Infatti, l'obiettivo di MoonCloud per espandere i propri clienti è quello di non solo essere in grado di analizzare dei target nel cloud, ma anche in una rete aziendale fisica. Ciò pone un grande problema: l'accesso alla rete fisica, infatti non si potrebbero più utilizzare gli hook forniti dal cloud provider.

A maggior ragione, il problema si fa più complesso se si vogliono verificare proprietà relative a servizi *solo interni* di una rete, che non sono esposti all'esterno.

Occorre quindi cambiare “trasporto”, e trovare un modo per avere accesso *dall'interno*, visto che tali reti interne (definite *reti target*) sono ragionevolmente e auspicabilmente protette da un firewall.

Nella fattispecie, si è investigato l'utilizzo di una soluzione basata su VPN, per la quale si porta fisicamente nella rete target un VPN client, responsabile di

---

<sup>1</sup>L'ultimo capitolo è dedicato ad ulteriori evoluzioni dell'architettura MoonCloud

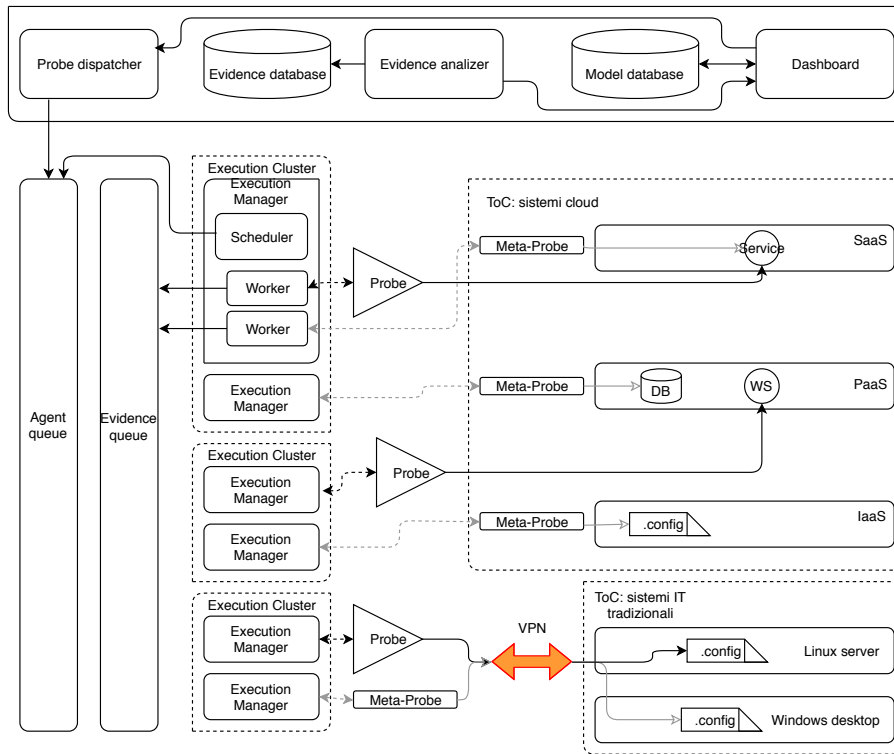


Figura 1.2: L'architettura con l'utilizzo di una VPN

connettersi alla rete MoonCloud e di fornire un canale di comunicazione altamente sicuro. Le Docker machine restano deployate nel cloud in cui MoonCloud stessa è, e le loro richieste di analisi passano tramite la VPN. Questo è stato infatti argomento della prima e più complessa parte della mia tesi.

Sono stati definiti i seguenti requisiti:

- soluzione altamente sicura
- se possibile, basata su TLS/HTTPS. Questo perché si ritiene che almeno il traffico HTTP ed HTTPS sia consentito anche dai firewall più stringenti. Una VPN basata su TLS quindi è ragionevolmente consentita senza necessità di riconfigurare il firewall aziendale. Questo si è anche tradotto in una preferenza di TCP rispetto ad UDP.
- Preferire VPN configurabili e flessibili rispetto a VPN performanti ma poco adattabili ai requisiti.
- La soluzione VPN deve essere il più possibile lightweight per la rete target, ovvero deve poter, nei limiti del possibile, funzionare senza dover configurare nulla nella rete target (ad eccezione del VPN client stesso).



# Capitolo 2

## VPN

Questo capitolo rappresenta la versione modificata dello studio che ho fatto come primo passo della mia attività di tesi, in cui ho fatto una ricerca sulle principali tecnologie di VPN esistenti, allo scopo di individuare quali potessero soddisfare i requisiti di MoonCloud.

### 2.1 Introduzione

VPN è l'acronimo di *Virtual Private Network*, ed indica diverse tecnologie e protocolli utilizzati per, sostanzialmente, connettere tra loro diversi reti locali geograficamente non contigue, oppure per connettere singoli host ad una rete, od anche singoli host tra loro. Questa *connessione* avviene a diversi livelli dello stack ISO/OSI, principalmente al livello 2 od al livello 3.

Si possono realizzare diverse topologie:

**Remote Access** Si realizza questa configurazione quando si vuole un connettere uno o più host ad una rete. Un esempio classico è quello di dipendenti che connettono il proprio dispositivo (PC, tablet, smartphone, etc ...) alla rete aziendale quando non si trovano in sede, potendo così accedere alle risorse che normalmente avrebbero disponibili solo se si trovassero fisicamente nella sede fisica dell'azienda.

**LAN-to-LAN** A volte ci si riferisce a questa topologia anche come *Intranet VPN*. In questo caso, si utilizza una VPN per connettere tra di loro due o più reti che sono geograficamente dislocate in posti diverse. E' molto utile per aziende che hanno più sedi sul territorio (anche in diversi stati), e che desiderano connettere tra di loro le reti di ciascuna sede.

**Extranet VPN** Si tratta di un caso speciale di *Intranet VPN*, e si realizza quando quando l'accesso ad una certa rete viene ristretto, ovvero è possibile accedere solo ad alcune sue parti mediante VPN.

Ma cosa significa davvero *connettere tra di loro più reti*? La domanda sorge spontanea, poiché una volta che due reti sono in Internet, in qualche modo esse sono (indirettamente) connesse tra loro. In Internet, una normale rete privata raggiunge solo le risorse che altre reti hanno rese disponibili su Internet, siano esse dei

server web o dei server mail. Semplificando, si potrebbe dire che le varie reti accedono alle varie risorse delle altre *a livello di protocolli applicativi*.

Nel caso delle VPN, *connettere tra di loro più reti a diversi livelli dello stack ISO/OSI (o TCP/IP)* significa realizzare un collegamento *ad un livello più basso di quello applicativo*. La maggior parte delle VPN realizza un collegamento al livello 2 od al 3, e questo ha i seguenti effetti pratici sulle  $n$  reti connesse tra di loro in VPN:

- livello 2: le reti si trovano all'interno di un unico spazio di indirizzi IP (con stesso indirizzo di rete), ed è *come se fossero separate da uno switch*.
- Livello 3: le diverse reti sono ciascuna in uno spazio di indirizzamento separato, ed è *come se fossero separate da un router*.

Supponendo che vi siano due reti,  $A$  e  $B$ , connesse tramite una VPN al livello 3, un host della rete  $A$  può accedere a *tutte* le risorse della rete  $B$ , non solo quelle che  $B$  espone verso la rete Internet.

Si distinguono tre tipologie di VPN:

**Secure VPN** Una VPN realizzata utilizzando crittografia dei dati in transito tra i diversi host/reti coinvolte, *secure* poiché l'utilizzo di algoritmi crittografici (ammesso che siano utilizzati correttamente) garantiscono che nessun attaccante che intercetti tale traffico sia in grado di decifrarlo o alterarlo in qualsiasi modo.

**Trusted VPN** Sono fornite dagli ISP, i quali garantiscono che nessun'altro cliente sia sullo stesso circuito VPN, *trusted* poiché ci si fida di questa garanzia, inoltre, non vengono fornite particolari proprietà di sicurezza mediante protocolli crittografici.

**Hybrid VPN** Termine che indica una VPN composta da tratti *Secure* e da tratti *Trusted*.

Di seguito, ci si concentrerà esclusivamente sulla prima tipologia, poiché essa è quella di interesse per MoonCloud.

## 2.2 Anatomia

Nell'ambito delle VPN del primo tipo, un principio base su cui si fonda il loro funzionamento è quello dell'*incapsulamento*. Si tratta di un termine ben noto nell'ambito dei protocolli di rete, che può essere sintetizzato come di seguito. Siano  $A, B$  due protocolli di rete, con  $A$  che si trova ad un livello più basso rispetto ad  $B$  (quindi più vicino al *fondo* della pila protocollare). Entrambi i protocolli hanno un *header* ed un *payload*.

Si dice che  $B$  *incapsula*  $A$  quando il payload di  $B$  è costituito dall'intero pacchetto di  $A$ , come mostrato in figura ???. È importante notare che l'incapsulamento può essere ulteriormente generalizzato, non è richiesto che  $A$  sia di un livello più basso rispetto a  $B$ . Nello pila di protocolli di rete, sia essa ISO/OSI o TCP/IP, l'incapsulamento è il modo di procedere normale, per cui ad esempio un segmento

TCP ha nel proprio payload un intero pacchetto IP, il cui a sua volta ha come payload un intero frame Ethernet.

Di seguito si procede a dare una descrizione del funzionamento della maggior parte delle VPN secure, iniziando con il definire con *protocollo VPN* il protocollo utilizzato nel collegamento tra le reti/host. Generalmente si distingue tra VPN client e VPN server, sebbene una volta stabilito il collegamento, il protocollo VPN sia spesso peer-to-peer (un pò come nel protocollo TLS, nella fase di *handshake* vi è una chiara distinzione tra client e server, ma una volta stabilita una connessione le due parti sono di fatto paritetiche); in ogni caso vi è sempre un host che inizia la connessione.

Il VPN server può ricevere più connessioni dai VPN client, i quali in quanto tali iniziano la connessione verso il server. A seconda della topologia/tecnologia, i client possono essere responsabili di connettere alla rete del server la rete a cui essi appartengono; viceversa, il server *può* rendere visibile ai client la rete cui appartiene. Altrettanto opzionalmente, il server può connettere tra di loro i diversi client e le loro reti (nel senso che i client possono comunicare tra loro anziché solo con il server/con la rete del server).

Per poter effettivamente collegare tra loro più *reti*, il VPN *peer* deve avere i seguenti *punti di contatto*:

- collegamento con l'altro peer, raggiungibile tramite la rete Internet;
- punto di ricezione da cui riceve pacchetti provenienti dagli host della rete in cui esso si trova e diretti alla rete dell'altro peer;
- punto di invio al quale l'host invia i pacchetti provenienti dalla rete dietro la VPN e diretti ad host della propria.

Il collegamento di cui al primo punto si realizza spesso (ma non sempre) con un socket, mentre gli ultimi due punti si realizzano con una *scheda di rete virtuale*.

Nel collegamento lungo il socket, la tecnologia VPN definisce un protocollo (che ovviamente garantisca proprietà di sicurezza), può essere un protocollo già esistente come TLS, oppure uno sviluppato ad hoc.

Una scheda di rete virtuale (o virtual NIC – Network Interface Card) è una scheda di rete che esiste nel kernel del sistema operativo ma che non ha un corrispettivo fisico. Se ne può creare una al livello 2 od al livello 3 (utilizzando TUN, modulo del kernel implementato in diversi sistemi operativi). Una scheda di rete al livello 3 è dotata di un indirizzo IP proprio come ogni NIC reale.

Una scheda di rete virtuale è associata ad un software che compie operazioni su essa, e, come per ogni scheda di rete, tali operazioni sono quelle di invio e ricezione.

**Inviare** Il software associato invia o funzione equivalente) un pacchetto sulla NIC, l'effetto è che tale pacchetto viene *ricevuto* dal kernel del sistema operativo e processato come un qualsiasi altro pacchetto, pertanto l'OS deciderà dove e come inviarlo, e se applicare ulteriori trasformazioni.

**Ricevere** Il sistema operativo riceve un pacchetto che ha per indirizzo destinazione quello della scheda di rete virtuale, l'OS quindi inoltra il pacchetto

alla NIC virtuale in questione, l'effetto è che il software associato riceve i dati che il sistema operativo ha inoltrato. Il pacchetto che viene inoltrato alla NIC può essere benissimo un pacchetto inviato da un altro host, e che quindi è arrivato al sistema operativo mediante una scheda di rete reale (oppure un'altra virtuale).

A livello di processo associato alla NIC, *inviare un dato alla NIC* significa chiamare la funzione `write()`, mentre *ricevere* significa utilizzare `read()` (o funzioni equivalenti).

L'idea di base è che ciò che il software VPN legge dalla scheda di rete virtuale è ciò che è destinato ad un altro membro della VPN, e quindi venga incapsulato secondo il protocollo VPN usato. Allo stesso modo, ciò che si riceve dal socket è destinato alla proprio rete (salvo il caso si tratti di un server che connette più client), pertanto viene scritto sulla NIC, per essere dato in gestione al proprio sistema operativo in modo che possa inoltrarlo al destinatario.

Due note fondamentali che occorre sempre tenere presenti:

- se si realizza una VPN al livello 3, tutte le reti partecipanti devono avere degli spazi di indirizzamento diversi.
- In una VPN al livello 2, tutte le reti partecipanti devono stare nella stessa rete IP.

Per capire meglio quanto spiegato, si procede con un esempio. Si supponga ora di voler configurare una certa VPN  $X$ , e che si voglia realizzare una topologia LAN-to-LAN tra due reti  $A, B$ , in cui nella prima si trova il server VPN  $X$ , nella seconda naturalmente si trova il client. Lo scenario è il seguente:

- rete  $A$ : indirizzo di rete:  $192.168.1.0/24$
- rete  $B$ : indirizzo di rete:  $192.168.10.0/24$
- indirizzo interno del server VPN in  $A$ :  $192.168.1.200$
- indirizzo pubblico del server VPN:  $2.7.200.70$
- indirizzo interno del client VPN:  $192.168.10.20$

Per brevità, ci si riferirà al server con  $s$ , ed al client con  $c$ . Si definiscono infine gli host  $192.168.1.5$  come  $a_5$ , e  $192.168.10.5$  in  $b_5$ , ed infine si suppone che gli indirizzi IP dei default gateway delle due reti finiscano in  $.254$ .

Per il momento non ci si concentra troppo sulla configurazione delle rotte, supponendo che i pacchetti arrivino agli host corretti. Si anticipa soltanto che in tutte le reti partecipanti alla VPN (in questo caso  $A$  e  $B$ ), occorre configurare *almeno una rotta*; nel capitolo in cui si descrivono le configurazioni di OpenVPN, questo aspetto viene affrontato nel dettaglio.

Si vede quindi cosa succede quando  $b_1$  vuole comunicare con  $a_1$ , posto che il collegamento VPN tra  $c$  ed  $s$  sia già stato stabilito con successo. Si precisa che si utilizza il termine generico *pacchetto* per indicare un qualsiasi messaggio di un qualsiasi protocollo di rete.



- Il pacchetto da  $b_1$  viene inviato, e quindi ricevuto da  $c$ .
- Il sistema operativo di  $c$  invia il pacchetto alla scheda di rete virtuale della VPN.
- Il pacchetto originale viene incapsulato da  $c$  in un nuovo pacchetto secondo il protocollo VPN, quindi inviato ad  $s$  e da  $s$  ricevuto.
- $s$  decifra (ed effettua verifiche di autenticità, integrità, ecc. . . ) il pacchetto, a questo punto il risultato della decifratura è un pacchetto esattamente uguale a quello generato al punto 1.
- $s$  scrive il pacchetto sulla propria scheda di rete virtuale.
- L'OS di  $s$  riceve il pacchetto, lo invia a  $a_1$ .

Il protocollo di trasporto preferito è UDP in quanto introduce meno overhead rispetto al TCP. A causa dei particolari requisiti di MoonCloud, ci si è concentrati invece su soluzioni che supportassero il TCP, questo perché è possibile che nelle reti target UDP sia bloccato, mentre è infattibili ritenere che TCP stesso sia bloccato. Probabilmente vi saranno delle restrizioni, e nel caso in cui proprio la VPN non riesca a funzionare si può sempre chiedere di aprire una porta sul firewall, l'importante è non essere troppo invasivi.

## 2.3 Topologie

In questa sezione si esaminano le diverse topologie realizzabili per connettere una rete target a MoonCloud. Non tutte le tecnologie consentono di realizzare le topologie qui dettagliate.

**LS - Local Server** In questa configurazione si prevede di installare i VPN server in MoonCloud, mentre nelle reti target si porta (o si installa) il device VPN client, il quale è incaricato di connettersi al VPN server. Un server può essere potenzialmente responsabile per più reti client diverse, anzi questo è ciò che si vorrebbe fortemente. I container che fanno analisi (o l'host su cui sono in esecuzione) sono configurati per inoltrare i pacchetti al VPN server, il quale li invia alla rete target mediante il collegamento VPN.

**RSMC - Remote Server Multi Client** Opposta alla precedente, in RSMC si installa un singolo VPN server nelle reti target. In una configurazione di tipo *Multi Client*, ciascun Docker host è connesso direttamente in VPN con il server, il quale, una volta ricevuti i pacchetti dalla VPN li invia agli host target.

**RSSC - Remote Server Single Client** Simile ad RSMC, il VPN server è presente nella rete target, tuttavia in MoonCloud si realizza, per ciascuna rete target, un unico VPN client, a cui i Docker host responsabili per una certa rete target inviano i pacchetti per tale rete. Il client invia quindi i pacchetti lungo la VPN al server, che quindi provvede ad inoltrarli agli host nella rete target.

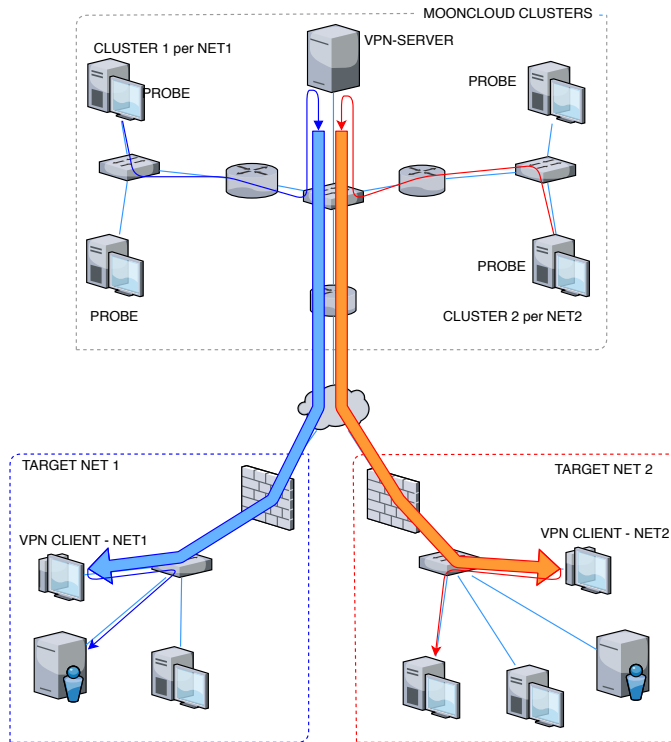


Figura 2.1: Configurazione *Local Server*. Le frecce più spesse mostrano il traffico che transita sulla VPN (di due colori diversi per identificare le due reti), mentre quelle *thin* indicano del traffico non incapsulato.

Nelle configurazioni di tipo *Remote Server*, è necessario che il server sia direttamente raggiungibile da MoonCloud, e quindi deve disporre di un indirizzo IP pubblico. Si è supposto ragionevolmente che la rete target disponga di un collegamento ad Internet, e che sia dietro router/firewall che esegua NAT. Come tale, la rete target è raggiungibile mediante un indirizzo IP pubblico dinamico. Poiché vi è NAT, è fondamentale che il server disponga di un qualche meccanismo di NAT Traversal, poiché si vuole evitare di dover configurare port forwarding sul router del cliente.

Vi sono naturalmente casi in cui non c'è NAT, ma sono una minoranza.

In configurazioni *Local Server* il problema non si pone perché la comunicazione viene *iniziata* dall'interno della rete target.

Già in questa fase iniziale di studio avevo previsto l'utilizzo del NAT *al contrario*, un termine coniato qui per indicare un particolare utilizzo del NAT. Si supponga di essere nella situazione delle due reti elencate precedentemente, ma in cui non vi sia possibilità di intervenire sulle rotte nella rete *A* (non si sono ancora viste quali rotte, ma si è detto che sono necessarie delle rotte introdotte sull'intera rete) perché essa è la rete target, e quindi si vogliono limitare gli interventi in essa. I pacchetti inoltrati da *s* (nella rete *A*) verso la rete e provenienti da *B*, hanno come indirizzo IP sorgente un indirizzo IP in *B*. Una volta che tali pacchetti sono stati ricevuti da un host di *A*, esso non ha modo di sapere che le risposte devono tornare ad *s* (proprio perché non vi sono rotte configurate), e quindi invierebbe il pacchetto al proprio default gateway, che quindi lo dropperebbe (trattandosi di una destinazione con indirizzo IP privato).

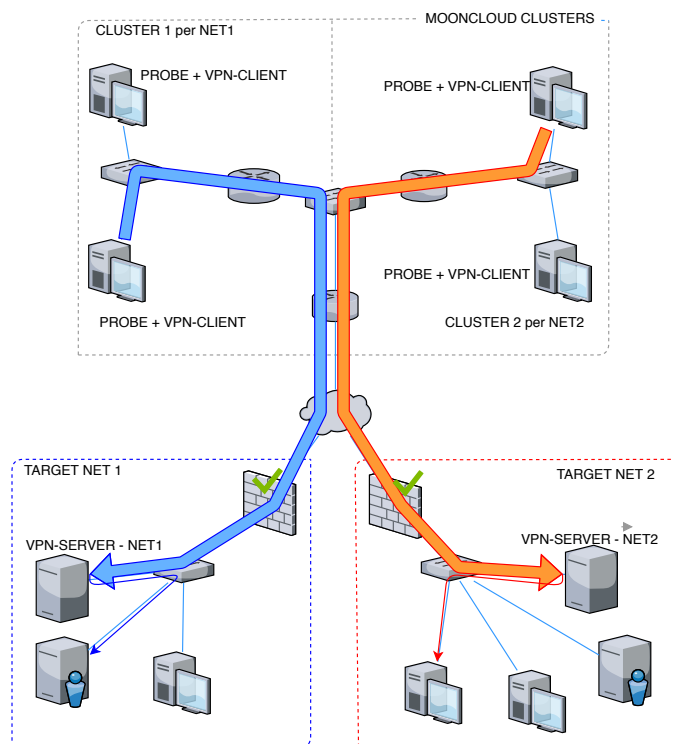


Figura 2.2: Configurazione *RSMC – Remote Server Multi Client*. Ciascun probe è anche un VPN client. La spunta verde sui firewall nelle reti target indica che deve essere consentito l'accesso dall'esterno verso i VPN server.

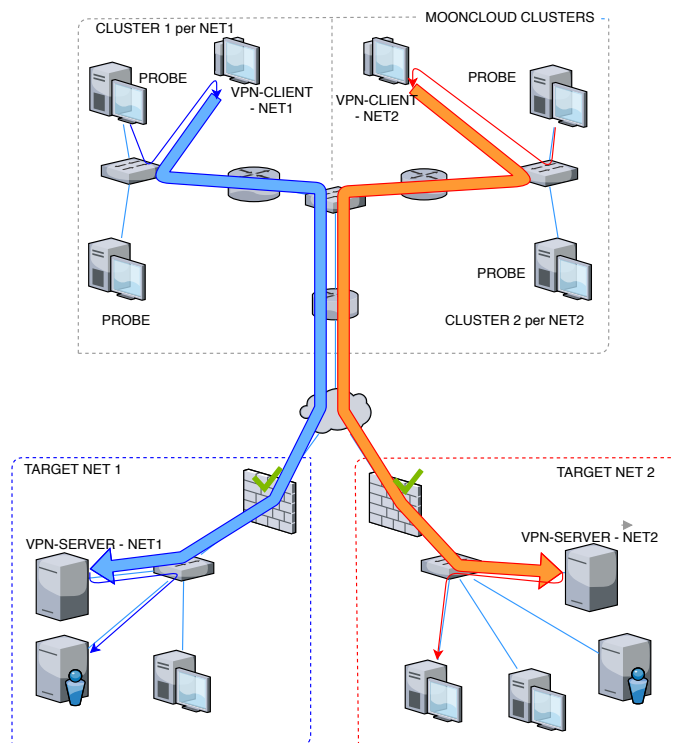


Figura 2.3: Configurazione *RSSC – Remote Server Single Client*. Per ogni cluster dedicato ad una certa rete target vi è una host che svolge il ruolo di VPN client. E' ancora richiesto che i firewall consentano il traffico.

Technology	Protocollo di trasporto	Protocollo incapsulato	Passa fw. stringenti	NAT Traversal
OpenVPN	TCP/UDP	Ethernet/IP	No	Sì, keepalive-based
IPsec (IKEv2)	IPsec	IP	No	Sì
SoftEther	HTTPS (anche ICMP, DNS)	Ethernet/IP	Sì	Sì, parzialmente
L2TP/IPsec (IKEv1)	IPsec	IP	No	Sì (con SoftEther)
SSTP	HTTPS	PPP (IP)	Sì	Sì (con SoftEther)
OpenConnect (Cisco Any Connect)	DTLS e HTTPS	IP	Sì con HTTPS	?
OpenSSH	SSH	Ethernet/IP/TCP	No	?
WireGuard	WireGuard	IP	No	Sì, keepalive-based
L2TPv3/IPsec	IPsec	Ethernet (e altri layer 2)	No	Sì (con IKEv2 o SoftEther)
EtherIP/IPsec	IPsec	Ethernet	No	Sì (con IKEv2 o SoftEther)
PPTP	GRE	PPP (IP and others)	No	No

Tabella 2.1: Principali tecnologie per VPN.

Il *NAT al contrario* consiste nell'applicare NAT sui pacchetti da  $s$  (o da  $c$  a seconda se sia client o server nella rete target) provenienti dalla VPN e destinati alla propria rete: in questo modo raggiungono l'host target con l'indirizzo IP di  $s$ , che si trova nella stessa rete del target, e per tale ragione il target può inviare ad  $s$  le risposte senza passare dal default gateway.

Questo meccanismo è stato descritto in maniere molto sintetica, poiché è una soluzione che è stata davvero applicata, viene analizzato molto più nel dettaglio nel capitolo dedicato alle configurazioni di OpenVPN.

Tra queste topologie, si anticipa che quella scelta è *LS*.

## 2.4 Introduzione alle tecnologie

La tabella 2.4 riassume le principali tecnologie VPN. Nella penultima colonna, con *firewall stringenti* ci si riferisce a firewall che lavorano a livello applicativo.

Prima di proseguire, occorre subito chiarire che il problema principale dell'usare una VPN è che **spesso è necessario configurare almeno un firewall** o default gateway o router di confine. Per questo motivo si cercherà, nei limiti del possibile, di valutare quali opzioni sono più *soft* da questo punto di vista.

Con le VPN si possono realizzare due topologie:

- **Remote Access:** in questa configurazione si connette un host ad una LAN. Sulla LAN sarà necessario installare il componente server, sul PC il componente client dell'infrastruttura VPN<sup>1</sup>.
- **LAN-to-LAN:** si realizza una connessione tra due reti locali, in modo che gli host di ciascuna rete possano connettersi agli host dell'altra, e viceversa (a livello 2 o a livello 3). Per fare ciò, su un host della rete si installa il server, su un host dell'altra il client oppure un altro componente specifico. Attuando le opportune configurazioni, i due host sono responsabili di instradare tutto il traffico della propria rete verso l'altro. Il problema, come si vedrà più avanti, è che per far sì che tutto funzioni occorre aggiungere delle rotte in entrambe le reti.

Esiste anche **extranet**, che corrisponde a Remote Access o LAN-to-LAN con accesso ristretto alle risorse della rete remota. Visto lo scopo di questo documento, non ci si sofferma su questa differenza. Da ora in poi si userà il termine *rete target* per indicare la rete che MoonCloud analizza.

Le due topologie si traducono nel seguente modo per l'architettura MoonCloud:

- Remote Access: in questo caso i container diventano client del server VPN installato nella rete target<sup>2</sup>.
- LAN-to-LAN: i container che fanno analisi e la rete target sono connessi in un'unica rete. Sono possibili soluzioni per cui il server si installa su MoonCloud oppure nella rete target.

Ciascuna di queste due strade ha vantaggi e svantaggi, che verranno dettagliate tecnologia per tecnologia.

Ora si passa a valutare le singole tecnologie VPN. Per ciascuna di esse si fornisce una overview generale, si elencano i passi necessari per raggiungere una certa configurazione (in modo da avere un'idea della complessità), si valutano le diverse topologie possibili. Si conclude infine suggerendo quali sono le opzioni migliori.

Un'ultima nota di proseguire: durante tutto questo capitolo ed il prossimo, si utilizzerà il termine *pacchetto* per indicare una generica sequenza di byte di un generico protocollo, presente in un certo momento in un host dopo essere stata ricevuta o prima di essere inviata, oppure in viaggio sulla rete. Sono perfettamente consapevole che per ogni livello dello stack ISO/OSI vi sia un termine specifico per indicare tale sequenza, ad esempio *frame* per il livello 2 con Ethernet, oppure *datagrammi* per il protocollo UDP, e così via. Tuttavia, come si è visto dalla tabella 2.4, le VPN possono funzionare su diversi protocolli di trasporto (non solo al livello 4), e possono altresì incapsulare pacchetti di livelli diversi. Per questa ragione ho scelto di utilizzare *indiscriminatamente* il termine *pacchetto* per riferirmi ad essi.

Tra le soluzioni indicate di seguito, la prima scelta è stata SoftEther. Per motivi che verranno spiegati in seguito, la soluzione definitiva è poi OpenVPN.

---

<sup>1</sup>Nella maggior parte delle tecnologie VPN vi è una distinzione tra client e server, anche nei casi *peer-to-peer* come IPsec.

<sup>2</sup>Si tenga presente che è possibile *invertire* questo paradigma, ed avere il server in MoonCloud.

## 2.5 OpenVPN

### 2.5.1 Overview

OpenVPN è un software open source molto diffuso per la creazione di VPN, è in grado di incapsulare pacchetti al livello 2 o 3, e come protocollo di trasporto può usare TCP o UDP. E' la soluzione che è stata scelta, ed in questo capitolo se ne presenta una panoramica, mentre viene approfondita nel dettaglio nel prossimo.

Per ciascun tunnel VPN, vengono create due connessioni separate:

- Control Channel cifrato con TLS per scambiarsi informazioni di servizio
- Data Channel cifrato con un protocollo simile a TLS per lo scambio di dati.

Entrambe le connessioni sono multiplexate su di un unico protocollo di trasporto da OpenVPN, sia esso TCP o UDP. OpenVPN è una tecnologia ampiamente usata per creare VPN. La figura 2.4 mostra un semplice schema del suo funzionamento.

OpenVPN viene distribuito anche in una versione *enterprise* a pagamento: si

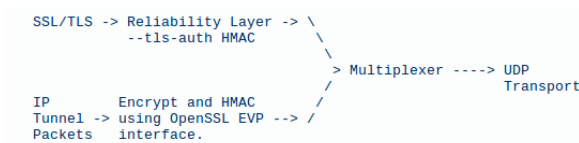


Figura 2.4: Il multiplexing di OpenVPN: si vede come le due connessioni siano multiplexate su un unico protocollo L4, in questo caso UDP.

chiama *OpenVPN AS – Access Server*, offre una web GUI, sicuramente più intuitiva rispetto ai file di configurazione con cui si amministra la versione open source (detta *Community Edition*). D'ora in poi ci si concentra su quest'ultima versione, a livello di VPN le funzionalità sono le stesse; per una comparazione tra le due versioni si veda [3].

L'infrastruttura OpenVPN si compone di:

- server: in grado di connettere tra loro diversi client, è possibile regolare il fatto che diversi client possano *vedersi* tra loro o meno
- client, il quale si connette al server.

E' necessario che il server abbia un indirizzo IP pubblico, oppure dinamico e raggiungibile con un DNS. E' possibile usare servizi che dinamicamente seguono gli aggiornamenti agli indirizzi IP, in modo che i client usino sempre un unico nome per raggiungerlo. Si noti che questa tecnologia (*dynamic DNS*) è integrata in SoftEther.

E' possibile avere il server dietro un firewall, a patto di abilitare il port forwarding dal gateway al server.

L'autenticazione tra i diversi membri della VPN può essere fatta in due modi:

- certificate-based, cioè autenticazione basata su chiavi pubbliche e certificati X509. I passi da seguire sono:
  - creazione di una nuova CA, quindi della coppia di chiavi e di un certificato self-signed
  - ciascun membro della VPN deve avere la propria coppia di chiavi e certificato
  - il certificato della CA deve essere presente in ogni macchina partecipante
- PSK: si generano a priori delle chiavi statiche, che devono essere distribuite su ciascun host membro della VPN. Questa modalità non è consigliata, in quanto se un host venisse in qualche modo compromesso, occorrerebbe cambiare tutte le chiavi. Inoltre non si garantisce *perfect forward secrecy*: se un avversario intercettasse il traffico sull VPN e poi ottenesse queste chiavi potrebbe decifrarlo senza problemi.

E' possibile inoltre configurare anche username e password per ciascun client.

Si possono realizzare sia topologie *Remote Access* sia *LAN-to-LAN*, per ciò che concerne la configurazione effettiva di OpenVPN, le due cose sono praticamente identiche, se si vuole realizzare l'ultima topologia occorre però definire diverse rotte nelle reti partecipanti. In particolare, l'host VPN (client o server) deve essere configurato come gateway per le reti raggiungibili mediante l'altro host VPN a cui è connesso.

Per ciò che concerne la versione free di OpenVPN, client e server sono implementati nello stesso singolo programma.

### 2.5.2 OpenVPN e MoonCloud

Si passa ora ad analizzare le possibilità di implementare OpenVPN in MoonCloud al livello 3. Il primo step da fare è decidere dove posizionare il server: nel cloud o nella rete target?

**Nel cloud** Con questa opzione si preparano nella rete MoonCloud una serie di VM destinate ad essere VPN server; se esse sono configurate propriamente, ciascun server può servire diverse reti target. I probes hanno delle rotte configurate del tipo `rete target via VPN server`. Nelle reti dei client si porta quindi una appliance che quindi client di un server in MoonCloud. Tale appliance deve rendere visibile l'intera rete (possibilmente anche più di una) nella VPN. In particolare si pone un problema di routing nella rete target a prescindere che lì vi sia il server od il client, pertanto è descritto tra qualche riga.

**Nella rete target** Portare un server nella rete target subito un problema: deve essere raggiungibile dall'esterno, e questo non è per niente facile poiché la maggior parte delle reti target utilizzano NAT, e quindi gli host interni non sono direttamente raggiungibili dall'esterno, se non con port forwarding sul router. Chiedere ai clienti di configurare questo non è pensabile,

pertanto questa opzione non è stata consigliata ed infatti non è stata presa effettivamente in considerazione come soluzione praticabile. In via del tutto ipotetica, se si fosse scelta questa strada si sarebbe poi dovuto decidere anche come effettivamente configurare i client VPN. Ad esempio, sarebbe stato possibile connettere direttamente al server ciascun Docker host, oppure dedicare un host al compito apposito di servire i vari Docker host nel collegamente VPN.

Il problema comune ad entrambe queste due opzioni è il seguente: in una configurazione *normale* i pacchetti provenienti dai probes e diretti ai target, una volta che sono decifrati ed immessi nella rete dal VPN client (poiché l'opzione server è stata scartata) hanno come indirizzo IP sorgente un IP che si trova nella rete MoonCloud, ed i target non hanno modo di sapere che le risposte a tali pacchetti devono passare per il VPN client. La soluzione a cui avevo già pensato in questa prima fase di studio è stata quella del *NAT al contrario*, per cui i pacchetti immessi nella rete target dal VPN client vengono NAT-tati ed hanno come IP sorgente quello del VPN client, che si trova nella stessa rete dei target e quindi le risposte torneranno ad esso direttamente.

Senza NAT l'unica strada sarebbe stata configurare delle rotte sul default gateway della rete target o su tutti gli host. In ogni caso, questa assieme a tutte le altre soluzioni adottate per la VPN sono descritte molto dettagliatamente nel prossimo capitolo.

### 2.5.3 Conclusioni

Per ciò che concerne OpenVPN, si è consigliato di realizzare una soluzione *Local Server* per evitare di dover configurare port forwarding nella rete target. Nel caso in cui questo problema non si dovesse presentare, ad esempio perché la rete target dispone di un pool di IP pubblici, sarebbe possibile tentare la strada di un *Remote Server*. A causa di un mio errore, nella prima fase di studio avevo suggerito quest'ultima configurazione.

## 2.6 SoftEther

### 2.6.1 Overview

SoftEther è una tecnologia per VPN molto particolare, sviluppata da uno studente giapponese per la sua tesi di laurea magistrale (ora è diventato *associate professor*), è un progetto attivamente sviluppato, open source e gratuito.

E' disponibile cross-platform, Linux, MAC OS X e Windows. Per le versioni MAC OS X e Windows è disponibile un'interfaccia grafica sia per client sia per server, che consente di configurazioni in maniera molto intuitiva; su Linux bisogna cavarsela con file di configurazione ed una shell (`vpncmd`).

La particolarità di questo software è che in grado di gestire anche *altri* protocolli VPN, oltre a quello sviluppato appositamente per SoftEther. I protocolli supportati sono:

- SoftEther (VPN over HTTPS);



- OpenVPN (L2 e L3);
- SSTP (PPTP over TLS);
- L2TP/IPsec;
- L2TPv3/IPsec;
- EtherIP.

Il grande vantaggio che dà l'utilizzo del protocollo SoftEther è l'incapsulamento in HTTPS, per cui si usa tutta la sicurezza derivante da TLS, unitamente al fatto che praticamente qualsiasi firewall lascerà passare questo traffico.

La figura 2.6.1 mostra come funziona SoftEther.

SoftEther è composto da diversi software:

- *VPN Server*: il componente server in grado di gestire i multipli protocolli VPN.
- *Virtual Hub*: a dispetto del nome è uno switch che viene usato internamente dal VPN Server per collegare i diversi tunnel.
- *Virtual L3 Switch*: anche in questo caso il nome trae in inganno, poiché si tratta di un router che si utilizza in topologie LAN-to-LAN.
- *Local Bridge*: componente del server che viene usato per collegare la rete dei Virtual Hub alla rete fisica su cui si trova il server VPN. E' necessario avere un'interfaccia di rete fisica dedicata che sia connessa ad uno switch (su cui non sia attivo nessun protocollo di rete); è possibile anche utilizzare una scheda di rete virtuale usando TAP al layer 2. E' comunque consigliabile una NIC di buona qualità, che possa lavorare in modalità promiscua. Non è obbligatorio avere questa seconda interfaccia, sebbene sia caldamente consigliato.
- *VPN Bridge* è la combinazione di *Virtual Hub* e *Local Bridge* che consente di collegare tra loro diverse reti in VPN (e non singoli host). E' utilizzabile lato client per realizzare topologie LAN-to-LAN.
- *VPN Client* è il classico componente client che connette un PC ad una VPN. Utilizza il protocollo SoftEther VPN sopra descritto. Come client si può comunque utilizzare un qualsiasi software che usi uno dei protocolli sopra citati. E' da utilizzare solo in configurazioni Remote Access.

Con SoftEther si possono realizzare le topologie:

- **Remote Access.** Si crea in maniera molto semplice una nuovo server, mediante la GUI o la linea di comando. Dal punto di vista logico si fanno i seguenti passaggi:
  1. creazione di un nuovo server;
  2. creazione di un nuovo *Virtual Hub* che "riceve le connessioni";

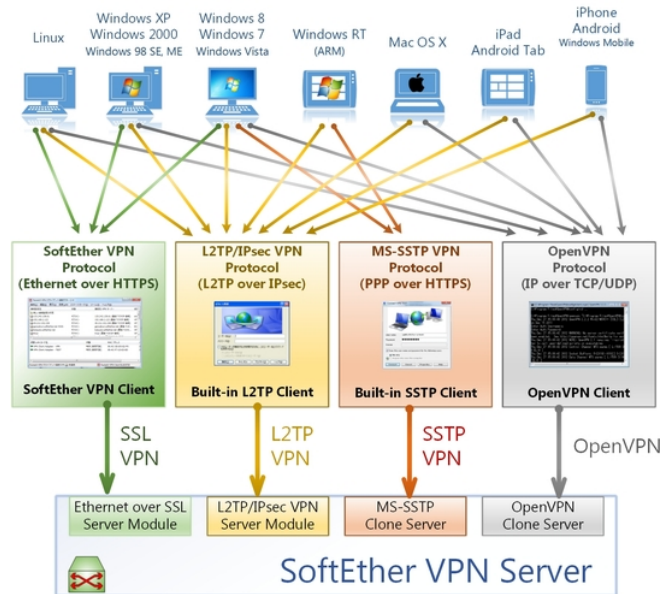


Figura 2.5: Il design di SoftEther. Mediante il *Virtual Hub* si collegano i diversi moduli.

3. connessione del *Virtual Hub* alla scheda di rete del server (possibilmente dedicata a fare questo). Così facendo si dà accesso alla rete in cui si trova il server.
  4. Connessione al server utilizzando il software client.
- **LAN-to-LAN.** E' possibile scegliere se collegare le diverse (non necessariamente 2) LAN a livello 2 o a livello 3, le differenze tra le due sono già state descritte in precedenza. Una volta scelta la topologia che si preferisce si crea un *VPN server*, poi occorre differenziare in base al livello.
    - L2:
      1. sulla rete da connettere al server si crea un nuovo *VPN Bridge*;
      2. si connette il bridge al server (si chiama *cascaded connection*);
      3. si connette il bridge alla LAN locale utilizzando *Local Bridge*.
    - L3:
      1. sul server si creano  $n$  *Virtual Hub*, uno per ogni LAN remota che si vuole collegare; esso "accumula" tutte le connessioni provenienti da ciascuna LAN;
      2. si configura uno *Virtual L3 Switch* che connette i 3 hubs. A dispetto del nome, esso si comporta come un router, per cui avrà  $n$  interfacce di rete, ciascuna di esse con un indirizzo IP compatibile con quello della LAN remota. Per cui l'interfaccia connessa al *Virtual Hub 1* dovrà avere un IP che sia compatibile con gli indirizzi IP della LAN che si "accumula" nel *Virtual Hub 1*.
      3. Per ciò che concerne i client, si utilizza un *VPN Bridge* come per L2.

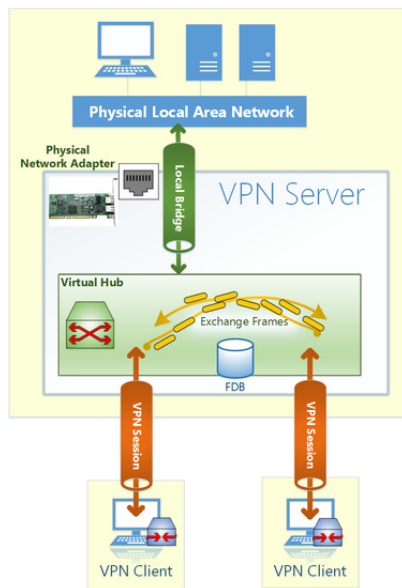


Figura 2.6: Configurazione di SoftEther per Remote Access VPN.

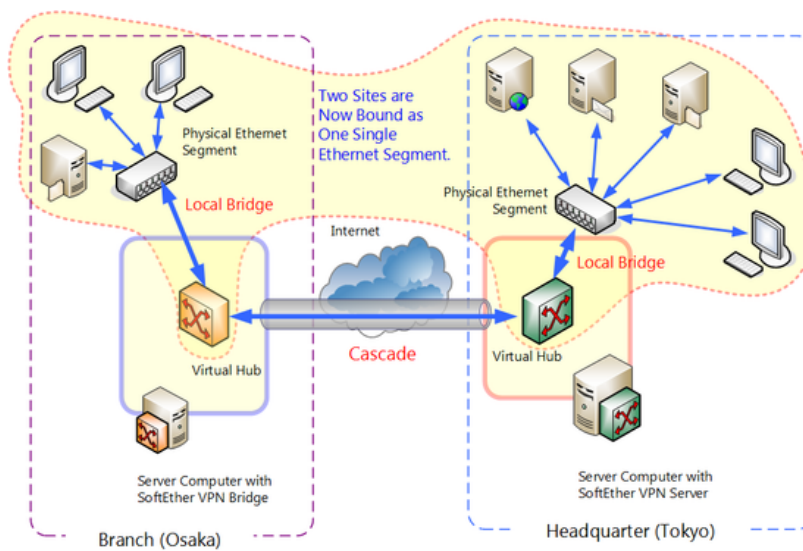


Figura 2.7: Configurazione LAN-to-LAN con SoftEther al layer 2.

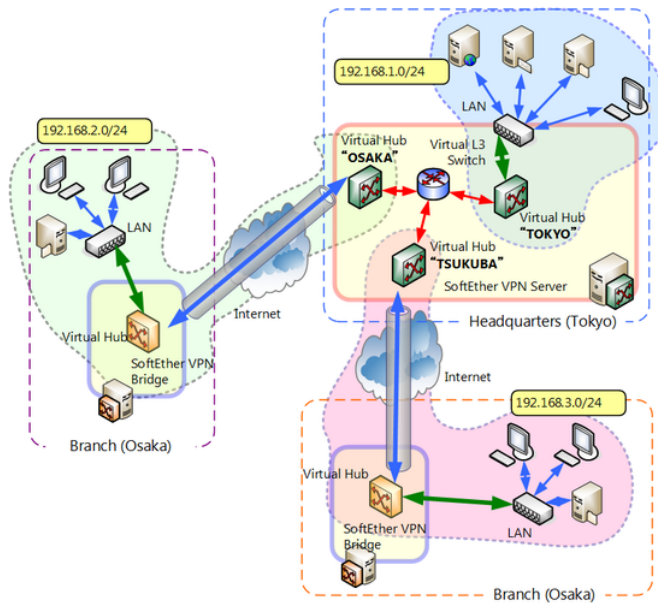


Figura 2.8: Configurazione LAN-to-LAN con SoftEther al layer 3.

4. L'ultimo step prevede di configurare, su ciascun default gateway delle LAN, una rotta fatta così: per raggiungere la LAN  $x$  inoltra all'indirizzo IP relativo nel *Virtual L3 Switch*.

SoftEther possiede alcune caratteristiche molto interessanti, oltre a quelle già citate:

**VPN over ICMP e VPN over DNS** Due opportunità non completamente stabili (il sito ufficiale stesso riporta che a volte si verificano degli errori) per il quale si incapsula il traffico in pacchetti ICMP o DNS per bypassare anche i firewall più stringenti. Occorre valutare *quanto* siano stringenti tali firewall, perché anche queste soluzioni potrebbero non essere praticabili (ad esempio se si usa un server DNS interno e sul firewall si configura l'abilitazione al protocollo DNS solo se proveniente da tale server).

**NAT Traversal** Se il server si trova dietro ad un NAT, è comunque possibile connettersi poiché esso si occupa di fare continue connessioni verso l'esterno in modo che, quando un client tenta di connettersi il NAT lascia passare la connessione, similmente a quanto fatto da Skype.

**Dyamic DNS** Non occorre che si possieda un indirizzo IP statico per identificare il server. Gratuitamente SoftEther registra un nuovo nome DNS per ciascun server che viene installato, ed aggiorna automaticamente tale nome sulla base degli assegnamenti degli ISP (verificando quale sia l'IP del server che si connette all'infrastruttura SoftEther).

**SecureNAT** Si tratta di una tecnologia sviluppata appositamente per SoftEther ed è integrata nel server, e consiste di un DHCP+NAT. Va notato che questa funzione non richiede privilegi particolari, poiché è eseguita in user-space

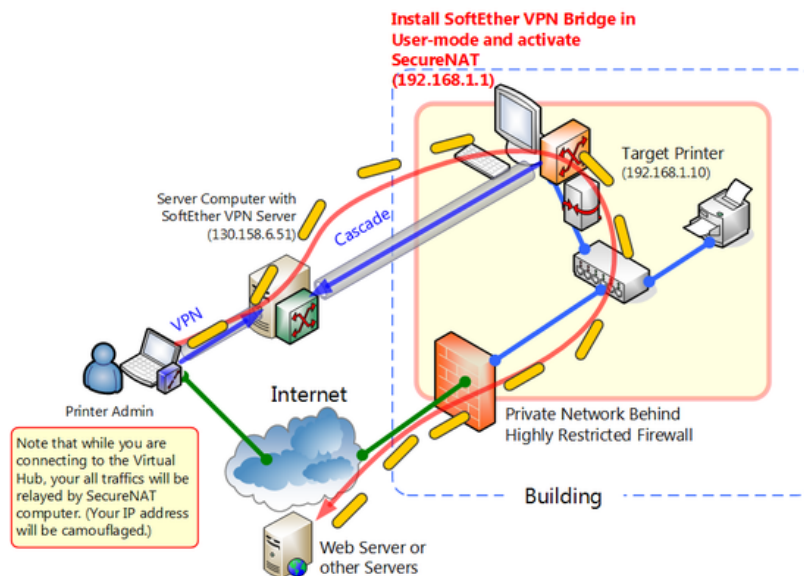


Figura 2.9: La topologia realizzata con *SecureNAT*.

(in Windows, mentre per Linux occorrerebbe verificare se non siano richiesti i privilegi di root). Per ciascun *Virtual Hub* è possibile abilitare questa funzione, ciò fa sì che si crei una nuova interfaccia di rete virtuale connessa al *Virtual Hub* come se fosse un altro computer collegato in VPN, ed infatti, per gli altri host, la scheda virtuale appare proprio come un altro computer. Può essere usato per vari scopi:

- **Network Gateway:** *SecureNAT* può essere usato come alternativa al *Local Bridge* per connettere un *Virtual Hub* alla rete fisica. Si usa una nuova interfaccia virtuale che viene connessa al *Virtual Hub*.
- **DHCP Server:** è possibile scegliere di abilitare solo la funzione di DHCP Server in *SecureNAT*, pertanto tale server assegnerà gli indirizzi IP agli alle interfacce dei client connesse al *Virtual Hub* a cui l'interfaccia *SecureNAT* è collegata.
- **Remote Access.** Questa è in assoluto la caratteristica più interessante, le cui potenzialità andrebbero studiate nel dettaglio. Normalmente, quando ci si vuole connettere ad una rete da remoto è necessario installare su in essa *VPN Server*, e di conseguenza connettersi da un client. Usando *SecureNAT* è invece possibile invertire questo paradigma. La figura 2.6.1 illustra la topologia che si realizza.

Senza scendere troppo nei dettagli, si illustrano gli step necessari. Per *rete remota* si intende la rete a cui ci si vuole connettere, per *rete principale* si intende la rete da cui la connessione parte, ovvero la rete da cui l'utente fisicamente si collega.

1. Sulla rete principale si crea un server con *Virtual Hub*
2. su un host della *rete remota* si installa e si esegue il *VPN Bridge*
3. si abilita la funzione di *SecureNAT* sul *VPN Bridge* (può essere fatto da remoto usando la GUI oppure la command line utility)



Figura 2.10: Comparazione performance delle diverse tecnologie VPN (2012).

4. si configura il server della rete principale per creare una nuova connessione verso il *VPN Bridge*.  
Ciò che accade è che si è stabilita una *cascaded connection* tra il bridge ed il server (il *Virtual Hub* del server).
5. Per utilizzare davvero questa funzionalità l'utente usa un client VPN e si connette al server. La NIC del client sarà configurata dal DHCP Server di *SecureNAT* integrato nel bridge, ed il bridge stesso sarà il suo default gateway. Tutto il traffico del client andrà verso il bridge, e dal bridge poi uscirà su internet (usando l'IP pubblico del NAT dietro a cui c'è il bridge). Grazie a ciò che SoftEther offre, non è necessaria alcuna configurazione nella rete remota (se non l'installazione del *VPN Bridge*).

**Performance** Secondo dei test eseguiti nel 2012 SoftEther *VPN Server* è più veloce (maggiore throughput) di tutte le altre alternative. Il grafico in figura 2.10 mostra il risultato dei test. Si tenga comunque conto che sono benchmark vecchi di 5 anni.

## 2.6.2 SoftEther e MoonCloud

Dopo aver analizzato SoftEther ora si vedono quali sono le possibili configurazioni per un suo uso in MoonCloud. Si è valutata come migliore soluzione quella basata su incapsulamento in HTTPS di L3. La prima distinzione sta nel dove posizionare il server.

**Nel cloud** Installare il server nella rete MoonCloud pone un problema non così banale, cioè la necessità (non mandatoria ma caldamente consigliata) di disporre di una scheda di rete da poter configurare in modalità promiscua, ovviamente a livello di VM. E' possibile che dei cloud provider non consentano questo. Nella rete target si porterebbe un'appliance con *VPN Bridge*.

**Nella rete target** In questa configurazione, se l'implementazione di NAT Traversal di SoftEther lo consente, sarebbe possibile per i client raggiungere il server nella rete target anche se dietro ad un NAT. Occorrerebbe poi valutare

utilizzare un unico client in MoonCloud (RSSC) oppure diversi (RSMC); ovviamente per *unico client* si intende un *unico* client per ciascuna rete.

In ogni caso, nemmeno SoftEther poteva eliminare la necessità di configurare rotte nella rete target, se non con l'utilizzo del già citato NAT *al contrario*.

Oltre a queste soluzioni classiche, si sarebbe potuto sfruttare anche *Secure NAT*. Si tratta di una modalità interessante, che avrebbe richiesto uno studio approfondito per capire come potesse davvero aiutare il nostro use case. In questo caso:

- l'appliance è costituita dal *VPN Bridge*;
- in MoonCloud si installa il server;
- i container sono client VPN che si connettono al server.

### 2.6.3 Conclusioni

Per SoftEther, in forza del NAT Traversal, si è consigliato di realizzare una topologia *Remote Server*, eventualmente combinabile con il Dynamic DNS fornito da SoftEther o con uno gestito da MoonCloud. Un ulteriore ed importante vantaggio è che SoftEther utilizza HTTPS. Per tutte queste ragioni SoftEther è stata inizialmente preferita rispetto ad OpenVPN.

## 2.7 IPsec

### 2.7.1 Overview

IPsec è una suite di protocolli usata per rendere sicuro il livello 3 dello stack di rete. Mediante la crittografia si realizza una connessione che garantisce numerose proprietà di sicurezza. Due protocolli disponibili:

- *AH - Authentication Header*: integrità ed autenticazione (no IP spoofing) del pacchetto con HMAC. E' fondamentale comprendere che questa soluzione non è compatibile con NAT.
- *ESP - Encapsulating Payload*: confidenzialità del pacchetto mediante cifratura simmetrica. Opzionalmente in *ESP* si può abilitare direttamente l'autenticazione e l'integrità, si consiglia questa strada.

Si può abilitare anche la protezione da replay attacks.

Il terzo membro della suite IPsec è *IKE - Internet Key Exchange* che viene utilizzato per negoziare i parametri per i due protocolli precedentemente citati.

IPsec (sia *AH* o *ESP*) può funzionare in due modalità:

- *Transport Mode*: aggiunta di un nuovo header (AH o ESP) tra l'header IP ed il payload. Il suo uso tipico è quello in comunicazioni *dirette* tra due host, che non si affidano a due gateway di confine per eseguire IPsec.

- *Tunnel Mode*: si crea un nuovo pacchetto. Esso è costituito da un nuovo header IP, dall'header IPsec (AH o ESP), e quindi dal payload di IPsec, costituito dal vecchio header IP e dal suo payload.

Per ciò che concerne IKE, è un protocollo abbastanza complesso. Brevemente, la negoziazione dei parametri avviene in due fasi:

- nella *phase 1* viene negoziata una *Security Association* (set di parametri) bidirezionale, utilizzata nella
- *phase 2*; in questa fase si utilizza la SA precedente per negoziare i parametri per la sessione IPsec, tra essi vi sono le chiavi di sessione (stabilite con uno scambio Diffie-Hellman).

Attualmente, è disponibile la versione 2 di IKE. Si tratta di un upgrade importante sotto molteplici punti di vista, ed è fortemente consigliata una implementazione di IPsec che supporti IKEv2. La porta standard utilizzata è la 500 UDP.

L2TP (Layer 2 Forwarding Protocol) è un protocollo standardizzato il cui scopo è quello di realizzare una connessione punto-punto, per cui non include vere funzionalità di sicurezza. L'RFC 3193 (e successive) standardizza l'uso combinato di IPsec per creare una connessione sicura, sulla quale si negoziano i parametri L2TP.

Per realizzare VPN con IPsec tipicamente si utilizzano due configurazioni:

- L2TP on IPsec (negoziare di L2TP su un canale IPsec sicuro—*ESP Transport Mode*);
- IPsec with IKEv2.

Lo stack tipico di IPsec su Linux è il seguente:

- ESP ed AH implementati nel kernel, per cui molto veloci
- demone IKE in user-space, negozia le policy per la sessione e le passa al kernel.

L'implementazione IKE di riferimento per Linux, sebbene sia disponibile anche cross-platform, è *strongSwan*. Maggiori dettagli su essa sono descritti tra qualche riga.

E' possibile realizzare configurazioni Remote Access e LAN-to-LAN. IPsec è uno stack complesso, e configurarlo nella maniera corretta non è facile, tuttavia non è nemmeno impossibile, il sito di *strongSwan* è ricco di esempi e di documentazione. Nella terminologia IPsec con *strongSwan* si definiscono:

- *roadwarrior* è il client, cioè il dispositivo che si connette da remoto alla rete.
- *VPN Gateway* è il dispositivo server che riceve le connessioni.

Il client è l'*initiator*, il server è il *responder*.

Da subito ci si è concentrati su un setup basato su IKEv2, non si è considerato IPsec con L2TP perché considerato *legacy* dall'avvento della nuova versione di IKE.

Utilizzando *strongSwan* sono possibili moltissime forme di autenticazione, distinguendo tra:



- autenticazione degli host, può essere fatta con PSK (fortemente sconsigliato) o certificati X509
- autenticazione degli utenti, in questo è possibile anche appoggiarsi ad un server RADIUS.

Si tenga presente che l'autenticazione degli utenti è fatta *on top* dell'autenticazione degli host. Per ciò che concerne un'autenticazione basata su certificati, occorre seguire gli stessi fatti già descritti per OpenVPN, ovvero creare chiavi/certificato per la CA, quindi generare certificati con essa, ecc. . .

Come per la maggior parte dei software Linux, la configurazione di strongSwan viene fatta mediante file di configurazione. E' di particolare rilevanza configurare correttamente *quali* indirizzi IP vengono incapsulati nel tunnel IPsec, perché le policy presenti nel kernel descrivono anche questo, e se un pacchetto ha sorgente o destinazione IP non corretti sarà droppato dal kernel.

strongSwan è una soluzione molto interessante e completa, alcune caratteristiche che meritano una menzione sono relative ad alcuni algoritmi crittografici supportati:

- supporto a certificati Ed25519 (istanza dell'algoritmo di firma digitale *EdDSA* basato su *Twisted Edward Curves*, particolarmente interessante per numerosi motivi tra cui le performance e la facilità di implementazione)
- supporto ad alcuni algoritmo a chiave pubblica post-quantistici (basati su problemi matematici per i quali i computer quantistici non portano alcun vantaggio nella loro soluzione) come NTRU (*lattice-based cryptography*)

strongSwan offre NAT Traversal basato su UDP.

## 2.7.2 IPsec e MoonCloud

Si analizzano ora le topologie possibili per MoonCloud. In IPsec, una volta che le policy sono state stabilite, non vi è davvero distinzione tra chi sia client e server, pertanto, non vi è davvero la necessità di distinguere tra *RS* ed *LS*. Si distingue solamente nelle topologie che si realizzano, ed in base a dove si posiziona il server cioè l'*initiator*. Sono quindi possibili sia topologie *RS* ed *LS*, un server nella rete target potrebbe sfruttare il NAT Traversal che strongSwan offre, occorrerebbe testarne attentamente le funzionalità. Tuttavia, è fondamentale che si configurino correttamente le policy relativamente al IP sorgente e destinazione.

## 2.7.3 Conclusioni

Si è consigliata una topologia *RS*, posto che il NAT Traversal consente la raggiungibilità dall'esterno. Vi sono diverse voci fondate dalle dichiarazioni di Snowden, secondo cui l'*NSA avrebbe volontariamente indebolito il protocollo...*<sup>3</sup> Altre voci invece riportano di come l'*NSA* sia in grado di rompere sessioni IPsec configurate con PSK ([4]).

Tuttavia il problema principale di IPsec è che si tratta di un protocollo ulteriore che i firewall devono consentire.

---

<sup>3</sup>La bibliografia è il corso di Sicurezza dei Sistemi e delle Reti.

## 2.8 Altre tecnologie

### 2.8.1 SSTP

*Secure Socket Tunneling Protocol* è il protocollo VPN di nuova generazione sviluppato da Microsoft, il cui client è nativamente integrato in Windows. Il server è tipicamente disponibile nelle versioni Server di Windows, tuttavia esistono dei cloni gratuiti, come SoftEther; vi sono anche client per Linux.

SSTP incapsula in TLS, quindi un protocollo sicuro, tuttavia non offre particolari vantaggi rispetto a SoftEther, inoltre il supporto per piattaforme non-Microsoft va verificato. Non vi sono particolari vantaggi rispetto alla soluzione HTTPS-based di SoftEther, pertanto non si era dedicata particolare attenzione a SSTP.

### 2.8.2 OpenConnect

*DTLS - Datagram Transport Layer Security* è uno standard RFC che specifica sostanzialmente l'uso di TLS su UDP, per fornire le stesse garanzie di sicurezza che TLS dà su TCP. Come tale, può essere utilizzato per costruire VPN.

Cisco Any Connect è una soluzione VPN basata su DTLS e HTTPS, e OpenConnect è una soluzione open source sia client sia server compatibile. OpenConnect fornisce una configurazione molto simile ad OpenVPN, ad esempio le direttive di configurazioni lato server sono davvero molto simili. Vi è una prima fase di autenticazione su HTTPS, il traffico poi è su DTLS, eventualmente si sposta su HTTPS se UDP non è supportato. Da questo punto di vista non avevo rilevato particolari vantaggi rispetto a SoftEther.

### 2.8.3 OpenSSH

*Secure SHell* è un protocollo usato da lungo tempo per creare sessioni di login remoto. La prima versione del protocollo SSH mirava semplicemente a rendere sicura una sessione telnet, mentre la versione 2 rende disponibile un protocollo di trasporto sicuro sul quale è possibile poi offrire diversi servizi multiplexati su quest'unico trasporto.

Sono offerte vari tipi di autenticazione, tra cui username e password, oppure basata su chiavi pubbliche (negoziata a priori).

Uno dei software più diffusi che implementano questo protocollo è OpenSSH.

Le ultime versioni di tale software consentono anche di creare diversi tipi di tunnel oltre al classico di login remoto, tra cui anche tunnel verso un host remoto al livello 2 o 3, realizzando quindi una VPN.

Per funzionare si usa TUN, quindi si creano delle interfacce di rete virtuali su client e sul server. E' una soluzione interessante, che non ho approfondito perché principalmente non offriva vantaggi rispetto a SoftEther od OpenVPN.

### 2.8.4 WireGuard

*WireGuard* è una moderna soluzione VPN per Linux, creata con i seguenti obiettivi:

- facile da configurare
- sicurezza mediante crittografia:
  - autenticazione con chiavi pubbliche Diffie-Hellman su curva ellittica X25519 e scambiate a priori
  - pacchetti scambiati usando *ChaCha20-Poly1305* con *Authenticated Encryption with Associated Data*
  - chiavi di sessione rinegoziate periodicamente
- elevate performance
- minimalità: key exchanging, aggiunta e rimozione delle rotte dal kernel non sono intenzionalmente gestite.

WireGuard è implementato come un modulo del kernel Linux e come tale riesce ad essere molto performante, i benchmark forniti dagli autori riportano un throughput vicino ad 1 Gbps (figura 2.11). Lavora a livello 3 ed i partecipanti sono dei peer, il protocollo di trasporto è UDP. Per funzionare correttamente con il NAT vi è la possibilità di configurare i peer per mandarsi periodicamente dei keep-alive.

Poiché è una proposta molto interessante, l'ho analizzata più in dettaglio rispetto alle altre proposte di questa sezione. Per configurare un peer, posto di avere la chiave pubblica dell'altro endpoint, occorre:

- aggiungere una nuova interfaccia virtuale e configurarla specificandone l'indirizzo IP, sarà l'indirizzo IP del tunnel VPN
- aggiungere le rotte verso le reti indirizzate dagli altri endpoint alla scheda di rete virtuale. Normalmente questa operazione è fatta automaticamente dal software VPN, non in questo caso.
- Configurazione della *cryptokey routing table*, ovvero specificare quali remote sono responsabili di quali reti a livello di VPN.

WireGuard ha un grande potenziale, soprattutto è estremamente performante e facile da configurare, infatti, le opzioni che possono essere specificate sono davvero poche. Un'interfaccia WireGuard può gestire più peer, ed occorre che le chiavi pubbliche di tutti i peer a cui ci si vuole connettere siano presenti su tutti i partecipanti. Si parla di *peer* e si distingue solo tra *initiator* e *responder*, considerano il primo client ed il secondo server, a causa della presenza di NAT avevo consigliato una topologia *Local Server*.

### 2.8.5 IPsec/L2TPv3 e EtherIP/IPsec

- L2TP versione 3 è l'ultimo aggiornamento del protocollo L2TP, principalmente implementato in devices Cisco. Analogamente ad L2TP, è possibile usarlo per creare VPN LAN-to-LAN al livello 2. Viene supportato dai principali produttori di apparati di rete, oltre che da Linux.

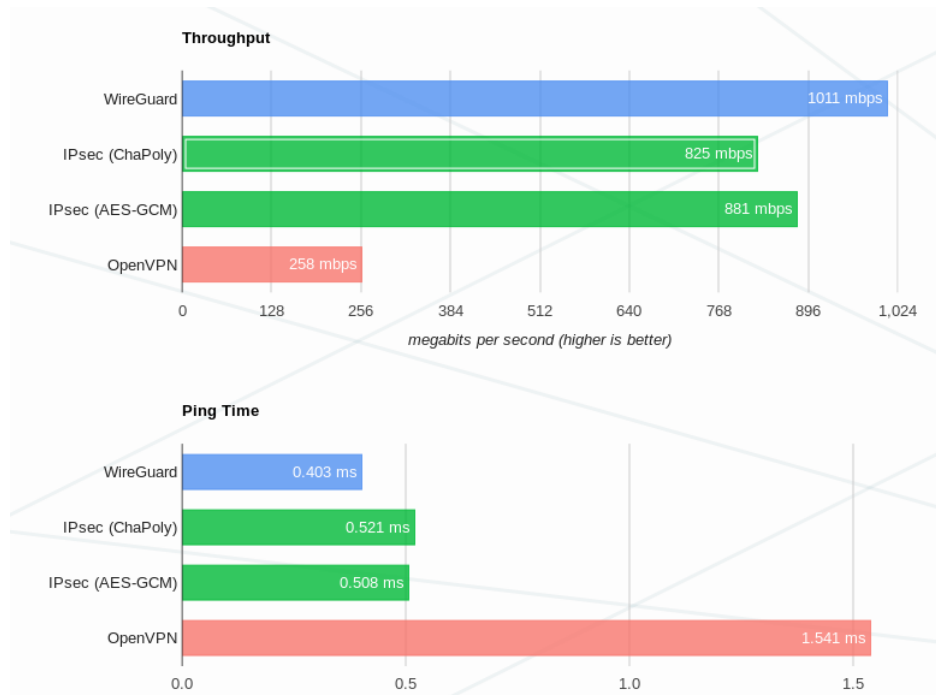


Figura 2.11: Benchmark tra IPsec, OpenVPN e WireGuard.

- EtherIP è un protocollo per l'incapsulamento di frame Ethernet in frame IP, il cui scopo è quello di connettere tra loro diverse LAN a livello 2.

Entrambi i due protocolli devono essere incapsulati in IPsec per ottenere una VPN che garantisca proprietà di sicurezza. EtherIP è supportato da Linux ed in SoftEther VPN Server.

Le due tecnologie sono in qualche modo simili, poiché realizzano connessioni a livello 2. Nel caso in cui MoonCloud abbia esigenze di lavorare al livello 2, queste soluzioni sono da prendere in considerazione, sebbene anche OpenVPN e SoftEther supportino VPN a questo layer.

### 2.8.6 PPTP

Il protocollo *PPTP - Point To Point Tunneling Protocol* è un protocollo *legacy*, sviluppato da Microsoft e non più sicuro, sebbene (purtroppo) ancora usato. Questo protocollo offre VPN, ma non è assolutamente una soluzione praticabile per MoonCloud.

## 2.9 Conclusioni

In conclusione del capitolo si fornisce un riassunto di pro e contro delle principali tecnologie. Nonostante WireGuard non sia stato analizzato in dettaglio, viene comunque inserito in questa lista grazie ai suoi vantaggi.

**OpenVPN :**

- ampio supporto e documentazione
- sicurezza basata su TLS e protocollo custom
- vari tipi di autenticazione
- possibile connessione anche al layer 2
- topologie consigliate:
  1. *MLS* con client in *NAT al contrario*
  2. *RS* con port forwarding abilitato sulla rete target
- Contro:
  - utilizzo di TLS su protocollo custom, quindi il traffico non riesce ad essere spacciato completamente per HTTPS
  - necessaria gestione dei certificati

**SoftEther :**

- multi-protocol e multi-platform
- incapsulamento in HTTPS
- possibile connessione anche al layer 2;
  - bypassa la maggior parte dei firewall
- NAT Traversal del server
- Dynamic DNS;
- VPN over ICMP e over DNS;
- topologia consigliate:
  1. *RS* senza bisogno di abilitare port forwarding sfruttando NAT Traversal
  2. *LS*
- Contro:
  - necessità di una secondo NIC sul server in modalità promiscua;
  - documentazione buona per Windows, meno per Linux

**IPsec :**

- IKEv2 dà ampia scelta di configurazioni
- sicurezza mediante ESP (cifatura payload pacchetto IP)
- NAT Traversal
- topologie consigliate:
  1. *LS*
  2. *RS*
- Contro:
  - setup non facile
  - necessaria gestione dei certificati

- non passa tutti i firewall poiché si tratta di un nuovo protocollo che deve essere abilitato
- voci di indebolimento da parte dell'NSA

**WireGuard :**

- molto più performante di ogni altra soluzione
- altamente sicuro
- keep-alive per funzionare con NAT
- Contro:
  - il sito stesso non lo definisce *production ready*;
  - necessità di scambiare chiavi a priori
  - traffico UDP potrebbe essere bloccato

La soluzione SoftEther con HTTPS è sembrata essere quasi perfetta, se il NAT Traversal funzionasse come dichiarato si sarebbe potuta creare una topologia RS. Per firewall molto stringenti sono sembrate molto interessanti le soluzioni di incapsulamento su ICMP e DNS, sebbene dichiarate come non completamente affidabili.

L'opzione IPsec con IKEv2 è altrettanto valida, il principale svantaggio è che i firewall più stringenti potrebbero bloccare il traffico *ESP*. In tal caso si può incapsulare *ESP* in *UDP*, ma comunque si richiede che il traffico UDP sia permesso (porta 4500).

Come prima soluzione si è scelta SoftEther, ma infine quella utilizzata è OpenVPN. Il prossimo capitolo affronta tutto nel dettaglio.

## Capitolo 3

# Configurazioni di OpenVPN

In questo capitolo si vedrà nel dettaglio come OpenVPN è stato configurato per funzionare in MoonCloud, si descriverà anche cosa sia e come funzioni l'“IP remapping”, il tassello fondamentale alla base della VPN di MoonCloud.

### 3.1 SoftEther

Dopo aver concluso lo studio sulle principali tecnologie VPN attualmente disponibili, è stata scelta, assieme al prof. Anisetti, SoftEther. La ragione di tale scelta sta principalmente nella funzionalità di NAT traversal e di VPN-over-HTTPS; la configurazione che si era pensata era di tipo *Remote Server*.

Il passo successivo è stato quello di testare tale soluzione in un ambiente virtuale, mi sono quindi iscritto ad Amazon AWS in modo che disponessi di un host in Internet su cui effettuare le mie prove. Ecco come si componeva la configurazione nel test iniziale:

- su Amazon EC2: 1 host che fungeva da client VPN
- sul mio PC ho creato, mediante VirtualBox, due reti locali separate (come se fossero due reti target di due client diversi), ciascuna composta da:
  - 1 router/firewall di confine che eseguisse il NAT verso l'esterno (utilizzando iptables)
  - 1 server VPN
  - due host “normali” con un server web in esecuzione

Tutti i PC coinvolti utilizzavano una distribuzione Linux, in particolare Ubuntu (in versione server su Amazon), mentre i firewall utilizzavano Alpine Linux. L'obiettivo di questo test era, innanzitutto riuscire a connettere client e server in VPN, e quindi poter visualizzare dal client VPN le pagine web offerte dagli host nelle due reti.

I firewall avevano una configurazione di tipo *stateful*, e l'unico traffico ammesso era sulle porte 80 e 443 TCP.

Questo test non ha avuto successo, infatti non sono riuscito nemmeno a raggiungere il punto 1, non c'è stato modo di far connettere client e server. Nel prossi-

mo paragrafo descriverò più nel dettaglio la configurazione di SoftEther che ho testato.

Sul server VPN ho installato il componente “*SoftEther VPN Server*”, sul client ho invece installato “*SoftEther VPN Bridge*”. Già solo realizzare questo primo passo non si è dimostrato facile, a causa della scarsa documentazione disponibile per Linux, ed anche a causa dell’*operosità* dell’installazione e della configurazione: infatti per gestire tali software è disponibile un programma linea di comando chiamato “*vpncmd*” con il quale l’interazione è abbastanza *tediosa*.

Una volta che l’installazione è terminata, mi sono scontrato con il primo vero problema: il tanto sbandierato NAT traversal offerto dal server funziona su UDP (scoprirlo non è stato affatto facile), pertanto la soluzione *Remote Server* è stata scartata. Tra i requisiti per la VPN vi era infatti la necessità di funzionare su TCP, e senza la possibilità per il server installato nella rete cliente di poter essere raggiunto dall’esterno (trovandosi ragionevolmente dietro un NAT) questa configurazione è stata abbandonata.

Il passo successivo è stato quindi quello di installare il *SoftEther VPN Server* sull’host in esecuzione nel cloud di Amazon ed il *Bridge* sulla VM che precedentemente ospitava il server. Ho quindi configurato propriamente i due software, ed ho quindi tentato di collegarli, tuttavia il collegamento VPN non si instaurava. Ho quindi esaminato i file di log, ed ho scoperto che i componenti riuscivano ad instaurare una connessione, ma vi era un misterioso `Error Code 33`. Ho cercato a lungo cosa significasse tale errore, ma dopo due giorni mi sono arreso.

Sia chiaro che non metto in dubbio la bontà di SoftEther, esso infatti è largamente usato, ad esempio anche nel progetto *VPN Gate* (<http://www.vpngate.net>); tuttavia SoftEther non ha funzionato nello scenario in questione.

## 3.2 OpenVPN

Dopo aver scartato SoftEther, ho quindi iniziato a testare OpenVPN, che era la seconda scelta dopo, appunto, SoftEther. Prima di passare a descrivere la configurazione utilizzata, è utile fare un breve riepilogo ed approfondimento su come OpenVPN funzioni. Già in questo capitolo si discute della sicurezza di OpenVPN; tuttavia si rimanda al capitolo 5 per un’analisi approfondita.

### 3.2.1 Introduzione

OpenVPN è una tecnologia VPN ampiamente diffusa ed accettata, capace di realizzare diverse topologie, incluse ovviamente l’accesso remoto ed LAN-to-LAN, può funzionare al livello 2 o al livello 3 dello stack ISO/OSI, e come protocollo di trasporto può utilizzare TCP oppure UDP. OpenVPN è quindi ampiamente configurabile, e per questo motivo è una soluzione molto flessibile, e, a mio parere, questa flessibilità è anche il suo punto di forza.

OpenVPN utilizza due canali tra i due peer, multiplexati su una singola connessione TCP/UDP. Esse sono:



- `Control Channel` una connessione cifrata con TLS utilizzata tra i partecipanti per scambiarsi informazione di *servizio*, come configurazioni inviate da server a client e negoziazione delle chiavi.
- `Data Channel` la connessione su cui sono effettivamente scambiati i dati della VPN, protetta con un protocollo simile a TLS.

Il `Control Channel` garantisce tutte le proprietà di sicurezza di TLS, tra cui ovviamente autenticazione, confidenzialità ed integrità dei dati scambiati. utilizzando un algoritmo di cifratura a chiave simmetrica combinato con un MAC.

Una volta che il normale TLS handshake viene portato a termine ed il `Control Channel` è in funzione, OpenVPN genera delle chiavi simmetriche da utilizzare per proteggere il `Data Channel`. Contrariamente al TLS handshake in cui si utilizza Diffie-Hellman come *base* per derivare le chiavi di sessioni, nel `Data Channel` le chiavi sono una semplice *composizione* di byte pseudo casuali generati dalla libreria TLS usata da OpenVPN. Queste chiavi vengono scambiate nel `Control Channel`, ed entrambi i partecipanti contribuiscono nel materiale pseudo casuale.

A questo punto si può dire che l'intero handshake sia stato completato, e le parti delle seguenti chiavi:

- $A_{TLS} \rightarrow B_{TLS}$
- $B_{TLS} \rightarrow A_{TLS}$
- $A_{OpenVPN} \rightarrow B_{OpenVPN}$
- $B_{OpenVPN} \rightarrow A_{OpenVPN}$

Ci sono casi in cui le chiavi non sono quattro in tutto ma bensì otto, ciò dipende dal fatto che si utilizzi un cifrario che *da solo* garantisca *Authenticated Encryption (with Associated Data)* (come AES-GCM o ChaCha20-Poly1305) oppure uno in cui occorra combinare anche un algoritmo di MAC, come AES-CBC-HMAC-SHA256. Entrambi i due canali garantiscono quindi *Authenticated Encryption*. Tutte le operazioni crittografiche sono effettuate mediante librerie esterne, attualmente OpenVPN supporta le seguenti tre:

- OpenSSL
- LibreSSL (fork di OpenSSL)
- mbedTLS

E' possibile utilizzare un'autenticazione delle parti partecipanti alla VPN mediante certificati X509 oppure mediante chiavi simmetriche precondivise. Nel caso di MoonCloud ho utilizzato la prima opzione poiché garantisce *Perfect Forward Secrecy*.

OpenVPN è disponibile in due versioni: una gratuita ed una a pagamento denominata "*Access Server*", la quale offre una interfaccia web per configurare il server. E' stato scelto di utilizzare la versione gratuita ("*community edition*"), e come tale per poter essere configurata si utilizza un file di configurazione. Tale

file viene letto all'avvio del servizio.

A partire dalla versione 2.0, OpenVPN è una VPN *client-server*, nel senso che vi è un host che svolge il ruolo di server ed accetta più connessioni provenienti dai client. Una volta che tale connessione è stata stabilita, client e server possono essere considerati dei *peer* poiché entrambi possono mandare pacchetti all'altro, senza necessità che un client richieda qualcosa ed il server vi risponda (come in un tradizionale protocollo client-server). Per questo, spesso si utilizzerà il termine "peer", quando non è necessario distinguere chi sia il client e chi il server.

OpenVPN funziona esattamente come descritto nel capitolo precedente, nella sezione dedicata alla anatomia di una VPN (a pagina ??), comunque, qua lo si descrive nuovamente. Server e client sono connessi mediante dei socket su un protocollo di trasporto, i pacchetti scambiati lungo tale socket sono cifrati. Su questo collegamento vi sono quei pacchetti provenienti dalla rete a cui uno di essi appartiene e destinati all'altra. Ciascun partecipante crea quindi una scheda di rete virtuale a cui invia i pacchetti destinati ad host della propria rete, e riceve pacchetti provenienti da essi e destinati alla rete del client.

Si esamina ora il flusso che compie un pacchetto generato da un certo host nella rete del server e destinato ad un host nella rete del client.

1. Il pacchetto viene ricevuto dalla scheda di rete fisica del server e passa al kernel del sistema operativo
2. il sistema operativo provvede ad inoltrare il pacchetto alla scheda di rete virtuale del VPN server
3. il processo VPN server riceve il pacchetto, esso è composto dall'header IP come protocollo di più basso livello
4. il processo verifica a quale client inoltrare il pacchetto sulla base delle rotte configurate
5. il pacchetto viene inviato cifrato al client designato lungo il socket. Nello specifico, *il vecchio pacchetto viene incapsulato in un nuovo pacchetto, ed il pacchetto incapsulato è cifrato, diventando il payload del pacchetto più esterno*<sup>1</sup>.
6. il pacchetto, dopo aver attraversato Internet, viene ricevuto dalla scheda di rete fisica del client
7. il sistema operativo lo inoltra quindi al socket del client
8. il client decifra il pacchetto e lo scrive sulla sua scheda di rete virtuale
9. il sistema operativo "riceve" il pacchetto e vi applica il routing, quindi lo inoltra alla scheda di rete fisica dell'host
10. il pacchetto viene infine ricevuto dal destinatario

Il pacchetto ricevuto al punto 1 è esattamente identico a quello del punto 10. Tutto ciò presuppone che siano configurate le seguenti rotte:

---

<sup>1</sup>Ciò che si è descritto, ovvero incapsulare un pacchetto di rete in un altro pacchetto facendolo diventare il payload di quest'ultimo, viene detto *tunneling*.

- nella rete del server:
  - *destinazione: rete-client, via: VPN server*; questa configurazione può essere fatta su ogni host della rete oppure sul default gateway
- nella rete del client analogamente:
  - *destinazione: rete-server, via: VPN client*

Infine, il VPN server deve essere configurato per sapere quale particolare è responsabile per una particolare rete.

### 3.2.2 Configurazione base

In questa sottosezione si propongono due esempi di una configurazione “tipica” e minimale per OpenVPN, per realizzare una topologia LAN-to-LAN. Si riporta il contenuto del file di configurazione, ciascuna entry sarà quindi discussa in dettaglio. Si supponga che la sottorete del server abbia il suo indirizzo: 192.169.100.0/24, mentre quella del client sia: 192.168.1.0/24. Il server è raggiungibile tramite l’IP pubblico 11.9.250.34 (esso potrebbe essere l’IP pubblico del firewall configurato in port forwarding, ma questo non è importante ai fini della configurazione di OpenVPN).

#### Configurazione del server

Ecco come si potrebbe presentare il file di configurazione del server.

```
mode server
tls-server

# il protocollo L4
proto tcp

# il tipo di scheda di rete virtuale
# ed il suo nome
dev-type tun
dev server

persist-tun
persist-key

# privileges downgrading
group nogroup
user nobody

# porta di ascolto
port 443

# VPN subnet
```

```
server 10.7.0.0 255.255.255.0

# topologia
topology subnet

# configurazioni TLS
remote-cert-eku "TLS Web Client Authentication"
tls-version-min 1.2
tls-cipher TLS-ECDHE-ECDSA-WITH-AES-256-GCM-SHA384
cipher AES-256-GCM

# certificato CA
ca /etc/openvpn/certs/ca.crt

# key/cert di questo server
key /etc/openvpn/server/server.key
cert /etc/openvpn/server/server.crt

# ECDH anziché DH
dh none

# nuove chiavi ogni venti minuti
reneg-sec 1200

# configurazioni log
log /var/log/openvpn/openvpn-server.log
verb 4
status /var/log/openvpn/openvpn-status.log

# configurazioni keepalive
keepalive 10 60

# informazioni di routing
push "route 192.168.100.0 255.255.255.0"
route 192.168.1.0 255.255.255

# spiegato poi
client-config-dir /etc/openvpn/server/ccd
```

Si supponga quindi che il CommonName nel certificato del client sia "client1", e che esista un file al percorso "/etc/openvpn/server/ccd/client1". Tale file avrà questo contenuto:

```
iroute 192.168.1.0 255.255.255.0
```

Ora si descrive il contenuto di questi due file, linea per linea:

**mode server** Indica che OpenVPN opera in modalità server.

**tls-server** Questa direttiva specifica quale è il "ruolo" nel TLS handshake.

**proto tcp** Indica quale è il protocollo di trasporto utilizzato, in questo caso è TCP.

**dev-type tun** Indica quale è il tipo di scheda di rete virtuale. Come è stato detto nel capitolo 2, “tun” è un tipo di scheda di rete virtuale creata al livello 3.

**dev server** La direttiva “dev” indica che nome dare alla scheda di rete virtuale, in questo caso la si chiama “server”.

**persist-tun** Analogamente alla direttiva spiegata sopra, questa indica di non chiudere e riaprire la scheda di rete virtuale tra i vari restart.

**persist-key** Indica di non rileggere la chiave privata quando una sessione VPN finisce, o quando si ricevono particolari segnali dal sistema operativo.

**group nogroup, user-nobody** Questa direttiva fa sì che una volta partito, OpenVPN non abbia più privilegi amministrativi.

**port 443** Specifica su quale porta il server si mette in ascolto per ricevere connessioni dai client.

**server 10.7.0.0 255.255.255.0** Questa direttiva indica che la sottorete assegnata al collegamento VPN è quella indicata.

**topology subnet** Indica di allocare un indirizzo IP alla scheda di rete virtuale, esso appartiene al pool di IP presente nella direttiva `server`. Il server avrà come IP il primo IP disponibile nella sottorete.

**remote-cert-eku "TLS Web Client Authentication"** Direttiva che specifica di che “tipo” deve essere il certificato dei client che si connettono, è un’opzione non obbligatoria ma che migliora la sicurezza.

**tls-version-min 1.2** Come si può immaginare, indica quale versione minima del protocollo TLS si deve usare. TLS 1.3 è già diventato standard, ma attualmente non è ancora supportato.

**tls-cipher TLS-ECDHE-ECDSA-WITH-AES-256-GCM-SHA384** Questa direttiva indica quali ciphersuite di TLS utilizzare nel Control Channel<sup>2</sup>.

**cipher AES-256-GCM** Analogamente alla precedente che regolava il cifrario da utilizzare sul Control Channel, la direttiva `cipher` specifica quale algoritmo, ed in quale modalità, per il Data Channel.

**ca /etc/openvpn/certs/ca.crt** Specifica il percorso a cui si trova il certificato della CA usato per creare i certificati dei client. E’ indispensabile ai fini della validazione degli stessi.

---

<sup>2</sup>E’ bene verificare quale versione di OpenVPN sia in uso e con quale libreria TLS sia stata compilata: attualmente solo LibreSSL supporta anche le ciphersuite TLS basate su ChaCha20-Poly1305, mentre nessuna libreria attualmente supporta tali algoritmi per DataChannel. OpenSSL effettivamente supporta l’algoritmo citato, ma solo nelle versioni più recenti, e OpenVPN non supporta, per ora, tali versioni.

**key /etc/openvpn/server/server.key, cert /etc/openvpn/server/server.crt** Indica il percorso a cui si trovano, rispettivamente, la chiave privata del server ed il suo certificato.

**dh none** Normalmente OpenVPN negozia le chiavi utilizzando il *normale* Diffie-Hellman (aritmetica modulare, utilizzando numero primo), e richiede che i parametri per tale *key agreement* siano generati a priori. Se si vuole utilizzare Diffie-Hellman su curva ellittica (ECHD) occorre specificare “none” come parametro a “dh”.

**reneg-sec 12000** Le chiavi del Data Channel sono di default rinegoziate ogni ora, volendo essere particolarmente conservativi, si può configurare la rinegoziazione ogni 20 minuti (o un qualsiasi altro valore), come fatto in questo caso. E’ sufficiente specificare questa configurazione su un partecipante alla VPN, l’host con la soglia più bassa scatenerà una rinegoziazione.

**log /var/log/openvpn/openvpn-server.log** Questa direttiva specifica il percorso del file di log.

**verb 4** Specifica il livello di verbosità del log, un livello di 4 o 5 è considerato adeguato in produzione, livelli più alti (disponibile fino al 9) sono utilizzati per il debugging.

**status /var/log/openvpn/openvpn-status.log** Questa direttiva indica quale è il file di *status* in cui OpenVPN scrive informazioni relative alle connessioni con i client.

**keepalive 10 60** La direttiva “keepalive” indica ogni quanto OpenVPN deve mandare dei *keepalive* ai client a cui è connesso. Il primo parametro indica ogni quanto mandare dei *keepalive* (in secondi), mentre il secondo indica dopo quanti secondi considerare la connessione caduta se non riceve niente dal client a cui è connesso. E’ sufficiente specificare questa opzione nel server, esso provvederà ad trasmetterla al client.

**push "route 192.168.100.0 255.255.255.0"** Questa direttiva viene usata lato server per inviare delle configurazioni ai client. In questo caso si indica ai client di aggiungere nella routing table del sistema operativo una rotta che apparirà così: *rete 192.168.100.0/24 via IP-virtual-NIC*. Essendo *192.168.100.0/24* la rete in cui si trova il server, l’effetto di questa rotta è che ogni volta che il sistema operativo deve decidere dove inoltrare un pacchetto destinato alla rete del server, lo invii alla scheda di rete virtuale del client. In questo modo il client VPN lo riceve da essa e lo invia al server lungo il socket.

**route 192.168.1.0 255.255.255** Indica di aggiungere una rotta così composta nella routing table del sistema operativo: *rete 192.168.1.0/24 via IP-virtual-NIC*. Poiché *192.168.1.0/24* è la rete del client, l’effetto dell’aggiunta di questa rotta è l’analogo di quello visto al punto precedente.

**client-config-dir /etc/openvpn/server/ccd** Specifica in quale directory si trovano i file contenenti configurazioni specifiche per ciascun client.

Tale directory conterrà una serie di file i quali dovranno chiamarsi semplicemente come il `CommonName` dei certificati dei client. La principale e più importante direttiva nei suddetti file è descritta nel punto successivo.

**`iroute 192.168.1.0 255.255.255.0`** E' molto importante comprendere il significato di questa direttiva e capire che essa si trova dentro un file contenente configurazioni specifiche per un certo client. "`iroute`" seguito da un indirizzo di rete, indica che *il client a cui il file di configurazione si riferisce è responsabile per la rete 192.168.1.0/24*. Usando questa direttiva, OpenVPN costruisce una sorta di routing table interna, e da essa decide a quale client inviare un certo pacchetto. Bisogna infine tener presente che occorre anche la direttiva `route . . .`, poiché essa aggiunge delle rotte alla routing table del sistema operativo, mentre `iroute` solo alla routing table di OpenVPN.

## Hook

Oltre alle configurazioni *tipiche* appena mostrate, ve ne sono molte altre. In effetti, uno dei punti di forza di OpenVPN è proprio l'elevatissima configurabilità. In particolare, OpenVPN consente di definire una serie di *hook*, cioè azioni da eseguire (lanciare programmi esterni) quando alcuni eventi si verificano. Esistono degli hook ben precisi, alcuni dei quali sono effettivamente utilizzati nella configurazione di MoonCloud. Per ciascun hook si specifica nel file di configurazione quale azione compiere, cioè quale programma deve essere eseguito; OpenVPN setterà per tali processi un certo numero di variabili d'ambiente.

I programmi eseguiti devono ovviamente terminare con un valore di uscita secondo la classica semantica per cui 0 indica *successo*, un qualsiasi altro valore *fallimento*. Sulla base di esso OpenVPN decide se proseguire nella connessione con un remote oppure terminarla. Alcuni hook sono disponibili solo in modalità server, in tal caso l'idea sarebbe quella di decidere se consentire ad un client di portare avanti la connessione al server oppure no, basandosi sui valori delle variabili d'ambiente. Ad esempio, un programma esterno potrebbe accedere al valore del seriale del certificato del client, verificare se è presente in un database, ritornare 0 in caso positivo, od un altro valore in caso contrario.

Affinché il server possa eseguire programmi esterni è necessario specificare uno `script-security-level` nel file di configurazione principale. I valori possibili:

- 0: non è possibile chiamare alcun programma esterno
- 1: possibile chiamare solo software del sistema operativo (es: `ifconfig`)
- 2: consentito chiamare anche script definiti dall'utente
- 3: tra le variabili d'ambiente vi possono essere anche password

Infine, la lista degli hook disponibili in ordine di esecuzione, si noti che ciascun hook prevede delle diverse variabili d'ambiente.

- `up`, eseguito dopo un socket bind

- `tls-verify` eseguito durante il TLS handshake quando l'identità del client non è stata ancora verificata
- `ipchange`: quando il client è stato autenticato od il suo IP remoto cambia
- `client-connect`, eseguito immediatamente quando il client è stato autenticato, disponibile solo per il server
- `route-up`, eseguito anch'esso immediatamente quando la connessione è stata autenticata
- `route-pre-down`: eseguita immediatamente prima che una rotta aggiunta venga rimossa in seguito ad una terminazione della connessione
- `client-disconnect`, disponibile solo per il server, eseguita quando un client si disconnette
- `down`: hook che si verifica quando il socket o l'interfaccia virtuale vengono chiusi
- `learn-address`: disponibile solo per il server, si verifica quando una nuova rotta è aggiunta alla routing table interna di OpenVPN
- `auth-user-pass-verify`: eseguita sul server quando un client si connette e non è ancora stato autenticato

### Configurazione del client

Si passa quindi a descrivere la configurazione dell'ipotetico client OpenVPN in riferimento agli esempi precedenti.

```
client
```

```
proto tcp
```

```
# IP porta del server
remote 11.9.250.34 443
```

```
dev-type tun
dev client
```

```
persist-tun
persist-key
```

```
# privileges downgrading
group nogroup
user nobody
```

```
# tls config
remote-cert-eku "TLS Web Server Authentication"
```



```

tls-version-min 1.2
tls-cipher TLS-ECDHE-ECDSA-WITH-AES-256-GCM-SHA384
cipher AES-256-GCM

ca /etc/openvpn/certs/ca.crt

key /etc/openvpn/client1/client1.key
crt /etc/openvpn/client1/client1.crt

# configurazione log
log /var/log/openvpn/openvpn-client1.log
verb 4
status /var/log/openvpn/openvpn-client1.log

```

Ciò che si può notare è che il file sia abbastanza simile a quello del server, sebbene manchino le direttive di routing. Ovviamente, “client” è l’opposto di “mode server” e “tls-server”. Infatti, “client” include:

- pull direttiva usata per accettare configurazione inviate dal server mediante “push”
- tls-client che è l’opposto di “tls-server”.

La direttiva “push “route ...” non è presente perché può esserci solo nel file di configurazione del server, mentre la direttiva “route <rete-altro-peer>” non c’è ma “è come se ci fosse”, poiché tale rotta viene aggiunta alla routing table del sistema operativo grazie al fatto che vi è la direttiva “push” lato server. Una volta che il collegamento tra i due è stato stabilito vi sarà una entry nella routing table del kernel così fatta: *rete 192.168.100.0/24 via IP-virtual-NIC*. Non possono nemmeno esserci direttive “iroute”, perché il client si connette ad un solo altro endpoint che è il server.

Si può notare la presenza di una direttiva nuova: “remote remote 11.9.250.34 443”, essa indica l’indirizzo IP pubblico mediante il quale è possibile contattare il server, e la porta su cui è in ascolto.

Infine, non è presente l’opzione di “keepalive” perché, come detto in precedenza, è sufficiente specificarla solo sul server.

Infine, affinché si possa realizzare un collegamento LAN-to-LAN, è necessario che nelle due (in questo caso) reti coinvolte, sia stato configurato correttamente il routing. In particolare si richiede che, per ciascuna rete, sia configurata una rotta che dica come raggiungere l’altra rete remota.

Nell’esempio in questione, è necessario che tutti gli host della rete del server, cioè 192.168.100.0/24 sappia che per inviare pacchetti ad host di 192.168.1.0/24 (rete del client), occorre inoltrare tali pacchetti al VPN server. Allo stesso modo, nella rete 192.168.1.0/24 vi deve essere configurato che per raggiungere 192.168.100.0/24 occorre inoltrare i pacchetti al client.

Queste due configurazioni possono essere aggiunte su ciascun host delle due reti oppure, più ragionevolmente, esse saranno solo applicate sui due default gateway. È importante capire questo passaggio, poiché è alla base delle soluzioni descritte nella prossima sezione. Infatti, *per poter configurare una rotta su un router*

*occorre aver accesso al router*, ma ragionevolmente si vuole evitare di dover configurare il router del cliente ogni volta che si vuole effettuare un'analisi nella sua rete...

### 3.3 Problematiche di configurazione

Nella precedente sezione si è descritto OpenVPN e si è mostrato una configurazione tipica. A questo punto, prima di dettagliare come OpenVPN è stata configurato per MoonCloud, è necessario soffermarsi su quali siano le problematiche che i particolari requisiti di VPN per MoonCloud introducono. Essi sono già stati accennati nel capitolo 2, tuttavia ora li si affrontano nel dettaglio. Naturalmente, per ciascun problema si analizza anche la relativa soluzione adottata. Durante questa sezione si potrà notare l'elevata versatilità di OpenVPN, grazie alle numerosissime opzioni di configurazione possibili.

Prima di procedere, si tenga presente che la configurazione *definitiva* è di tipo *Local Server - LS*, con i server OpenVPN installati in MoonCloud, e nella target vi sono i client. Maggiori dettagli naturalmente verranno discussi di seguito.

#### 3.3.1 Impossibilità di aggiungere configurazioni alla rete target

##### Il problema

A questo problema era già stata pensata una soluzione nella fase iniziale di valutazione delle diverse VPN. Il quesito è il seguente: *per realizzare una topologia LAN-to-LAN è necessario configurare delle rotte sui router di confine, ma non è pensabile di operare sul router della rete target. Come fare?* Per cercare di comprendere meglio il problema, si analizza il flusso seguito da un ipotetico pacchetto inviato da un probe dalla rete MoonCloud alla rete target, posto che il collegamento sia stato instaurato con successo. Naturalmente, non vi sono problemi nel configurare rotte lato MoonCloud. Si supponga quindi di trovarsi nel seguente scenario:

- rete target: 192.168.100.0/24
- rete MoonCloud: 192.168.200.0/24
- indirizzo IP interno VPN server: 192.168.200.20
- indirizzo IP pubblico VPN server: 100.4.78.5
- indirizzo IP del *Docker host*: 192.168.200.2
- indirizzo IP VPN client: 192.168.100.50
- subnet VPN: 10.7.0.0/24
- indirizzo IP scheda di rete virtuale VPN server: 10.7.0.1
- indirizzo IP scheda di rete virtuale VPN client: 10.7.0.2
- CommonName del VPN client: client100

Sul *Docker host* di MoonCloud è configurata la seguente rotta: `192.168.100.0/24 via 192.168.200.20`, che specifica che per contattare la rete target, è necessario passare per il VPN server. Oltre ad essa, naturalmente, il server ed il client VPN hanno configurato le rotte come si è visto nella sezione precedente (per cui si dice “rete dietro l’altro endpoint via IP-virtual-NIC”).

Si supponga quindi che un *probe* generi un pacchetto di qualsiasi tipo destinato ad un host della rete target, tale host è, ad esempio, `192.168.100.254`. Il flusso seguito dal pacchetto è quello mostrato di seguito.

1. Il *probe* genera il pacchetto, la sorgente di tale pacchetto è `192.168.200.2`, la destinazione `192.168.100.254`<sup>3</sup>
2. il sistema operativo del *Docker host* consulta la propria routing table ed instrada il pacchetto verso `192.168.200.20`
3. il sistema operativo del VPN server riceve il pacchetto, consulta la routing table e trova l’entry `192.168.100.0/24 via 10.7.0.1`, quindi inoltra alla scheda di rete virtuale di OpenVPN
4. OpenVPN riceve il pacchetto, consulta la propria routing table interna e trova l’entry che specifica che il client “`client100`” è responsabile per la rete `192.168.100.0/24`, provvede quindi ad inviarlo a tale client
5. il VPN client riceve il pacchetto dal socket, e lo scrive sulla scheda di rete virtuale
6. il pacchetto destinato a `192.168.100.254` viene ricevuto dal sistema operativo, dopo aver compiuto le dovute operazioni di routing viene quindi inviato sulla scheda di rete fisica dell’host
7. `192.168.100.254` riceve il pacchetto, lo elabora, produce una risposta, la destinazione di tale risposta è `192.168.200.2`
8. il sistema operativo di `192.168.100.254` invia la risposta al proprio default gateway come avviene in tutti i casi in cui un host non abbia una rotta ben determinata per un pacchetto
9. il default gateway della rete `192.168.100.0/24` riceve il pacchetto destinato a `192.168.200.2`, poiché è un indirizzo IP privato ed esso lo instraderebbe su Internet, se è configurato correttamente lo dropa, altrimenti lo invia al suo default gateway ed il dropping del pacchetto viene solo posticipato.

In conclusione quindi, il problema sta nel fatto che nella rete target manca una rotta che dica che per raggiungere `192.168.200.0/24` occorre contattare il VPN client. Di nuovo, non è pensabile di dover configurare ogni rete target in questa maniera, poiché l’utilizzo di MoonCloud deve essere il più possibile *leggero*; la necessità di dover metter mano al router potrebbe infatti scoraggiare un potenziale cliente dall’usare MoonCloud.

---

<sup>3</sup>Volendo essere precisi, al momento della generazione del pacchetto in un container Docker, la sua sorgente è l’indirizzo IP di tale container, che poi viene modificato da Docker nell’indirizzo IP dell’host in cui il container è esecuzione, ai fini dell’esempio questo non è importante.

### 3.3.2 La soluzione

La soluzione a questo è semplice, ovvero far sì che tutti i pacchetti inviati dal device VPN client provenienti dalla VPN e destinati alla rete target, abbiano come indirizzo IP sorgente l'indirizzo IP del device stesso. In questo modo, quando si produce la risposta, la destinazione di tale pacchetto è un indirizzo IP appartenente alla stessa rete in cui si è. Il sistema operativo quindi, mediante il protocollo ARP, invia il pacchetto al device client, il quale provvede quindi a modificare l'IP destinazione nell'IP di destinazione *vero*, ovvero quello del Docker host. Ciò che avviene quindi è che il device client fa del NAT, come se la rete target fosse la rete *esterna* verso la quale il NAT è tipicamente eseguito. Questa configurazione di NAT viene qui chiamata "*NAT al contrario*", ed è eseguita con `iptables`. In questo caso, un singolo comando ad `iptables` risolve tutto questo problema:

```
iptables -t nat -A POSTROUTING -d 192.168.100.0/24 -s
↳ 192.168.200.0 -j MASQUERADE
```

Utilizzare il NAT introduce però appartenente un'altra problematica, ovvero il fatto che solo risposte a richieste provenienti da `192.168.200.0/24` possono "passare" il NAT: in questo caso però non vi è alcun problema, proprio perché i pacchetti che vengono inviati da `192.168.100.0` sono risposte a `192.168.200.0/24`. Si analizza ora nuovamente il flusso indicato sopra alla luce del NAT.

1. Il *probe* genera il pacchetto, la sorgente di tale pacchetto è `192.168.200.2`, la destinazione `192.168.100.254`.
2. il sistema operativo del *Docker host* consulta la propria routing table ed instrada il pacchetto verso `192.168.200.20`
3. il sistema operativo del VPN server riceve il pacchetto, consulta la routing table e trova l'entry `192.168.100.0/24` via `10.7.0.1`, quindi inoltra alla scheda di rete virtuale di OpenVPN
4. OpenVPN riceve il pacchetto, consulta la propria routing table interna e trova l'entry che specifica che il client "`client100`" è responsabile per la rete `192.168.100.0/24`, provvede quindi ad inviarlo a tale client
5. il VPN client riceve il pacchetto dal socket, e lo scrive sulla scheda di rete virtuale
6. il pacchetto destinato a `192.168.100.254` viene ricevuto dal sistema operativo, dopo aver compiuto le dovute operazioni di routing decide di inviarlo sulla scheda di rete fisica dell'host
7. il sistema operativo esegue POSTROUTING modificando l'indirizzo IP sorgente in `192.168.100.50`
8. `192.168.100.254` riceve il pacchetto, lo elabora, produce una risposta, la destinazione di tale risposta è `192.168.100.50`
9. il sistema operativo di `192.168.100.254` invia la risposta a `192.168.100.50`

10. il sistema operativo di 192.168.100.50 riceve il pacchetto ed esegue l'inverso del NAT, l'indirizzo IP destinazione appare ora essere 192.168.200.2. Poiché il client ha una rotta che dice: *rete 192.168.200.2 via IP-virtual-NIC* tale pacchetto viene inoltrato alla scheda di rete virtuale,
11. il pacchetto viene ricevuto da OpenVPN ed inviato al server
12. il server OpenVPN riceve il pacchetto e lo scrive sulla scheda di rete virtuale
13. il sistema operativo del server riceve il pacchetto e lo inoltra quindi a 192.168.200.2.

### 3.3.3 Dinamicità della creazione dei client

### 3.3.4 Il problema

Si è visto precedentemente che è necessario introdurre nel file di configurazione del server due direttive fondamentali, per ogni client, esse sono:

- `route <rete-client>`
- `iroute <rete-client>` (nel file di configurazione specifico)

Il file di configurazione viene letto da OpenVPN all'avvio... Il problema ora è il seguente: *come poter aggiungere nuovi client dinamicamente al server senza bisogno di riavviarlo?*

### 3.3.5 La soluzione

La soluzione di questo problema sfrutta gli hook `client-connect` e `client-disconnect`. L'idea è quella di eseguire direttamente un comando per l'aggiunta di una rotta al kernel del sistema operativo quando un client si connette, e di rimuoverla quando si disconnette, questo poiché l'effetto dell'opzione "`route`" è quello, appunto, di aggiungere una rotta alla routing table del sistema operativo. In sostanza, si "bypassa" la direttiva "`route`" e si esegue direttamente il comando di aggiunta della rotta.

Per quanto riguarda la direttiva "`iroute`" invece non ci sono problemi, una volta che "`client-config-dir`" è stata configurata correttamente, è possibile aggiungere in tale cartella il file specifico per il nuovo client con la direttiva "`iroute`".

Il file di configurazione del server quindi non presenta più le direttive di tipo "`route`", vi sono però le seguenti aggiunte:

```
client-up /etc/openvpn/server/cs/client-up.sh
client-down /etc/openvpn/server/cs/client-down.sh
```

Quando OpenVPN esegue i due script, passa ad essi alcune variabili, in particolare quella di interesse è `common_name`, che indica, ovviamente, il nome presente nel certificato fornito dal client. Si supponga ora che vi sia un secondo client, chiamato "`client50`" responsabile per la rete 192.168.50.0/24; OpenVPN rilegge i due script ad ogni connessione/disconnessione, pertanto è possibile aggiornarli mentre esso è in esecuzione. Ecco come si presentano i due file:

```

#!/bin/bash

# /etc/openvpn/server/cs/client-up.sh

if [ "$common_name" == "client100" ]; then
    ip route add 192.168.100.0/24 via 10.7.0.1
fi
if [ "$common_name" == "client50" ]; then
    ip route add 192.168.50.0/24 via 10.7.0.1
fi

#!/bin/bash

# /etc/openvpn/server/cs/client-down.sh

if [ "$common_name" == "client100" ]; then
    ip route del 192.168.100.0/24 via 10.7.0.1
fi
if [ "$common_name" == "client50" ]; then
    ip route del 192.168.50.0/24 via 10.7.0.1
fi

```

Nel momento in cui un client si connette, si aggiunge una nuova rotta nel kernel del sistema operativo, e quando si disconnette la si rimuove.

La ragione per cui si usa “if-fi, if-fi” e non “if-elif-fi” è che questo file viene generato automaticamente, ed utilizzare “if-fi, if-fi” è più semplice: quando si aggiunge un nuovo client è sufficiente aggiungere un nuovo “if-fi” al file, mentre se si fosse seguita la seconda strada sarebbe stato necessario modificare l’ultimo “fi” in un “elif”.

Quindi, sfruttando le direttive “client-connect” e “client-disconnect” ho trovato un modo per bypassare la staticità del file di configurazione del server.

### 3.3.6 Dinamicità delle rotte server-side

#### Il problema

Questo problema è simile al precedente. La direttiva “push route <server-net>” viene utilizzata nel server per indicare al client quale sia la/e rete/i che si trovano “dietro” il server OpenVPN e che si vuole far sì che il client possa raggiungerli. Questo presuppone di sapere in partenza quali siano queste reti, ma in un ambiente dinamico quale MoonCloud ciò non è possibile. Può capitare che i *Docker host* che vengono aggiunti on-demand appartengano alla stessa rete del server, come può anche capitare che appartengano ad un’altra rete di cui il server non ha conoscenza, ma comunque vi è mutua raggiungibilità tra tale server e tale rete. Il quesito da risolvere in questo caso è il seguente: *come gestire la*

*dinamicità delle rotte lato server senza bisogno di riavviare il server?* Il punto di partenza di questo problema è la necessità del client di conoscere le reti “dietro” al server OpenVPN, che fin’ora era gestito mediante le “push route <server-net>”, ma ora si è visto che non sono più utilizzabili. Prima di vedere la soluzione, si esamina il flusso di un pacchetto nel caso in cui tale direttiva non venga più inclusa.

1. Il *probe* genera il pacchetto, la sorgente di tale pacchetto è 192.168.200.2, la destinazione 192.168.100.254.
2. il sistema operativo del *Docker host* consulta la propria routing table ed instrada il pacchetto verso 192.168.200.20
3. il sistema operativo del VPN server riceve il pacchetto, consulta la routing table e trova l’entry 192.168.100.0/24 via 10.7.0.1, quindi inoltra alla scheda di rete virtuale di OpenVPN
4. OpenVPN riceve il pacchetto, consulta la propria routing table interna e trova l’entry che specifica che il client “client100” è responsabile per la rete 192.168.100.0/24, provvede quindi ad inviarlo a tale client
5. il VPN client riceve il pacchetto dal socket, e lo scrive sulla scheda di rete virtuale
6. il pacchetto destinato a 192.168.100.254 viene ricevuto dal sistema operativo del VPN client, dopo aver compiuto le dovute operazioni di routing decide di inviarlo sulla scheda di rete fisica dell’host
7. il sistema operativo esegue POSTROUTING modificando l’indirizzo IP sorgente in 192.168.100.50
8. 192.168.100.254 riceve il pacchetto, lo elabora, produce una risposta, la destinazione di tale risposta è 192.168.100.50
9. il sistema operativo di 192.168.100.254 invia la risposta a 192.168.100.50
10. il sistema operativo di 192.168.100.50 riceve il pacchetto ed esegue l’inverso del NAT, l’indirizzo IP destinazione appare ora essere 192.168.200.2
11. poiché non vi sono rotte configurate per 192.168.200.0/24, il sistema operativo decide di inoltrare il pacchetto al default gateway
12. il default gateway della rete 192.168.100.0/24 riceve il pacchetto destinato a 192.168.200.2, poiché è un indirizzo IP privato ed esso lo instraderebbe su Internet, se è configurato correttamente lo dropa, altrimenti lo invia al suo default gateway ed il dropping del pacchetto viene solo posticipato.

### La soluzione

La soluzione è quella di far sì che tutti i pacchetti che provengono dalle reti interne dietro il server OpenVPN verso le reti target arrivino al client *con l'indirizzo IP sorgente uguale all'indirizzo IP della scheda di rete virtuale OpenVPN del server*. Quale vantaggio dà questo? L'indirizzo IP della scheda di rete virtuale del client si trova nella stessa subnet di quello del server, quindi il sistema operativo aggiunge automaticamente una rotta alla propria routing table verso quella scheda di rete, nell'esempio in questione, quando tale scheda viene creata, la rotta aggiunta è la seguente: 10.7.0.0/24 via 10.7.0.2.

Lato server si usa NAT con iptables, ed i comandi sono ancora una volta presenti nei file specificati nelle direttive "client-disconnect" e "client-disconnect"; nel seguente modo:

```
# /etc/openvpn/server/cs/client-up.sh

#!/bin/bash

if [ "$common_name" == "client100" ]; then
    ip route add 192.168.100.0/24 via 10.7.0.1
    iptables -t nat -A POSTROUTING -d 192.168.100.0/24 -j
    ↪ MASQUERADE
fi
if [ "$common_name" == "client50" ]; then
    ip route add 192.168.50.0/24 via 10.7.0.1
    iptables -t nat -A POSTROUTING -d 192.168.50.0/24 -j
    ↪ MASQUERADE
fi

# /etc/openvpn/server/cs/client-down.sh

#!/bin/bash

if [ "$common_name" == "client100" ]; then
    ip route del 192.168.100.0/24 via 10.7.0.1
    iptables -t nat -D POSTROUTING -d 192.168.100.0/24 -j
    ↪ MASQUERADE
fi
if [ "$common_name" == "client50" ]; then
    ip route del 192.168.50.0/24 via 10.7.0.1
    iptables -t nat -D POSTROUTING -d 192.168.50.0/24 -j
    ↪ MASQUERADE
fi
```

Si veda quindi il flusso seguito da un pacchetto dall'invio di un pacchetto da parte di un probe.



1. Il *probe* genera il pacchetto, la sorgente di tale pacchetto è 192.168.200.2, la destinazione 192.168.100.254.
2. il sistema operativo del *Docker host* consulta la propria routing table ed instrada il pacchetto verso 192.168.200.20
3. il sistema operativo del VPN server riceve il pacchetto, consulta la routing table e trova l'entry 192.168.100.0/24 via 10.7.0.1
4. il sistema operativo esegue POSTROUTING e modifica l'indirizzo IP sorgente in 10.7.0.1, quindi inoltra alla scheda di rete virtuale di OpenVPN
5. OpenVPN riceve il pacchetto, consulta la propria routing table interna e trova l'entry che specifica che il client "client100" è responsabile per la rete 192.168.100.0/24, provvede quindi ad inviarlo a tale client
6. il VPN client riceve il pacchetto dal socket, e lo scrive sulla scheda di rete virtuale
7. il pacchetto destinato a 192.168.100.254 viene ricevuto dal sistema operativo, dopo aver compiuto le dovute operazioni di routing decide di inviarlo sulla scheda di rete fisica dell'host
8. il sistema operativo esegue POSTROUTING modificando l'indirizzo IP sorgente in 192.168.100.50
9. 192.168.100.254 riceve il pacchetto, lo elabora, produce una risposta, la destinazione di tale risposta è 192.168.100.50
10. il sistema operativo di 192.168.100.254 invia la risposta a 192.168.100.50
11. il sistema operativo di 192.168.100.50 riceve il pacchetto ed esegue l'inverso del NAT, l'indirizzo IP destinazione appare ora essere 10.7.0.1
12. il sistema operativo inoltra il pacchetto alla scheda di rete virtuale, (poiché vi è una rotta aggiunta nel kernel che dice di instradare per 10.7.0.0/24 verso 10.7.0.2) viene quindi ricevuto da OpenVPN ed inviato al server
13. il server OpenVPN riceve il pacchetto e lo scrive sulla scheda di rete virtuale
14. il sistema operativo del server riceve il pacchetto ed applica l'inverso del NAT, ora l'indirizzo IP destinazione è 192.168.200.2
15. il pacchetto viene quindi inviato a 192.168.200.2.

Per poterci riferire a questa soluzione, essa viene detta "*NAT lato server*".

## 3.4 IP mapping

L'ultimo problema che viene descritto è anche l'ultimo problema di cui io ed il prof. Anisetti ci siamo accorti, in ordine cronologico; ed è stato anche il problema più difficile da risolvere, e che, potenzialmente, avrebbe potuto far fallire tutto il lavoro precedentemente fatto, ma procediamo con ordine.

### 3.4.1 Il problema

Si è detto in precedenza che si è scelta infine una configurazione *Multi Local Server*, con i server OpenVPN installati in MoonCloud e raggiungibili mediante IP pubblici da i client presenti nella rete target. Per ragioni di costi, si vuole limitare il numero di VM in MoonCloud con OpenVPN in esecuzione, e quindi poter dedicare un singolo server a servire più reti target contemporaneamente. OpenVPN consente di isolare i singoli client connessi ad un server, infatti di default tali client possono comunicare solo con il server, e non con altri client (a meno che si utilizzi la direttiva `"client-to-client"`, e, come si è visto, tale direttiva non viene utilizzata).

Per poter quindi realizzare in maniera corretta una topologia del genere è *fondamentale che ogni rete target abbia un spazio di indirizzamento diverso*... Tuttavia qui sorge il problema: ciascuna LAN target è ragionevolmente dotata di uno o più pool di indirizzi privati, i quali sono limitati. L’RFC 1918 [5] stabilisce che i seguenti spazi di indirizzi siano riservati per l’utilizzo esclusivo in una rete privata e che non siano instradabili su Internet:

- classe A (subnet mask a 8 byte) 10.0.0.0 – 10.255.255.255
- classe B (subnet mask a 16 byte) 172.16.0.0 – 172.31.255.255
- classe C (subnet mask a 24 byte) 192.168.0.0 – 192.168.255.255

Spesso accade poi che tali indirizzi siano utilizzati con una subnet mask non *canonica*, ad esempio si può utilizzare 10.1.1.0/24, rimane comunque il fatto che i detti prefissi sono tipicamente quelli che si trovano in una qualsiasi LAN.

Nell’estate del 2014 ho lavorato per tre mesi in una azienda che si occupava di gestire la parte IT di altre aziende, e, sebbene non mi vantì di avere una grande esperienza in merito, ho potuto constatare come nei fatti solo *alcuni* degli indirizzi sopra citati siano utilizzati, ad esempio ho visto usati spesso indirizzi nel range 10.1.1.0/24 – 10.10.10.0/24 e 192.168.1.0/24 – 192.168.100.0/24. E’ quindi chiaro come la probabilità che due reti target di due clienti diversi abbiano lo stesso spazio di indirizzamento sia piuttosto alta. Diventa quindi un grosso problema poter collegare più reti target ad uno stesso VPN server, proprio perché avranno uno stesso indirizzo IP di rete.

Una prima soluzione potrebbe essere quella di destinare ad ogni server un certo numero di reti target predefinite, ad esempio al server OpenVPN10-20 si assegnano le reti nello spazio 10.0.0.0/24 – 10.20.20.0/24, e così via fino ad esaurire lo spazio di indirizzi privati. Ogni volta che si collega una nuova rete target, essa viene assegnata al server VPN competente sulla base del suo indirizzo di rete... Chiaramente questa soluzione non è possibile, si possono individuare almeno tre problematiche:

- cosa fare nel caso due reti target diverse abbiano lo stesso indirizzo IP?
- Cosa fare nel caso in cui cliente sia dotato di più reti, che per competenza spetterebbero a diversi server OpenVPN? Non è possibile collegare uno stesso client a due server contemporaneamente.

- Molti server infine resterebbero poco utilizzati, perché sebbene, ad esempio, siano responsabili per venti target, ne hanno assegnate solo un paio. Viceversa, altri sarebbero utilizzati fino al proprio limite.

Un'altra soluzione potrebbe essere quella di allocare un server OpenVPN *per ogni* cliente, sia che esso abbia una oppure  $n$  reti che debbano essere analizzate da MoonCloud. Lato MoonCloud si creano quindi tante reti isolate tra loro costituite da un server OpenVPN e da un *Docker host*, ciascuna competente per un singolo cliente. Questa configurazione risolve effettivamente il problema di conflitto tra reti target diverse, poiché si assume che un singolo cliente, nel caso in cui sia composto da più di una rete, abbia configurato le proprie reti con indirizzi di rete *diversi*.

Perché non adottare quindi questa soluzione? La risposta è semplice, perché avrebbe fatto lievitare i costi di MoonCloud alle stelle, richiedendo di allocare un altissimo numero di VM per OpenVPN server, e più clienti MoonCloud acquisisce, più questi costi aumenterebbero!

Ecco quindi il problema a cui ho dovuto trovare una soluzione: *come gestire la conflittualità tra gli indirizzi IP delle reti target senza dover allocare un server OpenVPN per ogni cliente?* Prima di analizzare la risposta a tale quesito, si meglio cosa succederebbe in caso in cui vi siano dei pacchetti generati da dei *probe* destinati ad una delle due reti target, e che vi sia un conflitto di indirizzamenti. Si ipotizza di lavorare nella seguente configurazione:

- rete target 1: 192.168.100.0/24 (cliente A)
- rete target 2: 192.168.100.0/24 cliente (B)
- rete MoonCloud: 192.168.200.0/24
- indirizzo IP interno VPN server: 192.168.200.20
- indirizzo IP pubblico VPN server: 100.4.78.5
- indirizzo IP del *Docker host* per client A: 192.168.200.2
- indirizzo IP del *Docker host* per client B: 192.168.200.3
- indirizzo IP VPN client A: 192.168.100.50
- indirizzo IP VPN client B: 192.168.100.150
- subnet VPN: 10.7.0.0/24
- indirizzo IP scheda di rete virtuale VPN server: 10.7.0.1
- indirizzo IP scheda di rete virtuale VPN client A: 10.7.0.2
- indirizzo IP scheda di rete virtuale VPN client B: 10.7.0.3
- CommonName del VPN client A: clientA
- CommonName del VPN client B: clientB

Si supponga infine che si adottano le soluzioni discusse precedentemente, tra cui l'uso del NAT *al contrario*, ed il NAT *lato server*. Le direttive `iroute` dei file di configurazione relativi ai due client contengono entrambi l'indicazione della rete `192.168.100.0 255.255.255.0`. Ciò che si descrive di seguito è *ipotetico*, poiché non è nemmeno detto che OpenVPN accetti due `iroute` di due client diversi con lo stesso contenuto, supponendo che ciò sia possibile, accadrebbe la situazione che ora si dettaglia.

1. Un *probe* per client *A* genera un pacchetto, la sorgente di tale pacchetto è `192.168.200.2`, la destinazione `192.168.100.254`; si supponga che tale pacchetto sia una richiesta HTTP a tale indirizzo (e che quindi sia un TCP SYN).
2. Il sistema operativo del *Docker host A* consulta la propria routing table ed instrada il pacchetto verso `192.168.200.20`
3. contemporaneamente, un altro *probe* per client *B* genera un pacchetto, la sorgente di tale pacchetto è `192.168.200.3`, e la sfortuna vuole che la destinazione sia `192.168.100.254`, e che in questo caso si tratti di un pacchetto di inizio di una connessione SSH.
4. Il sistema operativo del *Docker host B* consulta la propria routing table ed instrada il pacchetto verso `192.168.200.20`
5. il sistema operativo del VPN server riceve il pacchetto da `192.168.200.2`, consulta la routing table e trova l'entry `192.168.100.0/24` via `10.7.0.1`, quindi inoltra alla scheda di rete virtuale di OpenVPN
6. il sistema operativo del VPN server riceve il pacchetto da `192.168.200.3`, consulta la routing table e trova l'entry `192.168.100.0/24` via `10.7.0.1`, quindi inoltra alla scheda di rete virtuale di OpenVPN
7. OpenVPN riceve i due pacchetti, essi hanno entrambi come destinazione `192.168.100.254`
8. OpenVPN consulta la propria routing table e trova che sia client *A* sia client *B* sono responsabili per tali pacchetti, per cui si supponga (non è dato sapere come OpenVPN si comporti davvero in questo caso<sup>4</sup>) che entrambi i due pacchetti siano inviati ad entrambi i due client.
9. Entrambi i due pacchetti sono ricevuti da `clientA`, il quale provvede a scriverli sulla propria scheda di rete virtuale
10. i pacchetti sono quindi ricevuti dal sistema operativo ed inviati a `192.168.100.254` nella rete *A*
11. parallelamente, i due pacchetti sono ricevuti da `clientB` ed inviati a `192.168.10.254` nella rete *B*.

---

<sup>4</sup>Non è dato sapere come OpenVPN si comporterebbe davvero in un caso come questo, né lo si è verificato in pratica perché ci si è piuttosto concentrati sulla soluzione di tale problema.

12. A questo punto si possono verificare diverse combinazioni più o meno sfortunate, qua se ne elenca solo una, ma abbastanza esemplificativa:
- 192.168.100.254 *A* li riceve e li elabora, si supponga che per una sfortunata coincidenza su tale host sia installato sia un server HTTP (a cui era destinato il pacchetto proveniente da 192.168.200.2) sia un server SSH: entrambi producono delle risposte “valide” per i due pacchetti.
  - 192.168.100.254 *B* riceve anch’esso i due pacchetti e li elabora, si ricorda che a tale host era destinato solo un pacchetto per il server SSH, e che non vi sia alcun server HTTP. Per tale motivo, la risposta al pacchetto SSH è una risposta “valida”, mentre invece la risposta alla richiesta HTTP, si supponga sulla porta 80, produce un `connection reset by peer`<sup>5</sup>.
13. i due host nelle due reti inoltrano quindi le risposte ai due client VPN, i quali applicano il NAT ed inviano al server OpenVPN
14. per ragioni non note, i pacchetti di risposta provenienti dalla rete del cliente *B* arrivano prima al server, OpenVPN provvede quindi a scriverli sulla scheda di rete virtuale
15. il sistema operativo li riceve ed riapplica il NAT, ed ora iniziano i veri problemi: si supponga che `iptables` riesca ad applicare il NAT ad entrambi i pacchetti (si ricorda che quello contenente l’RST di risposta ad un web server non esistente è una risposta che non dovrebbe esistere)
16. il pacchetto di risposta al TCP SYN per l’instaurazione di una connessione in vista della richiesta HTTP viene inviato al *Docker host* 192.168.200.2, il cui *probe* vedrà quindi un risultato diverso da quello reale (il pacchetto di risposta dalla rete *A* deve ancora arrivare)
17. il pacchetto di risposta all’inizio della connessione SSH (sarà un SYN/ACK) viene ricevuto dal *Docker host* 192.168.200.3 e quindi inviato al *probe*, in questo caso il *probe* vedrà il risultato reale.
18. Ora il server OpenVPN riceve i due pacchetti provenienti dalla rete *A*, che sono quindi scritti sulla scheda di rete virtuale e ricevuti dal sistema operativo
19. `iptables` dovrebbe riapplicare NAT come descritto tre punti prima, che cosa succede? Si riesce ad applicare il NAT? Anche ammesso che si riesca, le risposte arriverebbero duplicate, ammesso che non vengano scartate o che si verifichino qualche altra anomalia.

E’ quindi chiaro che, qualsiasi cosa succeda, i *probe* raramente riceveranno le risposte legittime, e durante il percorso possono verificarsi molti inconvenienti che facciano sì che tale risposte addirittura non raggiungano *mai* i *probe*. Si ribadisce

---

<sup>5</sup>Il protocollo TCP prescrive che quando si invii un SYN verso una porta chiusa si risponda con un RST.

che lo scenario descritto è *ipotetico*, non è nemmeno detto che OpenVPN accetti la situazione di conflitto presente nei file di configurazione *client-specific*.

### 3.4.2 La soluzione

Mentre questo problema emergeva con il prof. Anisetti, la prima soluzione che entrambi abbiamo abbozzato è stata quella in cui MoonCloud *rimappi* ogni rete target su una nuova rete, stabilita da MoonCloud e garantita essere univoca. “*Rimappare*” una rete significa trovare una nuovo NET ID per tale rete, e *spostare* gli indirizzi IP originali nel nuovo NET ID. Lato MoonCloud, i *probe* conoscono gli indirizzi IP modificati (garantiti univoci), ed il rimappaggio verso quelli originali viene effettuato sul device VPN client.

Entrambi concordi che questa fosse la soluzione corretta, si trattava ora di capire come potessa il client rimappare gli indirizzi IP, tenendo presente che tale rimappaggio deve essere perfettamente aderente a quello stabilito da MoonCloud. Prima di proseguire, si chiarisce meglio la terminologia che da qui in poi sarà utilizzata:

- “*rete originale OTN (Original Target Network)*”: l’indirizzo IP di rete della rete target così come è in realtà
- “*IP originale OTI (Original Target IP)*”: l’indirizzo IP di un host nella rete target così come è in realtà
- “*rete mappata MTN (Mapped Target Network)*”: il nuovo NET ID assegnato ad una rete target
- “*IP mappato MTI (Mapped Target IP)*”: l’indirizzo IP appartenente alla *rete mappata* a cui corrisponde un indirizzo IP originale
- “*rete rimappata RTN (Remapped Target Network)*”: l’indirizzo IP di rete della rete target così come è in realtà, ovvero uguale alla *rete originale*
- “*IP rimappato RTI (Remapped Target IP)*”: corrisponde all’*IP originale*.
- “*mappare una rete*”: assegnare alla *rete originale* una *rete mappata*
- “*mappare un indirizzo IP*”: assegnare un *IP mappato* ad un *IP originale*
- “*rimappare una rete*”: procedimento inverso del “*mappare una rete*”
- “*rimappare un indirizzo IP*”: procedimento inverso del “*mappare un indirizzo IP*”

Il motivo per cui esistono due sinonimi sarà presto chiaro. Si definiscono quindi le seguenti funzioni *deterministiche* (con  $S$  corrispondente ad un certo server OpenVPN):

- $map\_network : OTN, S \rightarrow MTN$ , una funzione che data una *OTN* ritorna una *MTN*, garantita univoca per ogni server

- $map\_address : OTI, OTN, MTN, S \rightarrow MTI$ , una funzione che dato in input:
  - $OTI \mid OTI \in OTN$ : un indirizzo IP originale appartenente alla rete originale
  - $OTN$ : la rete originale
  - $MTN \mid map\_network(OTN) = MTN$ : la rete mappata della rete originale
  - $S$ : il server OpenVPN a cui  $OTN$  è assegnata

ritorni  $MTI$ : un indirizzo  $\in MTN$  corrispondente a  $OTI$ .

- $remap\_network : MTN, S \rightarrow OTN$ , una funzione che data una rete mappata ritorna la corrispondente rete originale assegnata ad un certo server, con  $OTN = RTN$ .
- $remap\_address : MTI, MTN, OTN, S \rightarrow OTI$ , una funzione che dato in input:
  - $MTI \in MTN$ : indirizzo IP mappato
  - $MTN \mid map\_network(OTN) = MTN$ : la rete  $OTN$  mappata
  - $OTN$ : la rete originale
  - $S$  il server OpenVPN a cui  $OTN$  è assegnata

ritorna  $OTI$ : un indirizzo  $\in OTN$  corrispondente a  $MTN$ , e  $RTI = OTI$

Per queste funzioni valgono le seguenti uguaglianze:

- $remap\_network(map\_network(OTN, S), S) = map\_network(OTN, S)^{-1}$
- $remap\_address(map\_address(OTN, OTI, MTN, S), MTN, OTN, S) = map\_address(OTN, OTI, MTN, S)^{-1}$

In altre parole, ogni funzione è l'inverso dell'altra.

*Come effettuare il mappaggio?* Per rispondere a questa domanda, si vede ora un nuovo flusso che include questa nuova funzionalità. Si supponga di avere la seguente configurazione:

- rete target  $OTN$ : 192.168.100.0/24
- rete mappata:  $map\_network(OTN, S) = 192.168.1.0./24$
- rete MoonCloud: 192.168.200.0/24
- indirizzo IP interno VPN server  $S$ : 192.168.200.20
- indirizzo IP pubblico VPN server: 100.4.78.5
- indirizzo IP del *Docker host*: 192.168.200.2
- indirizzo IP VPN client ( $OTI \in OTN$ ): 192.168.100.50

- subnet VPN: 10.7.0.0/24
- indirizzo IP scheda di rete virtuale VPN server: 10.7.0.1
- indirizzo IP scheda di rete virtuale VPN client: 10.7.0.2
- CommonName del VPN client: client1
- target da analizzare ( $OTI \in OTN$ ): 192.168.1.254
- target mappato:  $MTI \in MTN = map\_address(192.168.100.254, 192.168.100.0/24, 192.168.1.0/24) = 192.168.1.254$

Le reti mappate sostituiscono le reti originali nella configurazione delle rotte lato MoonCloud. Quindi si mostra un estratto dei file `client-up.sh` e `client-down.sh`, come si può vedere, le rotte aggiunte e rimosse sono quelle mappate.

```
# /etc/openvpn/server/cs/client-up.sh

#!/bin/bash

if [ "$common_name" == "client1" ]; then
    ip route add 192.168.1.0/24 via 10.7.0.1
    iptables -t nat -A POSTROUTING -d 192.168.1.0/24 -j
    ↪ MASQUERADE
fi

# /etc/openvpn/server/cs/client-down.sh

#!/bin/bash

if [ "$common_name" == "client1" ]; then
    ip route del 192.168.1.0/24 via 10.7.0.1
    iptables -t nat -D POSTROUTING -d 192.168.1.0/24 -j
    ↪ MASQUERADE
fi
```

Allo stesso modo, il file `client1` che contiene le direttive client-specific avrà il seguente contenuto:

```
iroute 192.168.1.0 255.255.255.0
```

Infine, per applicare la soluzione di NAT *al contrario*, sul device client sarà eseguito il comando `iptables`:

```
iptables -t nat -A POSTROUTING -d 192.168.100.0/24 -s
↪ 10.7.0.0/24 -j MASQUERADE
```

Si noti che in questo comando la rete di destinazione è la *rete originale*. I passi eseguiti sono quindi i seguenti:



1. un cliente registra una nuova rete:  $192.168.100.0/24$
2. MoonCloud sceglie a quale server  $S$  assegnare tale rete, in questo caso è il VPN server all'indirizzo IP interno  $192.168.200.20$
3. MoonCloud mappa la rete in input  $MTN = map\_network(192.168.100.0/24, S)$ , ed  $MTN = 192.168.1.0/24$
4. si crea un nuovo device client, propriamente configurato
5. il device client viene collegato alla rete target e si connette al server OpenVPN
6. si alloca un nuovo *Docker host*, nell'esempio in questione è  $192.168.200.2$
7. il cliente configura una nuova analisi, specificando come target:  $192.168.100.254$
8. il target di tale analisi passato ai *probe* è il risultato dell'applicazione di  $map\_address(192.168.100.254, 192.168.100.0/24, 192.168.1.0/24)$ , ovvero  $192.168.1.254$  (detto *MTI*)
9. il *probe* genera un pacchetto destinato a  $192.168.1.254$ , cioè *MTI*
10. il sistema operativo del *Docker host* ha configurato una rotta del tipo:  $192.168.1.0/24$  via  $192.168.200.20$ , quindi il pacchetto viene inviato al server VPN
11. il sistema operativo del server  $S$  riceve il pacchetto, a sua volta ha configurato la rotta seguente:  $192.168.1.0/24$  via  $10.7.0.1$ ; tale rotta è stata configurata tramite il file di script `client-up.sh`.
12. a questo punto il pacchetto passa nello stato di `POSTROUTING`, e come si applica la regola di NAT descritta nel file sopra, quindi l'IP sorgente del pacchetto in questione è ora  $10.7.0.1$
13. il pacchetto viene inviato all'interfaccia virtuale di OpenVPN, e da esso ricevuto
14. OpenVPN consulta la propria routing table interna, in conseguenza di ciò quindi invia il pacchetto a `Client1` lungo la connessione TCP con esso
15. il pacchetto viene ricevuto dal device client, l'indirizzo IP di destinazione è quello mappato, cioè  $192.168.1.254$
16. il pacchetto viene scritto sull'interfaccia virtuale e ricevuto dal kernel del sistema operativo
17. in un modo che poi verrà descritto, il sistema operativo di `Client100` applica la seguente funzione per ottenere  $OTI = RTI$  cioè l'indirizzo IP effettivo del target:  $OTI = remap\_address(192.168.1.254, 192.168.1.0/24, 192.168.100.254)$ , il risultato è  $192.168.100.254$ , il quale ovviamente corrisponde all'indirizzo IP specificato dal cliente come target dell'analisi.

18. Si modifica l'indirizzo IP destinazione del pacchetto, inserendo come destinazione *OTI*, cioè 192.168.100.254
19. a questo punto il pacchetto è in stato di `POSTROUTING` e si applica il *NAT al contrario*, l'indirizzo IP sorgente è ora 192.168.100.50 (cioè l'IP del device client)
20. il pacchetto viene quindi inviato a 192.168.100.254
21. 192.168.100.254 riceve il pacchetto, lo elabora, invia una risposta con destinazione 192.168.100.50
22. il pacchetto viene ricevuto da 192.168.100.50 (*Client100*), come prima cosa si riapplica l'inverso di *NAT al contrario*, modificando l'indirizzo IP destinazione in 10.7.0.1 (l'indirizzo IP sorgente del pacchetto di richiesta inviato dal server OpenVPN)
23. il sistema operativo di *Client1* ora provvede a modificare tale pacchetto, andando a specificare che l'indirizzo IP sorgente non è più 192.168.100.254 (IP originale – *OTI*), ma 192.168.1.254 (IP mappato – *MTI*). Per farlo, applica: `source = map_address(192.168.100.254, 192.168.100.0/24, 192.168.1.0/24)` ; a questo il pacchetto è conforme a quello inviato da OpenVPN server dopo il *NAT lato server*, poiché l'indirizzo IP sorgente è quello mappato, cioè quello che MoonCloud conosce.
24. il sistema operativo quindi inoltra il pacchetto alla scheda di rete virtuale OpenVPN, il quale lo riceve e lo invia al server
25. il server OpenVPN *S* riceve il pacchetto, lo decifra, lo scrive sulla scheda di rete virtuale
26. il sistema operativo del server *S* lo riceve, a questo punto si riapplica il *NAT lato server*, l'indirizzo IP destinazione passa da 10.7.0.1 a 192.168.200.2: a questo punto preciso il pacchetto è esattamente la risposta precisa alla richiesta generata dal *Docker host*.
27. Il pacchetto viene quindi inviato al *Docker host* e quindi al *probe*.

Poiché questa descrizione è abbastanza lunga, è possibile trovarne una più schematica nella prossima sezione: *configurazione finale*.

Detto questo, non resta che spiegare *come* eseguire le funzioni di mappaggio e rimappaggio.

Lato server, *map\_network* è una funzione eseguita dal microservizio da me scritto e che sarà dettagliato nel prossimo capitolo. Per ora basti sapere che nel momento in cui avviene l'*enrollment* di un nuovo cliente, le sue reti sono mappate su nuove reti in modo che siano univoche.

Ogni volta che si specifica un target per l'analisi, il cliente inserisce l'indirizzo IP vero (*OTI*), quindi si contatta il mio microservizio per ottenerne la sua versione mappata (*MTI*): tale indirizzo *MTI* è l'IP target dei *probe* che fanno analisi.

All'interno di MoonCloud, ogni vi sono varie reti locali che comprendono il VPN server, responsabile per un certo numero di clienti, ed un certo numero di *Docker*

*host* i quali effettivamente fanno l'analisi tramite tale server. Ogni rete è isolata dalle altre, pertanto è possibile che due server VPN abbiano due reti target mappate uguali, ma questo non crea conflitto perché i due server sono isolati tra loro. Allo stesso modo, un server è ovviamente in grado di gestire più clienti contemporaneamente, poiché l'univocità delle reti mappate è garantita *per-server*.

L'ultima cosa che rimane da capire è come effettuare mappaggio e rimappaggio lato client; questo problema in particolare è stato il più difficile da risolvere. In particolare, il software responsabile di questa traduzione deve imprescindibilmente *andare d'accordo* con il NAT *al contrario* (poiché si è visto che è la soluzione all'Impossibilità di configurare rotte sulla rete target), e deve effettuare una modifica degli indirizzi IP che sia perfettamente aderente al mapping stabilito da MoonCloud. ... L'illuminazione è stato capire che questa operazione di mapping altro non è che un NAT 1:1, ed una volta afferrato questo è stato (relativamente) facile scrivere delle regole *iptables* che mi consentissero di farlo.

In generale, supponendo di voler fare NAT (quindi mappaggio e rimappaggio) da 192.168.1.254 a 192.168.100.254, con pacchetti provenienti dalla subnet 10.7.0.0/24, è possibile utilizzare i seguenti due comandi:

```
iptables -t nat -A PREROUTING -d 192.168.1.254 -s  
→ 10.7.0.0/24 -j DNAT --to-destination 192.168.100.254  
iptables -t nat -A POSTROUTING -s 192.168.100.254 -d  
→ 10.7.0.0/24 -j SNAT --to-source 192.168.1.254
```

Il primo comando viene utilizzato per modificare l'indirizzo IP destinazione, da quello mappato noto a MoonCloud (*MTI*) a quello originale (*OTI*); il secondo per modificare l'IP sorgente, da *OTI* a *MTI*.

Questa soluzione ha il vantaggio di integrarsi perfettamente con il NAT *al contrario* (poiché è eseguita dallo stesso software), e sfrutta la possibilità di un pacchetto di poter passare per più *hook* nel kernel Linux.

- Pacchetti MoonCloud → target:
  1. modifica IP destinazione (*hook* PREROUTING)
  2. modifica IP sorgente (MASQUERADE, *hook* POSTROUTING)
- Pacchetti target → MoonCloud:
  1. modifica IP destinazione (MASQUERADE, *hook* PREROUTING)
  2. modifce IP sorgente (*hook* POSTROUTING)

## 3.5 Recap

Prima di passare ad analizzare come tutte le soluzioni appena indicate si combinano tra loro, è utile fare un breve riassunto dei problemi affrontati e delle soluzioni attuate.

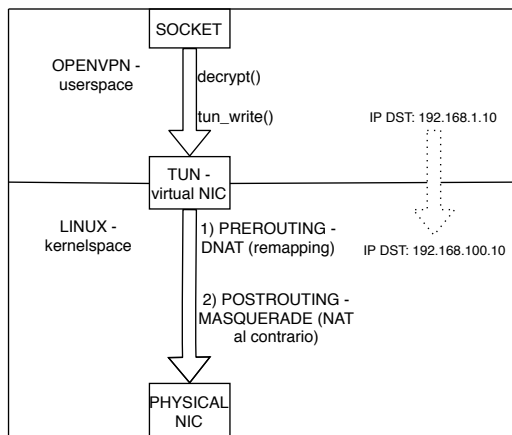


Figura 3.1: IP Mapping: MoonCloud → Target. Si mostra un esempio di ciò che accade nel VPN client quando si riceve un pacchetto tramite la VPN. Per essere rigorosi occorre notare che una volta che OpenVPN invia qualcosa su un socket, il controllo passa di nuovo al sistema operativo che scrive quindi il pacchetto sulla scheda di rete fisica, ma questo non è di interesse per il problema in questione.

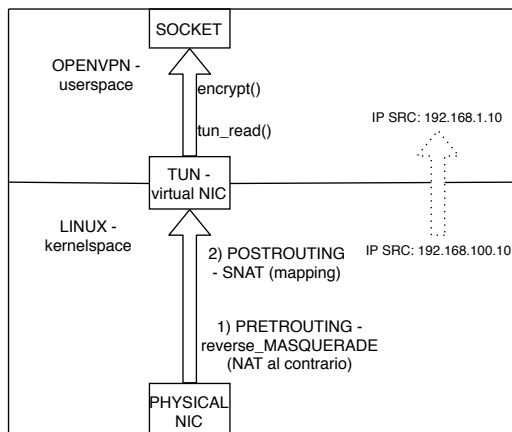


Figura 3.2: IP Mapping: Target → MoonCloud. In questo caso si mostra cosa succede con i pacchetti in risposta a richieste provenienti da MoonCloud lungo la VPN.

**NAT al contrario** I pacchetti provenienti dalla rete MoonCloud hanno come IP sorgente un IP appartenente a tale rete, affinché le risposte possano tornare indietro, occorre che in ciascuna rete target sia configurata la seguente rotta: *rete: rete-MoonCloud via: IP-client-VPN*. Essa può essere inserita nel default gateway di ciascuna rete oppure in ogni host di tali reti. Tuttavia, non è possibile intervenire sui dispositivi appena elencati, pertanto la soluzione è quella di applicare NAT ai pacchetti provenienti da MoonCloud e diretti alla rete target, quindi ai pacchetti in uscita dal device VPN client. In questo modo, i pacchetti hanno come indirizzo IP sorgente quello del device client, il quale si trova nella stessa rete degli host target, che quindi inviano ad esso le proprie risposte, senza bisogno di configurare alcuna rotta aggiuntiva,

**Bypassing della direttiva “route ...”** Lo scopo di tale direttiva, sia essa posta nel client o nel server, è quella di indicare quali rotte sono raggiungibili mediante l'altro. Tuttavia, poiché OpenVPN legge il file di configurazione solo all'avvio, e non si sa in principio quali client saranno connessi al server, diventa un problema inserirla nel file del server. L'effetto di questa opzione è di aggiungere una nuova rotta nel kernel del sistema operativo così fatta: `<net> via <virtual-NIC's-IP>`. Per ovviare a questo problema si aggiunge questa rotta con il relativo comando di sistema al momento in cui un client si connette, e la si cancella quando si disconnette; si sfruttano le direttive `client-connect` e `client-disconnect` che consentono di specificare uno script da eseguire in uno dei due eventi appena descritti.

**NAT lato server** Nell'OpenVPN client, i pacchetti ricevuti sulla scheda di rete virtuale sono risposte provenienti dagli host nella rete target, ed hanno come IP di destinazione un IP appartenente alla rete in cui si trovano i *Docker host*. Affinché OpenVPN sappia che tali IP siano raggiungibili tramite la VPN, è necessario che il server pubblicizzi tale rotte, e questo viene normalmente fatto mediante la direttiva `push "route <rotta-da-pushare>"`, il cui effetto è che il client aggiunga tale rotta alla routing table del kernel. Tuttavia, non si può sapere a priori quali reti saranno dietro il server, a tale scopo il sistema operativo del server effettua del NAT sui pacchetti diretti alla scheda di rete virtuale OpenVPN, andando a settare l'IP sorgente come l'IP della scheda di rete virtuale. Spostandosi ora nel client, si ha che ora i pacchetti di risposta a MoonCloud hanno come IP destinazione quello virtuale del server, il quale si trova nello spazio di indirizzamento della VPN, quindi il sistema operativo sa come instradarlo. Infatti, l'indirizzo IP della scheda di rete virtuale del client, si trova nella stessa subnet di quello del server, il cui effetto è quello di avere una rotta automaticamente aggiunta dal sistema operativo: `<VPN-subnet> via <virtual-NIC's IP>`.

**IP mapping** Per poter connettere tra di loro  $n$  reti mediante una VPN è necessario che ciascuna rete partecipante abbia un NET ID diverso. Poiché si vuole utilizzare un server per connettere diverse reti target di diversi clienti, è ragionevole presumere che vi sia prima o poi un conflitto, e che due reti abbiano lo stesso NET ID. Per superare questo problema è stato introdotto il nuovo concetto di *mappaggio delle reti/degli indirizzi IP*, per il quale

MoonCloud assegna ad ogni rete un nuovo NET ID, e mappa gli IP originali nel nuovo NET ID. Questo mappaggio è garantito univoco. Lato client, si utilizza `iptables` per modificare gli header IP e rendere ai clienti questa complessa operazione del tutto trasparente.

## 3.6 Configurazione finale

Dopo aver visto tutti i problemi affrontati e come sono stati risolti, è il momento di vedere come sono configurati *davvero* i server ed i client VPN in MoonCloud. Come per il mappaggio degli indirizzi IP, il mio microservizio, si occupa di automatizzare il tutto, e di generare tutto ciò che serve.

Durante tutte le sottosezioni successive, si supporrà la seguente configurazione di esempio:

- rete target 1: 192.168.100.0/24 (cliente A)
- rete target 1 mappata: 192.168.1.0/24
- rete MoonCloud: 192.168.200.0/24
- indirizzo IP interno VPN server: 192.168.200.20
- indirizzo IP pubblico VPN server: 100.4.78.5
- CommonName del VPN server: `ovpn1.moon-cloud.eu`
- indirizzo IP del *Docker host* per client A: 192.168.200.2
- indirizzo IP VPN client A: 192.168.100.50
- subnet VPN: 10.7.0.0/24
- indirizzo IP scheda di rete virtuale VPN server: 10.7.0.1
- indirizzo IP scheda di rete virtuale VPN client A: 10.7.0.2
- CommonName del VPN client A: `office1.company.com`

Si tenga presente che ci concentrerà sulla configurazione della VPN ed anche l' *IP mapping* viene affrontato solo dal punto di vista della VPN. Per avere ulteriori informazioni sulla generazione dei certificati, sull'algoritmo che mappa una rete o che mappa un IP, capitolo 6 è dedicato ad esso. Nei file `client-up.sh` e `client-down.sh` si noteranno alcune regole `iptables` che non state ancora spiegate: esse sono funzionali ad incrementare la sicurezza della VPN, e sono descritte nel prossimo capitolo.

### 3.6.1 Configurazione del server

Al momento della creazione di un nuovo server, viene generato il seguente file di configurazione:

```
# /etc/openvpn/ovpn1.conf

# ruolo
mode server
tls-server

# protocollo L4
proto tcp

# configurazione NIC
# virtuale
dev-type tun
dev ovpn-1

persist-tun
persist-key

# porta di ascolto 443
port 443

# VPN subnet
server 10.7.0.0 255.255.255.0

topology subnet

# path per client-specific file
client-config-dir /etc/openvpn/ovpn1/ccd

# TLS configuration
remote-cert-eku "TLS Web Client Authentication"
tls-version-min 1.2

tls-cipher TLS-ECDHE-ECDSA-WITH-AES-256-GCM-SHA384:TLS-
↳ ECDHE-ECDSA-WITH-CHACHA20-POLY1305-SHA256
cipher AES-256-GCM

# keys and certs
ca /etc/openvpn/certs/ca.crt
key /etc/openvpn/ovpn1/certs/ovpn1.key
cert /etc/openvpn/ovpn1/certs/ovpn1.crt

# per ECDH
dh none
```

```

# nuove chiavi ogni 20 minuti
reneg-sec 1200

# log configuration
log /var/log/openvpn/ovpn1/openvpn.log
verb 4
status /var/log/openvpn/ovpn1/openvpn-status.log

# livello di sicurezza per comandi user-defined
script-security-level 2

# scripts
client-connect /etc/openvpn/ovpn1/cs/client-up.sh
client-disconnect /etc/openvpn/ovpn1/cs/client-down.sh

# opzioni 'ping'
keepalive 10 60

```

Si può notare che non vi è nessuna direttiva ulteriore che non è stata spiegata in precedenza, l'unica aggiunta è un'ulteriore ciphersuite:

TLS-ECDHE-ECDSA-WITH-CHACHA20-POLY1305-SHA256. Per considerazioni sulla sicurezza si rimanda al capitolo specifico, ciò che è importante sapere è che queste sono le uniche ciphersuite che il server utilizzerà. ChaCha20-Poly1305 ha ottime prestazioni in implementazioni solo software, ed in particolare è molto più veloce di AES-GCM in mancanza di accelerazione hardware crittografica. Queste due ciphersuite sono state selezionate poiché sono di fatto, le più sicure, per inciso, sono le uniche due incluse in TLS 1.3. Attualmente non c'è supporto per ChaCha20-Poly1305 per il Data Channel.

Il file mostrato sopra verrà posizionato in `/etc/openvpn/ovpn1.conf`, ed ovviamente tutti i percorsi indicati in questo file devono esistere.

I file `/etc/openvpn/ovpn1/cs/client-up.sh` e `/etc/openvpn/ovpn1/cs/client-down.sh`:

```

# /etc/openvpn/ovpn1/cs/client-up.sh

#!/bin/bash

if [ "$common_name" == "officel.company.org" ]; then
    ip route add 192.168.1.0/24 via 10.7.0.1
    iptables -t nat -A POSTROUTING -d 192.168.1.0/24 -j
    ↪ MASQUERADE

    # non fondamentali, verranno spigate in seguito.
    iptables -t filter -I OUTPUT -d 192.168.1.0/24 -o ovpn+
    ↪ -m state --state NEW,RELATED,ESTABLISHED -j ACCEPT
    iptables -t filter -I INPUT -s 192.168.1.0/24 -i ovpn+ -m
    ↪ state --state RELATED,ESTABLISHED -j ACCEPT

```



```

iptables -t filter -I FORWARD -s 192.168.1.0/24 -i ovpn+
↪ -m state --state RELATED,ESTABLISHED -j ACCEPT
fi

# /etc/openvpn/ovpn1/cs/client-down.sh

#! /bin/bash

if [ "$common_name" == "officel.company.org" ]; then
    ip route add 192.168.1.0/24 via 10.7.0.1
    iptables -t nat -D POSTROUTING -d 192.168.1.0/24 -j
    ↪ MASQUERADE

    # verranno spiegate in seguito
    iptables -t filter -D OUTPUT -d 192.168.1.0/24 -o ovpn+
    ↪ -m state --state NEW,RELATED,ESTABLISHED -j ACCEPT
    iptables -t filter -D INPUT -s 192.168.1.0/24 -i ovpn+ -m
    ↪ state --state RELATED,ESTABLISHED -j ACCEPT
    iptables -t filter -D FORWARD -s 192.168.1.0/24 -i ovpn+
    ↪ -m state --state RELATED,ESTABLISHED -j ACCEPT
fi

```

Il file /etc/openvpn/ovpn1/ccd/officel.company.org:

```
iroute 192.168.1.0 255.255.255.0
```

### 3.6.2 Configurazione del client

Come prima cosa, ecco come si presenta il file di configurazione di OpenVPN del client.

```

# /etc/openvpn/officel.conf

client

# used protocol
proto tcp

# server address and port
remote ovpn1.moon-cloud.eu 443

# virtual NIC configuration
dev-type tun
dev ovpn-officel

persist-tun
persist-key

```

```
group nogroup
user nobody
```

```
# TLS configuration
```

```
remote-cert-eku "TLS Web Server Authentication"
```

```
tls-version-min 1.2
```

```
tls-cipher TLS-ECDHE-ECDSA-WITH-AES-256-GCM-SHA384:TLS-ECDHE-ECDSA-WITH
```

```
cipher AES-256-GCM
```

```
# chiavi e certificati
```

```
ca /etc/openvpn/certs/ca.crt
```

```
key /etc/openvpn/officel/certs/officel.key
```

```
cert /etc/openvpn/officel/certs/officel.crt
```

```
# log configuration
```

```
log /var/log/openvpn/officel/openvpn.log
```

```
verb 4
```

```
status /var/log/openvpn/officel/openvpn-status.log
```

Si possono osservare le seguenti differenze rispetto al server:

- mancanza della direttiva “`topology subnet`” poiché è riservata al server
- mancanza della direttiva “`client-config-dir`” per la stessa ragione
- mancanza delle direttive “`client-connect`” e “`client-disconnect`” per la stessa ragione
- aggiunta della direttiva “`remote`” che indica come raggiungere il server
- aggiunta delle direttive “`user nobody`” e “`group nogroup`” che consentono di fare il downgrade dei privilegi. Non è possibile farlo sul server poiché è necessario che abbia privilegi amministrativi per eseguire i comandi contenuti negli script di connessione e disconnessione dei client.

Si è supposto di mappare `192.168.100.0/24` in `192.168.1.0/24`, ed ogni IP della rete originale viene mappato nella nuova rete nel seguente modo: `192.168.100.1 ↔ 192.168.1.1`, e così via fino a `192.168.100.254 ↔ 192.168.1.254`. Si produce quindi uno script di comandi per iptables così composto:

```
# !/bin/bash
```

```
# ip mapping, changing destination addresses
```

```
# (done for request packets)
```

```
iptables -t nat -A PREROUTING -d 192.168.1.1 -s 10.7.0.0/24  
→ -j DNAT --to-destination 192.168.100.1
```

```
iptables -t nat -A PREROUTING -d 192.168.1.2 -s 10.7.0.0/24  
→ -j DNAT --to-destination 192.168.100.2
```

```
iptables -t nat -A PREROUTING -d 192.168.1.3 -s 10.7.0.0/24  
→ -j DNAT --to-destination 192.168.100.3
```

```
# ...
iptables -t nat -A PREROUTING -d 192.168.1.254 -s
↳ 10.7.0.0/24 -j DNAT --to-destination 192.168.100.254

# 'reverse NAT'
iptables -t nat -A POSTROUTING -d 192.168.1.0/24 -s
↳ 10.7.0.0/24 -j MASQUERADE

# ip mapping, changing source addresses
# (done for response packets)
iptables -t nat -A POSTROUTING -s 192.168.100.1 -d
↳ 10.7.0.0/24 -j SNAT --to-source 192.168.1.1
iptables -t nat -A POSTROUTING -s 192.168.100.2 -d
↳ 10.7.0.0/24 -j SNAT --to-source 192.168.1.2
iptables -t nat -A POSTROUTING -s 192.168.100.3 -d
↳ 10.7.0.0/24 -j SNAT --to-source 192.168.1.3
# ...
iptables -t nat -A POSTROUTING -s 192.168.100.254 -d
↳ 10.7.0.0/24 -j SNAT --to-source 192.168.1.254
```

## 3.7 Testing

Dopo aver visto come configurare i partecipanti alla VPN, si passa a descrivere i test che ho effettuato per verificare che il tutto funzionasse.

### 3.7.1 Test 1

Il primo test è stato eseguito sulla configurazione spiegata all'inizio del capitolo, eccola spiegata nel dettaglio:

- OpenVPN server ospitato su AWS, raggiungibile da nome DNS, sistema operativo Ubuntu Server, indirizzo IP privato 172.31.40.221
- Docker host ospitato su AWS, nella stessa rete privata del server, sistema operativo Ubuntu Server, indirizzo IP privato 172.31.43.141
- due reti virtuali, create con VirtualBox:
  - rete 1, detta net50, costituita da:
    - \* firewall stateful, NAT verso l'esterno e traffico consentito solo sulle porte 80 e 443 TCP, indirizzo IP 192.168.100.254, sistema operativo Alpine Linux
    - \* client VPN, sistema operativo Ubuntu desktop, indirizzo IP 192.168.100.20, default gateway 192.168.100.254
    - \* due host Alpine Linux, indirizzi IP rispettivamente 192.168.100.30 e 192.168.100.40, entrambi con un server web in ascolto sulla porta 80

- rete 2, detta `net200`, costituita da:
  - \* firewall stateful, NAT verso l'esterno e traffico consentito solo sulle porte 80 e 443 TCP (e 53 UDP – DNS), indirizzo IP `192.168.100.254`, sistema operativo `Alpine Linux`
  - \* client VPN, sistema operativo `Ubuntu desktop`, indirizzo IP `192.168.100.20`, default gateway `192.168.100.254`
  - \* un host `Alpine Linux`, con indirizzo IP `192.168.100.30` ed un server web in ascolto sulla porta 80<sup>6</sup>

Le due reti sono state appositamente scelte uguali per testare il funzionamento dell'*IP mapping*. In particolare:

- `net50` è stata mappata su `192.168.50.0/24`
  - `192.168.100.30 ↔ 192.168.50.30`
  - `192.168.100.40 ↔ 192.168.50.40`
- `net200` è stata mappata su `192.168.200.0/24`
  - `192.168.100.30 ↔ 192.168.200.30`

La figure 3.7.1 dà una rappresentazione grafica della topologia realizzata.

I due firewall sono stati configurati volutamente in modo restrittivo, al fine di verificare se OpenVPN fosse in grado di passarli. Riporto quindi i comandi iptables utilizzati (essi sono gli stessi per entrambi i firewall):

```
#!/bin/ash
# ash è la shell di Alpine

# natting
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE

# dns
iptables -t filter -A FORWARD -p udp --dport 53 -s
↳ 192.168.100.0/24 -j ACCEPT
iptables -t filter -A FORWARD -p udp --sport 53 -d
↳ 192.168.100.0/24 -j ACCEPT

# http stateful
iptables -t filter -A FORWARD -p tcp --dport 80 -s
↳ 192.168.100.0/24 -m state --state
↳ NEW,RELATED,ESTABLISHED -j ACCEPT
iptables -t filter -A FORWARD -p tcp --sport 80 -d
↳ 192.168.100.0/24 -m state --state RELATED,ESTABLISHED
↳ -j ACCEPT
```

---

<sup>6</sup>Non ho usato un secondo host anche in questa rete perché il carico di lavoro del mio portatile iniziava ad essere un pò elevato.

```
# https stateful
iptables -t filter -A FORWARD -p tcp --dport 443 -s
↳ 192.168.100.0/24 -m state --state
↳ NEW,RELATED,ESTABLISHED -j ACCEPT
iptables -t filter -A FORWARD -p tcp --sport 443 -d
↳ 192.168.100.0/24 -m state --state RELATED,ESTABLISHED
↳ -j ACCEPT
```

Sia i client sia il server sono stati configurati come indicato nel capitolo precedente. Qui di seguito sono mostrati i file di configurazione, a cominciare dal server, poi client in net50, infine quello in net200.

```
# server remoto
```

```
mode server
tls-server
```

```
proto tcp
```

```
dev-type tun
dev ovpn-server
```

```
persist-tun
persist-key
```

```
port 443
```

```
# VPN subnet
server 10.7.0.0 255.255.255.0
```

```
topology subnet
```

```
client-config-dir /etc/openvpn/server-single/ccd
```

```
# TLS configuration
remote-cert-eku "TLS Web Client Authentication"
tls-version-min 1.2
tls-cipher TLS-ECDHE-RSA-WITH-AES-256-GCM-SHA384
cipher AES-256-GCM
```

```
# chiavi e certificati
ca /etc/openvpn/certs/ca.crt
cert /etc/openvpn/server-single/certs/server-single.crt
key /etc/openvpn/server-single/certs/server-single.key
```

```
dh none
```

```
reneg-sec 1200
```

```
# log configuration
log /var/log/openvpn/server-single/openvpn.log
verb 4
status /var/log/openvpn/server-single/openvpn-status.log

script-security-level 2

client-connect /etc/openvpn/server-single/cs/client-up.sh
client-disconnect /etc/openvpn/server-single/cs/client-down.sh

keepalive 10 60
```

Il contenuto dello script eseguito quando un client si connette:

```
#!/bin/bash

if [ "$common_name" == "client50" ]; then
    ip route add 192.168.50.0/24 via 10.7.0.1
    iptables -t nat -A POSTROUTING -d 192.168.50.0/24
    ↪ -j MASQUERADE
fi

if [ "$common_name" == "client200" ]; then
    ip route add 192.168.200.0/24 via 10.7.0.1
    iptables -t nat -A POSTROUTING -d 192.168.200.0/24
    ↪ -j MASQUERADE
fi
```

Mentre quello eseguito quando i client si disconnettono:

```
#!/bin/bash

if [ "$common_name" == "client50" ]; then
    ip route del 192.168.50.0/24 via 10.7.0.1
    iptables -t nat -D POSTROUTING -d 192.168.50.0/24
    ↪ -j MASQUERADE
fi

if [ "$common_name" == "client200" ]; then
    ip route del 192.168.200.0/24 via 10.7.0.1
    iptables -t nat -D POSTROUTING -d 192.168.200.0/24
    ↪ -j MASQUERADE
fi
```

Il file `/etc/openvpn/server-single/ccd/client50` contiene la seguente linea:

```
iroute 192.168.50.0 255.255.255.0
```

Parallelamente, ecco come si presenta `/etc/openvpn/server-single/ccd/client200`:

```
iroute 192.168.200.0 255.255.255.0
```

Di seguito invece file di configurazione del client VPN in `net50`, il cui Common Name è `client50`.

```
# /etc/openvpn/client50.conf
```

```
client
```

```
proto tcp
```

```
remote ec2-52-15-172-187.us-east-2.compute.amazonaws.com 443
```

```
dev-type tun
```

```
dev ovpn-client50
```

```
persist-tun
```

```
persist-key
```

```
group nogroup
```

```
user nobody
```

```
# TLS configuration
```

```
remote-cert-eku "TLS Web Server Authentication"
```

```
tls-version-min 1.2
```

```
tls-cipher TLS-ECDHE-RSA-WITH-AES-256-GCM-SHA384
```

```
cipher AES-256-GCM
```

```
# chiavi e certificati
```

```
ca /etc/openvpn/certs/ca.crt
```

```
cert /etc/openvpn/client50/certs/client50.crt
```

```
key /etc/openvpn/client50/certs/client50.key
```

```
# log configuration
```

```
log /var/log/openvpn/client50/openvpn.log
```

```
verb 4
```

```
status /var/log/openvpn/client50/openvpn-status.log
```

Infine, il file del client in `net200`, dal Common Name: `client200`.

```
# /etc/openvpn/client200.conf
```

```
client
```

```
proto tcp
```

```

remote ec2-52-15-172-187.us-east-2.compute.amazonaws.com 443

dev-type tun
dev ovpn-client200

persist-tun
persist-key

group nogroup
user nobody

# TLS configuration
remote-cert-eku "TLS Web Server Authentication"
tls-version-min 1.2
tls-cipher TLS-ECDHE-RSA-WITH-AES-256-GCM-SHA384
cipher AES-256-GCM

# chiavi e certificati
ca /etc/openvpn/certs/ca.crt
cert /etc/openvpn/client200/certs/client200.crt
key /etc/openvpn/client200/certs/client200.key

# log configuration
log /var/log/openvpn/client200/openvpn.log
verb 4
status /var/log/openvpn/client200/openvpn-status.log

```

Una volta messa a punto la connessione, l'obiettivo del test era riuscire a visualizzare le pagine web offerte dai target dal *Docker host*. Mentre con SoftEther non ero nemmeno riuscito a completare la connessione, lo screenshot qui sotto mostrano che la mia soluzione funziona. Si tratta della mia shell connessa in SSH al *Docker host* sul quale è in esecuzione un container Alpine. Come detto prima, i tre host che si vogliono raggiungere sono mappati su:

- 192.168.50.30 (net50)
- 192.168.50.40 (net50)
- 192.168.200.30 (net200)

Per far sì che il *Docker host* possa raggiungere i target, è necessario configurare la routing table del suo kernel per instradare i pacchetti destinati ai target al VPN server. L'indirizzo IP di tale server è 172.31.40.221, per cui si eseguono i seguenti due comandi sul *Docker host*:

```

ip route add 192.168.50.0/24 via 172.31.40.221
ip route add 192.168.200.0/24 via 172.31.40.221

```

Ecco quindi il risultato: A titolo di documentazione, si mostra anche l'output del comando curl eseguito sui tre host target.



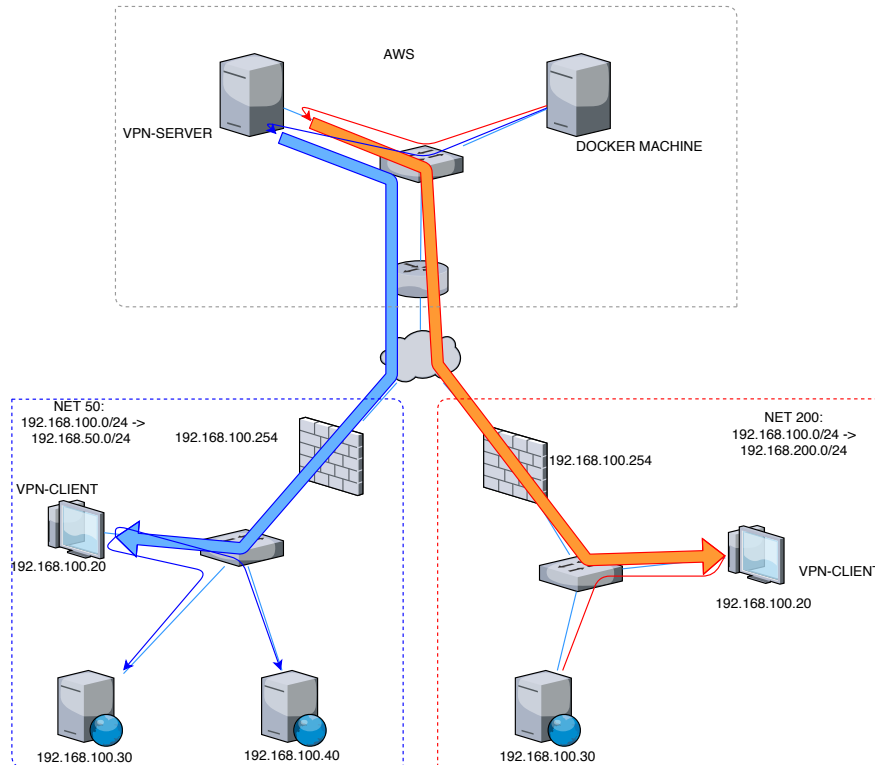
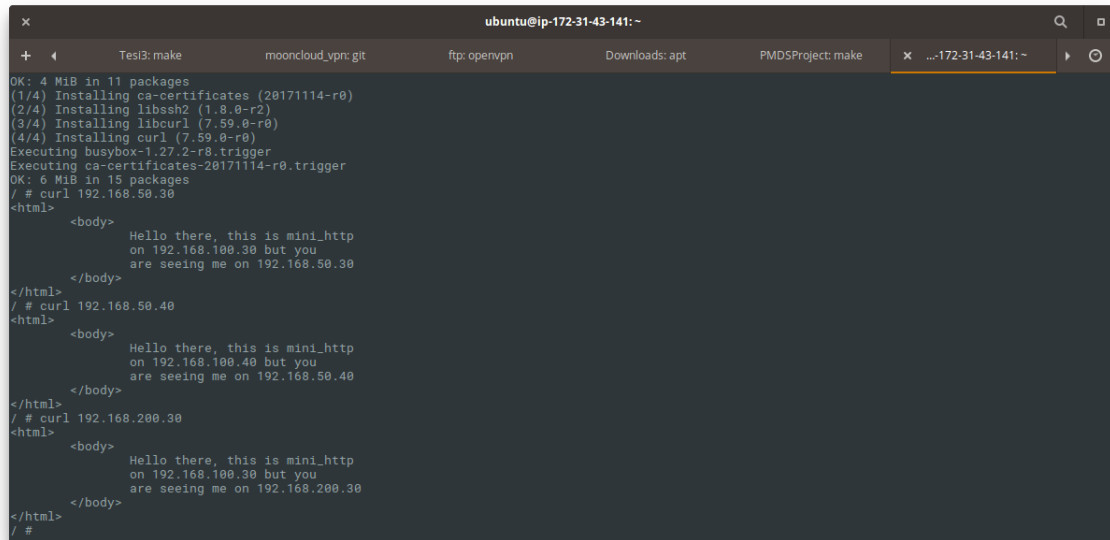


Figura 3.3: Test 1 per OpenVPN

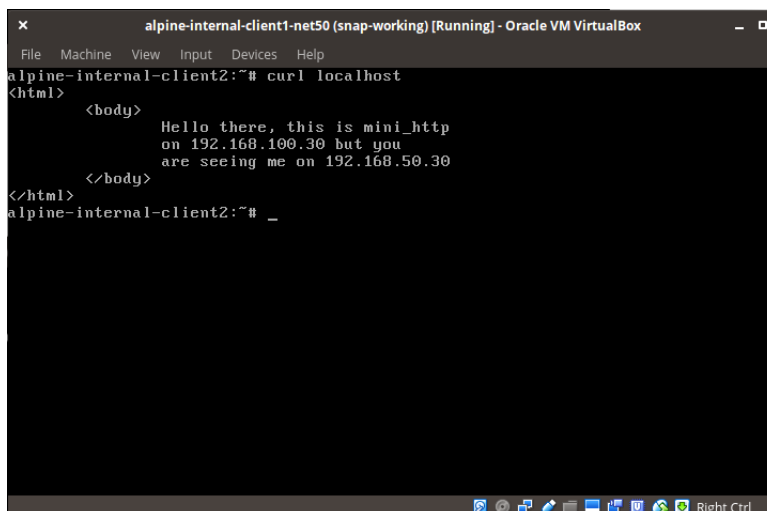
### 3.7.2 Test 2

Il secondo test è stato eseguito interamente su macchine virtuali messe a disposizione dal SESAR Lab. Per brevità, non si riporta di nuovo la configurazione dettagliata del test, poiché a parte gli indirizzi IP che in sé sono diversi, il setup è stato di fatto lo stesso. L'unica differenza è che si è usata una sola rete target anziché due. L'obiettivo era testare una connessione SSH dalla *Docker machine* nella stessa rete del VPN server verso un host nella rete target (passando ovviamente per il collegamento VPN). Il test ha dato esito positivo, ed è mostrato in figura 3.7.2.



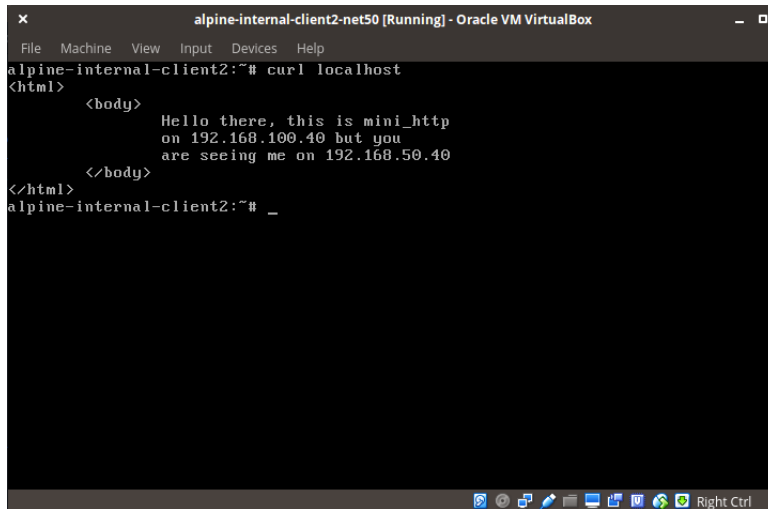
```
ubuntu@ip-172-31-43-141: ~  
+ Tesi3: make mooncloud_vpn: git ftp: openvpn Downloads: apt PMDSPProject: make x ...-172-31-43-141: ~  
OK: 4 MiB in 11 packages  
(1/4) Installing ca-certificates (20171114-r0)  
(2/4) Installing libssh2 (1.8.0-r2)  
(3/4) Installing libcurl (7.59.0-r0)  
(4/4) Installing curl (7.59.0-r0)  
Executing busybox-1.27.2-r8.trigger  
Executing ca-certificates-20171114-r0.trigger  
OK: 6 MiB in 15 packages  
/ # curl 192.168.50.30  
<html>  
  <body>  
    Hello there, this is mini_http  
    on 192.168.100.30 but you  
    are seeing me on 192.168.50.30  
  </body>  
</html>  
/ # curl 192.168.50.40  
<html>  
  <body>  
    Hello there, this is mini_http  
    on 192.168.100.40 but you  
    are seeing me on 192.168.50.40  
  </body>  
</html>  
/ # curl 192.168.200.30  
<html>  
  <body>  
    Hello there, this is mini_http  
    on 192.168.100.30 but you  
    are seeing me on 192.168.200.30  
  </body>  
</html>  
/ #
```

Figura 3.4: Il risultato dell'esecuzione del test

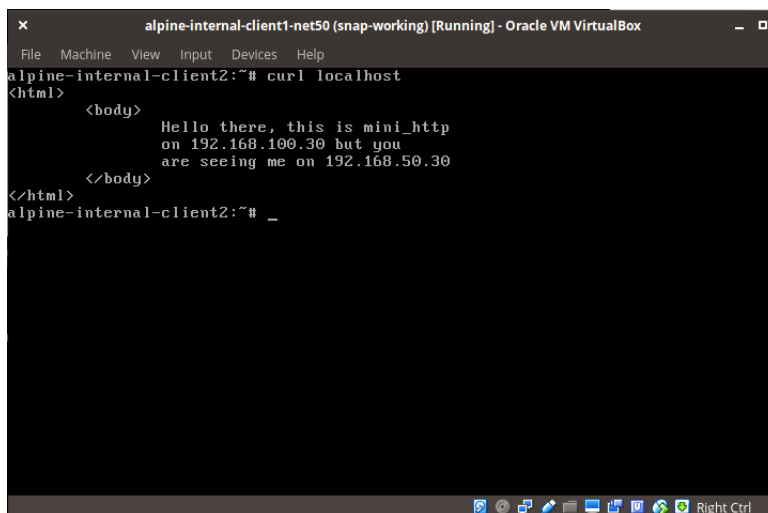


```
alpine-internal-client1-net50 (snap-working) [Running] - Oracle VM VirtualBox  
File Machine View Input Devices Help  
alpine-internal-client2:~# curl localhost  
<html>  
  <body>  
    Hello there, this is mini_http  
    on 192.168.100.30 but you  
    are seeing me on 192.168.50.30  
  </body>  
</html>  
alpine-internal-client2:~# _
```

Figura 3.5: L'output di curl su 192.168.50.30 (mappato)



```
alpine-internal-client2-net50 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
alpine-internal-client2:~# curl localhost
<html>
  <body>
    Hello there, this is mini_http
    on 192.168.100.40 but you
    are seeing me on 192.168.50.40
  </body>
</html>
alpine-internal-client2:~# _
```

Figura 3.6: L'output di `curl` su `192.168.50.40` (mappato)

```
alpine-internal-client1-net50 (snap-working) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
alpine-internal-client2:~# curl localhost
<html>
  <body>
    Hello there, this is mini_http
    on 192.168.100.30 but you
    are seeing me on 192.168.50.30
  </body>
</html>
alpine-internal-client2:~# _
```

Figura 3.7: L'output di `curl` su `192.168.200.30` (mappato)

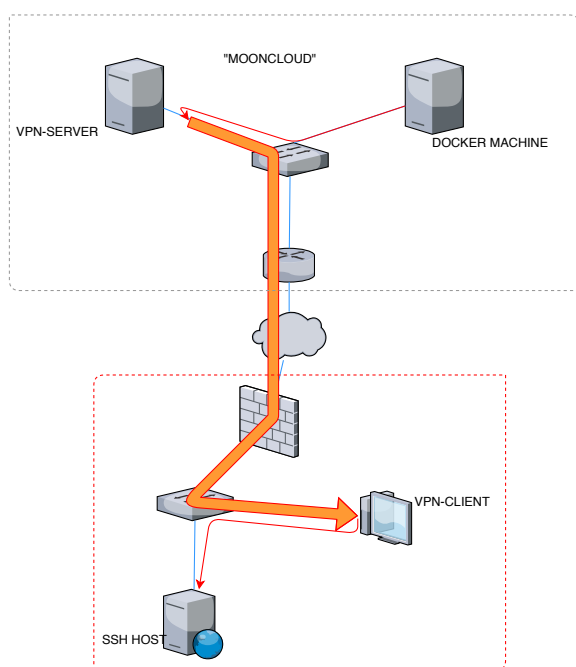


Figura 3.8: Test 2 per OpenVPN

# Capitolo 4

## nftables

`nftables` è la soluzione di nuova generazione per il firewalling in Linux, il cui intento è quello di sostituire `iptables`. In questo capitolo si analizza come funziona `nftables`, come e perché è migliore del suo predecessore, e soprattutto, come si è passati da `iptables` a `nftables` per realizzare il mapping delle reti lato client.

### 4.1 Prerequisiti

Prima di spiegare `nftables` si procede ad una breve introduzione di `iptables`, questo consentirà di comprendere meglio come mai `nftables` sia superiore.

**netfilter** *netfilter* è il componente del kernel Linux che “enables packet filtering, network address [and port] translation (NA[P]T) and other packet mangling”, il quale viene utilizzato mediante il popolare programma `iptables` (ed *ip6tables*, *arptables*, *ebtables*). Si tratta di uno strumento molto potente, che consente a qualsiasi dispositivo supportato dal kernel Linux di diventare un firewall (o di compiere praticamente qualsiasi altra operazione sui pacchetti in transito).

Sia `netfilter` sia `iptables` si scrivono senza maiuscole.

`netfilter` è composto da una serie di *hook* a cui è possibile registrare delle *callback* che sono chiamate ogni volta che un pacchetto passa per un certo hook.

Un pacchetto che entra in contatto con lo stack di rete di Linux transita per diverse parti di esso in diversi momenti; un hook non è un altro che un *punto* dello stack in cui il pacchetto passa. L'intero stack di rete offre numerosi hook, e non tutti sono disponibili in `netfilter`. In particolare, gli hook disponibili per quest'ultimo sottosistema sono:

- `NF_IP_PREROUTING`, raggiunto da tutti i pacchetti appena entrano nello stack di rete, la decisione di routing deve ancora essere presa.
- `NF_IP_LOCAL`, per i pacchetti dopo che sono stati *routed* dal sistema e destinati al sistema locale.
- `NF_IP_FORWARD`: dopo che si è presa la decisione di routing, per pacchetti *in transito* ad un altro sistema.

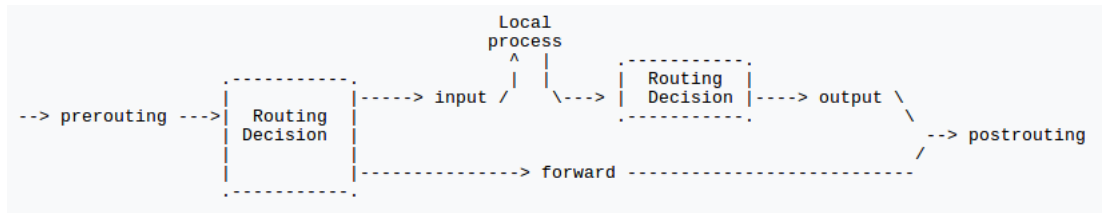


Figura 4.1: Gli hook di Netfilter *classici*. Tutti i pacchetti in arrivo passano per PREROUTING, viceversa quelli in uscita devono passare per forza per POSTROUTING.

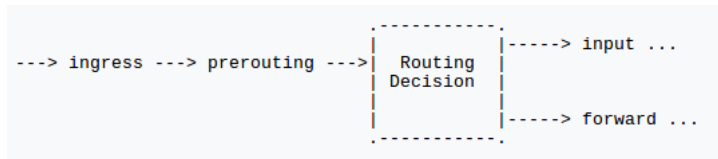


Figura 4.2: Hook INGRESS di Netfilter. E' l'hook che viene prima di tutti gli altri.

- `NF_IP_LOCAL_OUT`: pacchetti creati localmente e destinati ad altri sistemi (quindi viene dopo la decisione di routing).
- `NF_IP_POSTROUTING`, raggiunto da tutto i pacchetti dopo che sono stati *routed* e stanno per uscire dal sistema.
- `NF_IP_INGRESS` è un hook disponibile nelle versioni più recenti di Linux, avviene prima di `NF_IP_PREROUTING` per pacchetti in ingresso.

**iptables** iptables è uno dei programmi userspace usati per registrare degli hook in netfilter e specificare quale azioni devono essere eseguite. Ciò che tipicamente si scrive mediante iptables sono delle *regole*, esse sono organizzate in *tabelle*, ed a loro sono organizzate in *chain*. Esistono delle tabelle predefinite:

- `filter`: utilizzata per definire regole di firewalling
- `mangle`: utilizzata per *packet alteration*
- `nat`: utilizzata per gestire i protocolli NAT/PAT
- `raw`: utilizzata principalmente per creare esenzione dal tracking delle connessione.
- `security`: utilizzata in distribuzioni SELinux

E' possibile definire delle tabelle ulteriori. Le chain predefinite sono:

- `FORWARD` per pacchetti che vengono *routed* attraverso il sistema
- `INPUT` per i pacchetti ricevuti e destinati al sistema locale
- `OUTPUT`: pacchetti generati dal sistema in uscita da esso

- PREROUTING per operare su pacchetti appena arrivano nello stack di rete
- POSTROUTING per lavorare su pacchetti quando stanno per uscire dal sistema.

Non tutte le tabelle hanno a disposizione tutte le chain, ad esempio nella tabella di nat sono disponibili sono PREROUTING e POSTROUTING, mentre in filter è possibile usare INPUT, FORWARD, OUTPUT.

Come si può dedurre dal nome delle chain, esse corrispondono agli hook di netfilter:

Hook	Chain
NF_IP_PREROUTING	PREROUTING
NF_IP_LOCAL	INPUT
NF_IP_FORWARD	FORWARD
NF_IP_LOCAL_OUT	OUTPUT
NF_IP_POSTROUTING	POSTROUTING

Tabella 4.1: La corrispondenza tra gli hook di netfilter e le chain di iptables; si può notare come non sia possibile usare l'hook NF\_IP\_INGRESS perché non disponibile in iptables.

**Regole** Le regole iptables sono composte da due parti: una parte detta di *match* nella quale si specificano i criteri per cui un pacchetto matcha o meno una regola, ed una seconda di *target* (indicato con l'opzione `-j`) che specifica quale azione compiere per quel pacchetto se vi è matching. Alcune tra i target disponibili sono:

- ACCEPT per accettare il pacchetto
- DROP per dropare il pacchetto, eliminandolo ed impedendo che prosegua nel suo attraversamento del kernel
- REJECT per dropare il pacchetto inviando un pacchetto ICMP al mittente
- DNAT per applicare il protocollo NAT/PAT cambiando l'indirizzo IP/porta destinazione
- SNAT per applicare il protocollo NAT/PAT all'indirizzo sorgente.

Un'importante funzionalità offerta da netfilter è il tracking della connessione in tabelle di stato (nota come *Connection Tracking*), che è la base per il NAT e statefull firewalling. Infine, sono disponibili numerose estensioni sia per matching sia per target; Ad esempio, esiste una versione di iptables (*iptables L7*) che aggiunge comprensione anche dei protocolli applicativi. iptables viene usato per il protocollo IP versione 4 (e protocolli di livello superiore che si appoggiano ad IPv4), se si vuole lavorare con protocolli di livello 3 diversi, o protocolli di livello inferiore vi sono:

- arptables: manipolazione del protocollo ARP

- `ebtables` per lavorare con protocolli di livello 2 (Ethernet)
- `ip6tables` per pacchetti che usano IPv6 come protocollo di rete.

## 4.2 Criticità

Ogni volta che si invoca un comando in userspace per modificare il set di regole, il programma in userspace ottiene dal kernel il *blob* di *tutte* le regole, vi applica le modifiche, invia la versione modificata al kernel. Questo porta a numerosi problemi, tra cui il fatto che più il numero di regole aumenta, più la modifica delle regole è lenta. In sostanza, ciò che si vorrebbe è un'aggiunta *atomica* delle regole, che sia più veloce ed affidabile.

Ci sono vari altri problemi con `iptables` che negli anni sono emersi, e la maggior parte di essi non sono attualmente risolvibili se non con un grosso cambio a livello progettuale, cambio che non è possibile perché richiederebbe dei *breaking changes*, e vista la vasta diffusione di `iptables`, questo non è possibile. Alcuni dei problemi sono:

**Duplicazione del codice** Per ogni protocollo gestito da `iptables` (o sue controparti), occorre che tale programma sappia gestire nel dettaglio tale protocollo, in modo che possa creare le strutture dati necessarie per il file da inserire nel kernel.

Il kernel stesso deve infine poter gestire tale protocollo perché deve compiere le operazioni richieste.

Ciò significa che *per ogni nuovo protocollo*, esso deve essere supportato sia in userspace sia in kernel-space.

**Target multipli** Si supponga che si voglia droppare un pacchetto e dropparlo, in una sola regola non è possibile, è necessario, ad esempio, creare una nuova tabella.

**Matching avanzato** `iptables` supporta matching di fatto controllando l'header del protocollo di trasporto e l'header IP, tuttavia non è possibile scrivere in una *regola* il fatto che il traffico TCP sulla porta 8 e 443 sono consentite: occorre scrivere almeno due regole. Questo diventa particolarmente tedioso quando il numero di regole aumenta molto (esistono estensioni come `multiport` per il matching di porte multiple, ma ne supporta fino a 15).

**Multipli programmi** Un'altra criticità è il fatto che vi sono 4 programmi userspace diversi, particolarmente problematica è la gestione separata di IPv4 e IPv6.

**Scalabilità** Come si comporta `iptables/netfilter` quando vi sono davvero molte regole (centinaia o più) regole da gestire ed il throughput è elevatissimo? La risposta è purtroppo semplice, `iptables` non scala, a causa del suo design che prevede l'attraversamento delle varie regole.

Questi problemi hanno portato gli sviluppatori di `netfilter` ad introdurre una nuova soluzione, una soluzione molto più avanzata.



## 4.3 nftables

La soluzione a questi problemi c'è e si chiama `nftables`. Esso utilizza un approccio completamente diverso rispetto ai predecessori: una *in-kernel virtual machine* per il packet matching, concettualmente simile a BPF – Berkeley Packet Filter. Vi è un unico programma in userspace chiamato `nft`, il quale traduce in bytecode i comandi inseriti dall'utente e li *inietta* nel kernel (mediante `Netlink`<sup>1</sup>). Ecco un elenco delle principali novità:

- Il kernel non ha più conoscenza specifica dei protocolli di rete: fare match con una porta TCP o un flag dell'header Ethernet si traduce in semplici istruzioni come: *carica x byte del pacchetto nel registro z a partire dall'offset y*. Questo significa che è molto facile estendere `nftables` con nuovi protocolli, e che il codice del kernel è molto più semplice.
- Non ci sono tabelle e chain di default.
- Ogni regola consente di più target in una sola volta.
- Il programma `nft` diventa l'unico programma per la gestione delle regole, rimpiazzando `iptables`, `ip6tables`, `arptables`, `ebtables`.
- L'aggiunta, l'aggiornamento, la cancellazione ed ogni altra operazione sulle regole viene ora fatta in maniera atomica. Ciò significa che ogni volta che si fa un'operazione, anche su più regole, essa va a buon fine (cioè i suoi effetti sono applicati) solo se ogni singola operazione va a buon fine.
- Possibilità di gestire contemporaneamente IPv4 ed IPv6 in una singola tabella/chain.
- Supporto a strutture dati avanzate basate su `set` che consentono un matching/targeting molto veloce.
- La sintassi delle regole cambia e diventa più simile ad una frase.

La maggior parte dell'*intelligenza* è spostata su `nft`, il quale compila le regole ricevute in input e le passa al kernel, esso è anche responsabile di decompilare il bytecode presente nel kernel per presentarlo all'utente in un formato human-readable.

Vi sono due tipi di sintassi supportate: il primo è una sintassi assimilabile a quella di `iptables` dove si scrive una regola per volta. Il secondo tipo è invece una sintassi *strutturata* più simile a sintassi usati in specifici file di configurazione. Saranno mostrati esempi di entrambe.

---

<sup>1</sup>`Netlink` è un meccanismo IPC – Inter-Process Communication del kernel Linux che consente di comandare e ottenere informazioni riguardo lo stack di rete, è una interfaccia che offre API basate su socket.

**Tabelle e chain** Si è detto che non esistono tabelle di default, quindi è spetta all'utente il compito di crearle. In nftables, le tabelle sono di una certa *famiglia*:

- `ip` è la famiglia per operare su pacchetti dal livello 3 in su e che usano IPv4 come protocollo di rete; è la famiglia di default se non è specificata.
- `ip6` è la famiglia per operare con pacchetti di protocolli che utilizzano il protocollo IPv6.
- `arp` viene utilizzata per processare pacchetti ARP.
- `inet` è la famiglia che consente di lavorare con pacchetti sia IPv4 sia IPv6.
- `bridge` è usata per lavorare con frame Ethernet.
- `netdev` consente di *attaccarsi* direttamente ad una interfaccia di rete.

Le tabelle sono solo dei contenitori vuoti, il cui nome non ha alcuna semantica (rispetto ad iptables). Comunque, la scelta della famiglia influenza quali hook possono essere utilizzati (es: l'hook `ingress` è disponibile solo per `netdev`).

Le tabelle a loro volta racchiudono delle chain, ed anche in questo caso non vi sono delle chain predefinite ma occorre crearle. Quando si crea una chain occorre specificare almeno tre cose:

- tipo; le scelte possibili sono:
  - `filter` da utilizzare quando si vuole fare del filtraggio
  - `nat` per utilizzare le funzionalità NAT
  - `route` per fare generico *packet mangling* che non sia NAT
- hook, in questo caso le scelte possibili sono gli hook già visti, cambia solo il fatto che vengono scritti in minuscolo.
- priorità: espressa con un numero, si tratta di una priorità *inversa*, più è bassa più il suo valore è alto; indica l'ordine con cui alcune operazioni sono svolte. Un *buon* valore è 0.

Quando si crea una nuova chain è possibile specificare la policy di default, cioè l'azione da compiere se non si matcha nessun'altra regola; il valore di default è `accept` (accettazione il pacchetto), altri valori possibili sono: `drop`, `queue`, `continue`, `return`.

### 4.3.1 Set e altre strutture dati

nftables introduce tre strutture dati pensate indirizzando i problemi presenti in iptables (e sue controparti):

- matching limitato, per cui è problematico specificare più di una porta o più di un singolo indirizzo IP in una regola; il che porta a:
- elevato numero di regole a causa della suddetta limitazione.

E' possibile dichiarare queste strutture dati in due modi:

- *anonymous*: vengono create ed istanziate all'interno di una regola e sono immutabili, non è possibile modificarle dopo la loro creazione.
- *Named*: sono vere e proprie variabili che vivono nello *scope* di una tabella in cui sono definite. Una volta create è possibile modificarle in seguito. Per riferirsi ad una variabile, occorre preporre @ al nome della variabile.

L'idea principale che sta dietro le strutture dati di nftables è quella di poter ridurre drasticamente il numero di regole necessarie. Migliorie che si sono dimostrate efficaci; per una dettagliata comparazione delle performance tra iptables e nftables si veda [6].

## Set

La struttura dati avanzata di nftables si chiama *set*, ed è implementata utilizzando alberi rosso-neri, tabelle di hash, bitmap. Un set è, come dice il nome, un insieme di elementi di un certo *tipo*, esso può essere un tipo *base* predefinito come `ipv4_addr` o `inet_service` (porta protocollo di trasporto), oppure un tipo complesso *composto* da altri tipi, ad esempio un nuovo tipo `ipv4_port`. Quando si crea una nuova struttura dati è possibile specificare vari parametri, tra cui una *policy*: per cosa deve essere ottimizzata la struttura? I valori possibili sono *performance* o *memory*. Un esempio è il seguente.

I set sono utilizzati per:

- fare match su più elementi in un'unica regola, ad esempio, se si vogliono bloccare connessioni da 10 indirizzi IP diversi, è possibile definire un set che contenga questi 10 IP, anziché scrivere 10 regole diverse. Si consideri che i set sono molto performanti.
- Essere la struttura dati base su cui si creano ulteriori strutture dati.

## Map

Le map sono costruite sui set, infatti possono essere visti come dei set in cui ciascun elemento è composto a sua volta da due elementi: il primo di essi è la *chiave*, il secondo è il *valore* associato alla chiave. Essi si comportano effettivamente come una mappa poiché definiscono un *mappaggio* tra la chiave ed il valore, e sono utilizzati in contesti in cui si *trasforma* un input in un certo output secondo il mappaggio definito in questa struttura dati. Si può forse intuire l'utilizzo delle map nel mappaggio delle reti...

In particolare, le map sono molto utili con target di tipo `dnat` ed `snat`. Si supponga che si voglia modificare l'indirizzo IP destinazione da `192.168.1.0/24` nel corrispondente indirizzo in `192.168.100.0/24` (mantenendo lo stesso offset) in `prerouting`, ed il viceversa in `postrouting`. Si è già visto che farlo in iptables richiede 508 regole, 254 per ogni indirizzo in `prerouting` e 254 per ogni indirizzo in `postrouting`. Con nftables lo si può fare in due regole e due mappe! L'esempio qui riportato mostra come farlo, ed è esattamente ciò che è stato fatto nella mia soluzione finale di mappaggio delle reti (si veda la sezione successiva per maggiori dettagli).

## Dictionary

La struttura dati *dictionary*, a volte nota anche come *vmap* – *verdict map*, è concettualmente molto simile alle map, tuttavia gli elementi di questo insieme sono nelle forma: `key: verdict`, dove `Verdict` indica l'azione da applicare nel momento in cui si la chiave viene matchata.

**Esempi** Di seguito si riportano degli esempi che mostrano il funzionamento di nftables. Si noti in particolare come, nelle ultime quattro regole l'utilizzo dei set consente di specificare match multipli in un'unica regola. Questo non significa solo una comodità per chi scrive le regole, infatti i set sono molto performanti e sono le strutture dati base su cui nftables costruisce altre strutture più avanzate, che verranno descritte in seguito.

Supponendo che ciò che si vede listato qui sotto sia il contenuto di un file, si può notare lo shebang all'inizio, in modo che sia eseguito da nft nella modalità nativa di shell che offre: tutte queste regole sono inserite atomicamente in una transazione. Invocando questi comandi in una shell normale invece si perderebbe l'atomicità perché ciascun comando è una invocazione separata al programma nft (si noti che qui tutte le regole non sono appunto prefisse da "nft").

```
#!/usr/sbin/nft -f

# creazione della tabella, sia IPv4 sia IPv6
# chiamata 'filter'
add table inet filter

# aggiunta chain 'input'
add chain inet filter input { type filter hook input priority 0; \
policy drop;}

# chain 'output'
add chain inet filter output { type filter hook output priority 0; \
policy drop;}

# consentire il traffico in entrata sulla porta 22 in modo statefull
add rule inet filter input tcp dport 22 ct state\
new, related, established accept

# consentire le risposte
add rule inet filter output tcp sport 22 ct state\
related, established accept

# consentire tutto il traffico in uscita sulle
# porte 80 e 443 usando un set
add rule inet output tcp dport { 80, 443 } ct state \
new, related, established accept

# consentire le risposte
```

```

add rule input tcp sport { 80, 443 } ct state \
related, established accept

# creazione di un set chiamato 'allowed'

add set filter allowed { type: ipv4_addr }
add element filter allowed {
    192.168.1.20,
    192.168.2.20,
    192.168.100.1,
    192.168.200.254,
    10.7.0.20
}

# consentire tutte le connessioni dagli indirizzi IP
# elencati sulla porta 8080
# nota: per accedere ad una named set si utilizza '@'
add rule input ip saddr @allowed tcp dport 8080 \
ct state {new, related, established } accept
# consentire le risposte
add rule output ip daddr @allowed tcp sport 8080 \
ct state {related, established }accept

```

E' possibile ottenere lo stesso risultato usando una sintassi *strutturata*:

```

#!/usr/sbin/nft -f
table inet filter {
    set allowed {
        type ipv4_addr
        elements = {
            192.168.1.20,
            192.168.2.20,
            192.168.100.1,
            192.168.200.254,
            10.7.0.20
        }
    }

    chain input {
        type inet hook input priority 0; policy drop;
        tcp dport 22 ct state new, related, established accept
        tcp sport {80, 443} ct state {related, established accept
        ip saddr @allowed tcp dport 8080 ct state new, related,
            established accept
    }

    chain output {
        type inet hook output priority 0; policy drop;
    }
}

```

```

    tcp sport 22 ct state related, established accept
    tcp dport {80, 443} ct state new, related, established accept
    ip addr @allowed tcp sport 8080 ct state elated,
        established accept
}
}

```

Per ottenere la lista delle regole attive, è disponibile il comando `nft list ruleset`, che produce un output nello stesso formato *strutturato* appena mostrato.

L'esempio seguente si concentra invece sull'utilizzo dei dizionari. Si supponga di dover gestire una serie di regole relative alla porta 22:

- consentito dalle reti 192.168.100.0/24, 192.168.101.0/24, 192.168.200.0/24
- proveniente dalla rete 192.168.1.0/24 occorre REJECT
- tutto il resto non è consentito.

```

iptables -t filter -A INPUT -p tcp --dport 22 -s
↳ 192.168.100.0/24 -j ACCEPT
iptables -t filter -A INPUT -p tcp --dport 22 -s
↳ 192.168.101.0/24 -j ACCEPT
iptables -t filter -A INPUT -p tcp --dport 22 -s
↳ 192.168.200.0/24 -j ACCEPT

iptables -t filter -A INPUT -p tcp --dport 22 -s
↳ 192.168.1.0/24 -j REJECT
iptables -t filter -A INPUT -p tcp --dport 22 -j DROP

# per brevità le regole in OUTPUT sono omesse

```

Mentre con `nftables`:

```

#!/usr/sbin/nft -f

# supponendo tabelle e chain già create
# aggiunta di 'dict' alla tabella filter
add map filter dict { type ipv4_addr: verdict; }

add element filter dict { 192.168.100.0/24: accept, \
    192.168.101.0/24: accept, \
    192.168.200.0/24: accept, \
    192.168.1.0/24: reject }

add rule filter input tcp dport 22 ip saddr vmap @dict
add rule filter input tcp dport 22 drop

```

## 4.4 Come è stato usato

Si supponga di voler gestire un VPN client che gestisce una singola rete classe C: con il vecchio approccio di iptables, come già detto, sono necessarie esattamente 19 regole:

- 254 per trasformare gli indirizzi IP mappati negli indirizzi IP originali
- 254 regole per trasformare gli IP originali negli IP mappati
- 1 per gestire il *NAT al contrario*

nftables è stato adottato prima che si conducessero veri test sul campo in cui si potessero misurare le performance della soluzione basata su iptables, quindi non è stato possibile determinare quale fosse la scalabilità effettiva della vecchia proposta, certo è che con nftables *si riduce il numero di regole da 19 a 3!* Non solo, le nuove regole sono create utilizzando strutture dati performanti, e sono e rimangono sempre 3, a prescindere che si il client VPN rimappi una o  $n$  reti.

In particolare si è sfruttata la struttura dati *map* proprio come è stata descritta nella sezione precedente. Si definiscono due *named maps*, una per la gestione del PREROUTING chiamata *remapping*, ed una per il POSTROUTING chiamata *mapping*. La ragione per cui si sono scelte delle *named maps* è che consentono di essere modificate in seguito, per cui per qualsiasi eventualità un operatore può agire direttamente su esse lasciando intatte le regole. Di seguito un esempio, supponendo la seguente configurazione:

- OTN: 192.168.100.0/24
- MTN: 192.168.1.0/24
- subnet VPN: 10.7.0.0/24

Il file di script per nftables viene creato usando la sintassi *strutturata*, di seguito se ne riporta, per brevità, solo un estratto.

```
#!/usr/sbin/nft -f
table ip ovpn_nat {

    map mapping {
        type ipv4_addr: ipv4_addr
        elements = {
            192.168.1.1: 192.168.100.1,
            192.168.2.1: 192.168.100.2,
            192.168.1.3: 192.168.100.3,
            192.168.1.4: 192.168.100.4,
            192.168.1.5: 192.168.100.5,
            192.168.1.6: 192.168.100.6,
            192.168.1.8: 192.168.100.8,
            192.168.1.9: 192.168.100.9,
        }
    }
}
```

```
map remapping {
    type ipv4_addr: ipv4_addr
    elements = {
        192.168.100.1: 192.168.1.1,
        192.168.100.2: 192.168.1.2,
        192.168.100.3: 192.168.1.3,
        192.168.100.4: 192.168.1.4,
        192.168.100.5: 192.168.1.5,
        192.168.100.6: 192.168.1.6,
        192.168.100.7: 192.168.1.7,
        192.168.100.8: 192.168.1.8,
        192.168.100.9: 192.168.1.9
    }
}

chain prerouting {
    type nat hook prerouting priority 0; policy accept;
    ip saddr 10.7.0.0/24 dnat to ip addr map @remapping
}

chain postrouting {
    type nat hook postrouting priority 0; policy accept;
    ip saddr 10.7.0.0/24 ip daddr 192.168.100.0/24 masquerade
    ip daddr 10.7.0.0/24 snat to ip saddr map @mapping
}
}
```



# Capitolo 5

## Sicurezza della VPN

Questo capitolo è demandato ad un'analisi della sicurezza della soluzione proposta. La prima sezione discute le proprietà di sicurezza di OpenVPN e delle configurazioni che sono state adottate per renderla sicura il più possibile. La sezione successiva descrive come si protegge la rete MoonCloud, visto che i VPN server sono dei nuovi punti di ingresso. L'ultima sezione descrive alcuni scenari di attacchi e come siano stati mitigati.

### 5.1 Sicurezza di OpenVPN

#### 5.1.1 Protocollo

Come detto ampiamente in precedenza, OpenVPN si compone di due canali di comunicazione:

- `Control Channel` protetto con TLS ed usato per scambiarsi informazioni *di servizio*
- `Data Channel` utilizzato per l'effettivo scambio dei dati

Entrambi i due canali sono multiplexati su una singola connessione TCP o UDP. Poiché il protocollo TLS è stato progettato per funzionare su un livello di trasporto affidabile, questa affidabilità viene offerta da OpenVPN quando il protocollo di trasporto è UDP.

La sicurezza di OpenVPN può essere riassunto dalla seguente frase: “Use the IPsec ESP protocol for tunnel packet security, but then drop IKE in favor of SSL/TLS for session authentication”[7].

Di fatto, IPsec ESP qui corrisponde al `Data Channel`.

TLS è stato analizzato e studiato in dettaglio in tutti questi anni ed io non intendo aggiungere niente di nuovo rispetto alle analisi già fatte. La versione attuale di TLS è la 1.3, tuttavia quest'ultima versione non è completamente supportata nelle varie librerie SSL/TLS, in particolare non è ancora supportata dalla versione di OpenSSL utilizzata in OpenVPN. Le altre librerie supportate sono:

- LibreSSL (supportata non in maniera ufficiale)
- mbedTLS

TLS utilizza uno scambio di chiavi Diffie-Hellman (è possibile anche uno scambio basato su RSA), che se usato propriamente garantisce la proprietà di *Perfect Forward Secrecy*, ciò significa che se anche la chiave privata contenuta nel certificato fosse compromessa ed un avversario avesse registrato tutto il traffico comunque non riuscirebbe a decifrarlo, poiché le chiavi utilizzate per Diffie-Hellman vengono generate *al volo* ad ogni connessione, e quindi anche se l'avversario potesse vedere le chiavi pubbliche Diffie-Hellman, comunque dovrebbe essere in grado di rompere tale algoritmo, il che non è assolutamente fattibile. Le chiavi negoziate sono unidirezionali.

TLS offre diverse *cipher-suite*, in particolare quelle migliori sono basate su uno scambio di chiavi Diffie-Hellman effimero su curva ellittica (ECDHE) ed un cifrario AEAD - *Authenticated Encryption with Associated Data*. Diffie-Hellman effimero garantisce la proprietà di *Perfect Forward Secrecy*: se la chiave privata ECDSA o RSA relativa ad un certo certificato fosse in qualche modo compromessa e l'avversario avesse registrato tutto il traffico, non riuscirebbe comunque a scoprire la chiave segreta negoziata con Diffie-Hellman, da cui poi si derivano le chiavi simmetriche. Questo perché nella ciphersuite con Diffie-Hellman effimero la chiave pubblica contenuta nel certificato viene utilizzata *solo* per autenticare le chiavi pubbliche Diffie-Hellman generate *al volo* (ecco perché *effimere*). Quindi, anche avendo a disposizione la chiave privata, dovrebbe ancora essere in grado di rompere lo scambio Diffie-Hellman per ottenere il valore segreto negoziato.

*Authenticated Encryption with Associated Data* è una proprietà di sicurezza che garantisce confidenzialità, integrità ed autenticazione dei dati, oltre a autenticazione ed integrità di ulteriori dati *associati*, come ad esempio metadati relativi al protocollo che non devono essere anche essere segreti. Esistono algoritmi di cifratura che utilizzati nella maniera opportuna, tipicamente in combinazione con una funzione di MAC, garantiscono questa proprietà con un'unica chiave simmetrica (anziché con due chiavi, una per la cifratura ed una per il MAC). Tutti gli algoritmi utilizzati in TLS 1.3 garantiscono questa proprietà, essi sono: possibili, gli unici algoritmi ammessi in TLS 1.3 sono:

- AES-CCM, ovvero AES in modalità CTR combinato con CBC-MAC
- AES-GCM cioè AES in modalità CTR combinata con una funzione di hash nota come GHASH
- ChaCha20-Poly1305 che è la combinazione del cifrario a stream ChaCha20 con l'algoritmo di MAC Poly1305

Una volta completato il TLS handshake, i due partecipanti alla VPN (server ed un client) generano del materiale pseudo casuale e lo trasmettono sul *Control Channel*: tale materiale è utilizzato per le chiavi nel *Data Channel*. Il *Data Channel*, proprio come TLS, garantisce *Authenticated Encryption*.

OpenVPN non ha pubblicato una specifica formale del proprio protocollo, tuttavia esistono in letteratura diverse analisi che hanno provato a ricostruire, con buon successo, il suo funzionamento mediante *protocol state fuzzing* (si veda ad esempio [8] e [9]). Il risultato è stato che non sono state trovate particolari vulnerabilità nel protocollo, sebbene altri testing abbiano in passato scovato classiche vulnerabilità implementative, legate ad esempio ad una gestione scorretta della memoria (OpenVPN è scritto in C).

### 5.1.2 Configurazioni

In questa sottosezione si affrontano tutte le scelte in merito alla configurazione specifica di OpenVPN, comprese la scelta dei cifrari.

#### Crittografia

OpenVPN consente di specificare quali algoritmi crittografici utilizzare, in entrambi i canali. Fino a qualche versione fa, le ciphersuite del TLS vengono negoziate nel modo classico del TLS, mentre non era prevista alcuna modalità di negoziazione dei cifrari sul `Data Channel`, pertanto dovevano essere specificati nel file di configurazione. L'algoritmo di cifratura di default era `Blowfish` in modalità CBC combinato con `HMAC-SHA1`.

Nel 2016 due ricercatori francesi hanno pubblicato un attacco chiamato `SWEET32` ([10]), il quale sostanzialmente sfrutta l'utilizzo di cifrari con una dimensione del blocco *piccola* (minore di 128 bit) in modalità CBC. Registrando una sufficiente quantità di traffico (circa 700 GB), un attaccante riesce con successo a decifrare il plaintext, supponendo che esso conosca ne conosca una buona parte. Ad esempio, i ricercatori propongono l'esempio di una pagina servita su HTTPS che l'attaccante conosce, e ad esempio vuole scoprire il valore di un cookie. Questo attacco si applica a qualsiasi protocollo che utilizzi un cifrario con le caratteristiche sopra citate, e richiede molto traffico per poter aver successo, tuttavia nel caso di una VPN il fatto che vi sia molto traffico è la normalità.

Un esempio di cifrari vulnerabili è `Blowfish`, `DES`, `3DES`. Per tale ragione quindi OpenVPN è vulnerabile.

La prima contromisura che gli sviluppatori di OpenVPN hanno implementato è stata avvertire gli utenti mediante il file di log. Nelle ultime versioni è stata infine aggiunta la negoziazione dei cifrari, ed il cifrario di default non è più `Blowfish` ma `AES-GCM-256`, cioè AES in modalità GCM con chiavi a 256 bit. Poiché è un cifrario *AEAD*, garantisce autenticità ed integrità di per sé, per cui non occorre specificare anche un algoritmo MAC.

**Protezione del `Data Channel`** `AES-256-GCM` è il cifrario che ho scelto di utilizzare per la protezione del `Data Channel`. Si tratta di un algoritmo molto veloce, ed ovviamente molto sicuro. Sebbene sia il cifrario di default si è comunque scelto di specificare questo algoritmo in maniera esplicita. La direttiva è la seguente:

```
cipher AES-GCM-256
```

**Configurazione di TLS** Allo stesso modo, ho scelto di specificare in maniera esplicita quali siano le ciphersuite TLS, e sono stato molto stringente. Anche in questo caso l'unica possibile è quella basata su `AES-256-GCM`, cioè:

`TLS-ECDHE-ECDSA-AES-GCM-256-SHA384`. Si può notare che si utilizza una ciphersuite *ECDHE – Elliptic Curve Diffie-Hellman Ephemeral*, e come tale si garantisce *Perfect Forward Secrecy*. Si utilizza `ECDSA` in quanto i certificati utilizzati hanno chiavi pubbliche per questo algoritmo. Per specificare questo si usa la direttiva:

```
tls-cipher TLS-ECHDE-AES-265-GCM-SHA384
```

Esistono ciphersuite basate su ChaCha20-Poly1305, tuttavia di default non sono incluse nei file di configurazioni perché non sono ancora supportate nella versione di OpenSSL utilizzata da OpenVPN. E' supportata in alcune versioni di OpenVPN compilate con LibreSSL, tuttavia solo come ciphersuite di TLS, per il Data Channel ancora non vi è nessun supporto. Con *di default non è supportata* si intende il fatto che sia possibile configurare dinamicamente il mio microservizio per indicare se ChaCha20 sia supportato o meno.

La fase di handshake del TLS include anche la negoziazione della versione supportata da TLS, in modo che client e server utilizzino la versione più alta supportata da entrambi. In passato gli attaccanti riuscivano con successo a portare avanti il Downgrade Attack per il quale si riusciva a forzare un browser a negoziare una versione di TLS/SSL più bassa rispetto a quella effettivamente possibile. Per evitare alcun tipo di problema, si è scelto di specificare esplicitamente la versione nel seguente modo:

```
tls-version-min 1.2
```

Un'altra contromisura che viene consigliata per incrementare la sicurezza di OpenVPN è quella dello specificare il tipo di certificato che l'host si aspetta. E' infatti ragionevole presupporre che un server dovrebbe aspettarsi solo certificati demandati all'uso come client e viceversa. Nel file del server:

```
remote-cert-eku "TLS Web Client Authentication"
```

Mentre nel client:

```
remote-cert-eku "TLS Web Server Authentication"
```

E' infine possibile configurare anche direttive in cui si può specificare, ad esempio, come deve essere composto il certificato del remote, ad esempio quale debba essere il suo CommonName. Attualmente questa configurazione non è applicata, non si esclude che in futuro lo sia.

### Certificate Revocation List

Una *Certificate Revocation List* è una lista che contiene un elenco di certificati revocati (mediante i loro seriali), la data di revoca, ed eventualmente la ragione della loro revoca. Essa è emessa dalla CA che ha precedentemente emesso i certificati revocati, ed è firmata digitalmente dalla CA, in modo da attestarne l'autenticità e l'integrità.

OpenVPN supporta la verifica mediante CRL, tale verifica è ovviamente fatta lato server: quando un client si connette al server, OpenVPN legge tale CRL e, se il numero seriale del certificato del client è presente nella lista, la connessione con il client in questione viene terminata. A differenza dei file di configurazione in `/etc/` che vengono letti solo all'avvio dal software, la CRL viene riletta ad ogni nuova connessione, pertanto è possibile aggiornarla senza preoccuparsi di dover riavviare il server.

La gestione della CRL (aggiornamento e trasferimento verso i server) viene effettuata dal microservizio `MoonCloud_VPN`, vi è una API REST dedicata alla revoca/cancellazione dei client, uno dei suoi effetti è appunto la revoca del certificato di quel client.

La direttiva per utilizzare una CRL nel server è la seguente, assumendo che il file della CRL si trovi nella posizione indicata:

```
crl-verify /etc/openvpn/certs/crl.pem
```

### 5.1.3 Opzioni non ancora utilizzate

In questa sottosezione si mostrano alcune configurazioni che OpenVPN offre che non sono state utilizzate. Per ciascuna di esse vale la seguente considerazione: in futuro potranno essere adottate.

#### **tls-crypt e tls-auth**

Si tratta di ulteriori misure di sicurezza:

- **tls-crypt**: il Control Channel viene cifrato ulteriormente con una chiave precondivisa nota a tutti i partecipanti.
- **tls-auth**: il Control Channel viene protetto da un ulteriore layer di MAC usando una chiave statica che deve essere nota a tutti i partecipanti a priori.

Lo scopo principale è quello di mitigare attacchi DoS: se il server riceve un pacchetto con HMAC non valido, o che non riesce a decifrare, immediatamente dropa la connessione, anziché iniziare il TLS handshake (che eventualmente fallirebbe nel caso l'attaccante non avesse nessun certificato valido). Se questa chiave precondivisa fosse compromessa rimarrebbero comunque i certificati come forma effettiva di protezione, infatti i certificati sono e rimangono l'effettiva forma di autenticazione.

Si è scelto di non adottare questa soluzione per ora a causa della necessità di dover gestire in modo sicuro la chiave precondivisa: essa deve essere nota a tutti i partecipanti alla VPN. Nel caso in cui essa fosse compromessa occorrerebbe cambiare *tutte* le chiavi, poiché sebbene rimanga la protezione dei certificati, una chiave compromessa non garantirebbe nessun livello di protezione, per cui appunto, bisognerebbe generarne una nuova. In futuro si parla di usare chiavi diversi per ogni coppia client-server, rimuovendo quindi la necessità di sostituire *tutte* le chiavi in caso di compromissione. Non si esclude che a quel punto tale soluzione verrà adottata.

A puro scopo di documentazione, il comando per generare una chiave valida per **tls-auth** o **tls-crypt** e scriverla in un file chiamato `tls_crypt.key` è il seguente:

```
openvpn --genkey --secret tls_crypt.key
```

Il contenuto di tale file è simile al seguente:

```
#
# 2048 bit OpenVPN static key
#
-----BEGIN OpenVPN Static key V1-----
54d84d5baa8534de608864311958bc82
26d630157c0f7c03306d83f52eda7b6e
bd5b95cfb9f107e1e968fb14d72ba515
81f9fdb74473eea7e68682483c026170
532994275e0c81a07f23231aeefa816a
837dbd8ccba4fbbac1fd281dc986e403
cb6486b57066b96790bae3b1d102757d
3c09b96c0f21f8e5d5836301b41aab11
d55c4167110952971fd698fc89c03c5f
897c487adb18149d892668c0c54f1fe2
6a65c112acbbd4f00977993d073e34ae
140bc7159da6a9a7c5075b6baf6dcf26
545cfe36478ed731cb53e5a3a36bdc0
3f92ecec02d8060ff3afb3b58c560ae5
48f0a13a443cebe31431f8234586c0f9
20c5f523330dc5b0ff5e4c9478bd2357
-----END OpenVPN Static key V1-----
```

### tls-verify

`tls-verify` è un hook che scatta quando un client ha appena concluso il TLS handshake ed ha passato tutte le verifiche, tranne, eventualmente, quella della CRL. Il lanciato con questo hook ha accesso ad un certo numero di variabili d'ambiente (minori rispetto a `client-connect` perché non è stato ancora completamente *accettato*).

Si può usare questo hook per una sorta di *abilitazione esplicita* dei client, per la quale dei dati relativi al nuovo client (come ad esempio il seriale), devono essere *trasmessi* al server per poter accettarlo. Si noti che comunque affinché un client possa completare il TLS handshake deve avere un certificato valido creato dalla CA in questione.

A puro titolo esemplificativo, si presente uno script in Python che verifica che il seriale settato al lancio da OpenVPN sia in una lista di seriali noti, in caso positivo ritorna 0, sennò -1.

```
#!/usr/bin/env python3

import os

# lista dei seriali ammessi
clients = ['3', '4']

returned_value = -1

# accesso alla variabile d'ambiente
```

```

serial = os.getenv('tls_serial_0', None)

# se la variabile d'ambiente non è definita ritorna 0,
# perché vi sono casi 'leciti' in cui
# tale variabile non sia settata, cioè quando si stanno
# verificano tutti i certificati superiori nella
# gerarchia a quello del client in questione.
if serial is None or (serial is not None and serial in clients):
    returned_value = 0

exit(returned_value)

```

Per abilitare l'uso dello script:

```
tls-verify /etc/openvpn/server/cs/tls_verify.py
```

## 5.2 Protezione di MoonCloud

L'utilizzo di OpenVPN per collegare la rete MoonCloud alle reti target presenta un semplice problema di sicurezza: la VPN *apre* un nuovo punto di ingresso in MoonCloud, e come tale deve essere protetto il più possibile.

A tale scopo sono state predisposte delle regole iptables da applicare al server OpenVPN, essendo esso il *punto di esposizione*.

Sebbene il NAT *lato server* di fatto agisca come una sorta di firewall consentendo solo al traffico proveniente dalla rete di MoonCloud ed alle sue risposte di arrivare ed essere accettato dal server, esso non è sufficiente. Lo scenario dettagliato di seguito ne è infatti la prova.

Se qualcuno, ad esempio un attaccante, effettuasse delle connessioni dal VPN client senza utilizzare il rimappaggio degli indirizzi IP (passando per la VPN) esse arriverebbero senza problemi al VPN server ed anche oltre. Questo accadrebbe perché OpenVPN server si limiterebbe a decifrare i pacchetti provenienti dal client ed a scriverli sulla scheda di rete virtuale; allo stesso modo iptables non applica il NAT in uscita sul server per i *pacchetti* in uscita, ma solo per quelli che hanno come destinazione le reti mappate.

Un attaccante riesce quindi con successo (è stato testato) ad inviare pacchetti al server in questo modo. Gli si presentano però alcuni problemi:

- se volesse inviare pacchetti ad una qualsiasi *Docker machine* passando per il VPN server, dovrebbe conoscerne l'indirizzo IP
- non avrebbe modo di vedere le risposte alle proprie richieste, poiché esse sono fatte con l'IP sorgente reale e non mappato, e le *Docker machine* non hanno nessuna configurata che dica che le risposte verso tali indirizzi IP *reali* debbano passare per il VPN server. Alla peggio, l'indirizzo IP sorgente originale della rete target usato dall'attaccante corrisponde ad un'altra rete rimappata, a cui eventualmente tornerebbero risposte.

Sebbene esistano questi due ostacoli, rimangono comunque due possibilità che l'attaccante può sfruttare:

- mandare pacchetti al server
- mandare pacchetti alle *Docker machine*, assumendo di conoscerne l'indirizzo IP, senza curarsi delle risposte

Per contrastare questa possibile minaccia, come anticipato, si è utilizzato iptables. L'idea delle regole applicate si può riassumere così:

- usare una politica `DEFAULT DENY` sul VPN server per cui *solo il traffico in risposta a richieste delle Docker machine* può passare sul server
- ogni volta che un nuovo client si connette, si *concede alle reti mappate di tale client* l'accesso

Naturalmente, queste regole devono riguardare solo il traffico proveniente/destinato alla VPN, pertanto un ulteriore matching applicato alle regole riguarda la scheda di rete virtuale destinazione/sorgente dei pacchetti, che appunto deve essere quella di OpenVPN. Poiché tali schede seguono tutte la stessa nomenclatura, ovvero iniziano con `"ovpn"`, è stato facile applicare le restrizioni in questione<sup>1</sup>.

La protezione è stata affidata ad iptables inizialmente ed a nftables poi. Concretamente, le regole applicate sono le stesse. Si mostrano prima le regole iptables.

### 5.2.1 Politica di default

```
iptables -t filter -I INPUT -i ovpn+ -j DROP
iptables -t filter -I FORWARD -i ovpn+ -j DROP
```

Le due regole che si vedono applicano la politica di `DEFAULT DENY`:

- `iptables -t filter -I INPUT -i ovpn+ -j DROP` dice di droppare il traffico ricevuto (INPUT) da tutte le schede di rete che iniziano con `"ovpn"` (il carattere `"+"` fa da *jolly*) e destinato al server locale
- `iptables -t filter -I FORWARD -i ovpn+ -j DROP` invece dice di droppare il traffico in transito sul server locale (FORWARD), quindi destinato all'esterno, che è stato ricevuto da schede di rete OpenVPN

Si tratta di una politica di `DEFAULT` poiché si usa il flag `"-I"` che indica di aggiungere la regola in questione all'inizio della chain, e per tale ragione sarà la prima valutata da iptables.

Queste regole vengono inserite sui server OpenVPN appena vengono creati, e devono essere sempre reinserite ad ogni riavvio, poiché sono la *base* di tutto il meccanismo di protezione di MoonCloud. E' fuori da questa sede la discussione di come garantire la persistenza delle regole iptables tra un riavvio e l'altro.

---

<sup>1</sup>Si ricorda che è possibile specificare nel file di configurazione di OpenVPN quale nome dare alle schede di rete virtuali.



### 5.2.2 Traffico consentito

Lo step successivo la definizione delle politiche di default è definire le regole che consentano al traffico legittimo di passare.

Queste regole si attivano ogni volta che un client si connette al server, e per questo motivo si sfrutta ancora una volta l'hook `client-connect` di OpenVPN. Nello script che viene eseguito dal server quando un client si è connesso, si hanno quindi anche le regole iptable per consentire *solo le richieste dalle Docker machine alle reti rimappate di tale client ed alle sue risposte* di transitare. Supponendo che la rete rimappata di un client dal `CommonName = client100` sia `192.168.100.0/24`, le regole per farlo sono:

```
iptables -t filter -I OUTPUT -d 192.168.100.0/24 -o ovpn+
↪ -m state --state NEW,RELATED,ESTABLISHED -j ACCEPT
iptables -t filter -I INPUT -s 192.168.100.0/24 -i ovpn+ -m
↪ state --state RELATED,ESTABLISHED -j ACCEPT
iptables -t filter -I FORWARD -s 192.168.100.0/24 -i ovpn+
↪ -m state --state RELATED,ESTABLISHED -j ACCEPT
```

Innanzitutto, si utilizzano regole *statefu*, per cui iptables mantiene una tabella *di stato*, ad ogni nuova connessione, se consentita, si inserisce una nuova entry in tale tabella, tutte gli altri pacchetti di tale connessione non verranno più verificati ma saranno consentiti; ciò rende più difficile attacchi di TCP SESSION HIJACKING, dove un attaccante tenta di inserirsi in una connessione TCP.

- `iptables -t filter -I OUTPUT -d 192.168.100.0/24 -o ovpn+ ...-j ACCEPT` è la regola che consente al traffico generato localmente e destinato ad OpenVPN di passare. Non c'è un corrispettivo nelle regole di default poiché si tratta di *traffico generato dal server*, non da qualcuno di esterno, pertanto non vi è motivo di negare di default questo tipo di traffico. L'unico traffico consentito rimane comunque quello verso la rete mappata `192.168.100.0/24`.
- `iptables -t filter -I INPUT -s 192.168.100.0/24 -i ovpn+ ...-j ACCEPT` invece è la regola inserita ("`-I`") più specifica rispetto alla regola di default che nega il traffico ricevuto da OpenVPN, quindi destinato al server locale, di passare. Con questa regola si consente solo al traffico proveniente da `192.168.100.0/24` destinato al server di transitare. In particolare, si consente *alle sole risposte* (`-m state -state RELATED,ESTABLISHED`) provenienti da `192.168.100.0/24` di passare. E' molto importante la mancanza di "`-state NEW`", poiché è ciò che abilita, appunto, le sole risposte.
- `iptables -t filter -I FORWARD -s 192.168.100.0/24 -i ovpn+ ...-j ACCEPT` al pari della precedente, è più specifica rispetto alla regola che di default vieta il traffico dal server OpenVPN verso l'esterno. Eseguendola, si consente *alle sole risposte* provenienti da `192.168.100.0/24` di raggiungere le *Docker machine*. Come nella precedente, si noti che manca "`-state NEW`".

**Rimozione delle regole** Poiché le regole appena elencate devono essere aggiunte al server OpenVPN solo quando un client si connette, è chiaro che devono essere altresì rimosse quando esso si disconnette. Per farlo, molto semplicemente, ci si affida all'hook `client-disconnect`, riscrivendo le stesse regole ma con l'opzione `"-D"`, che indica di rimuovere la regola così composta.

Supponendo sempre di aver a che fare con `client100`, per rimuovere le precedenti regole:

```
iptables -t filter -D OUTPUT -d 192.168.100.0/24 -o ovpn+
↪ -m state --state NEW,RELATED,ESTABLISHED -j ACCEPT
iptables -t filter -D INPUT -s 192.168.100.0/24 -i ovpn+ -m
↪ state --state RELATED,ESTABLISHED -j ACCEPT
iptables -t filter -D FORWARD -s 192.168.100.0/24 -i ovpn+
↪ -m state --state RELATED,ESTABLISHED -j ACCEPT
```

Le regole combaciano alla perfezione con quelle usate in `client-connect`, affinché `iptables` possa rimuoverle con successo.

### 5.2.3 Configurazione applicata

Sebbene sia già stato mostrato nel capitolo precedente (a partire da pagina 70), di seguito si riporta come si presentano i file `client-up.sh` e `client-down.sh`, assumendo, come in precedenza, che la rete mappata di `client100` sia `192.168.100.0/24`.

```
# /etc/openvpn/ovpn1/cs/client-up.sh

#!/bin/bash

if [ "$common_name" == "client100" ]; then
    # aggiunta rotta
    ip route add 192.168.100.0/24 via 10.7.0.1

    # aggiunta nat lato server
    iptables -t nat -A POSTROUTING -d 192.168.100.0/24 -j
↪ MASQUERADE

    # aggiunta traffico consentito
    iptables -t filter -I OUTPUT -d 192.168.100.0/24 -o
↪ ovpn+ -m state --state NEW,RELATED,ESTABLISHED -j
↪ ACCEPT
    iptables -t filter -I INPUT -s 192.168.100.0/24 -i
↪ ovpn+ -m state --state RELATED,ESTABLISHED -j ACCEPT
    iptables -t filter -I FORWARD -s 192.168.100.0/24 -i
↪ ovpn+ -m state --state RELATED,ESTABLISHED -j ACCEPT
fi

# /etc/openvpn/ovpn1/cs/client-down.sh
```

```
#!/bin/bash

if [ "$common_name" == "client100" ]; then
    # rimozione rotta
    ip route add 192.168.100.0/24 via 10.7.0.1

    # rimozione nat lato server
    iptables -t nat -D POSTROUTING -d 192.168.100.0/24 -j
↪ MASQUERADE

    # rimozione regole di filtraggio
    iptables -t filter -D OUTPUT -d 192.168.100.0/24 -o
↪ ovpn+ -m state --state NEW,RELATED,ESTABLISHED -j
↪ ACCEPT
    iptables -t filter -D INPUT -s 192.168.100.0/24 -i
↪ ovpn+ -m state --state RELATED,ESTABLISHED -j ACCEPT
    iptables -t filter -D FORWARD -s 192.168.100.0/24 -i
↪ ovpn+ -m state --state RELATED,ESTABLISHED -j ACCEPT
fi
```

### 5.2.4 Protezione con nftables

Si sfrutta appieno la struttura dati *Set*: l'idea è che si abbiano delle regole di default analoghe alle precedenti, e che si usi un insieme in cui si inseriscano le reti mappate. In questo modo, quando un client si connette si devono solo aggiungere elementi a tale insieme; viceversa quando si disconnette ("client-disconnect") è sufficiente rimuovere elementi da esso. In questo modo, si mantengono un certo numero di regole fisse (poche), aumentando le prestazioni del firewall.

Le regole sono raccolte in un file `init.nft` che deve essere eseguito sui VPN server al loro avvio:

```
#!/usr/sbin/nft -f

table ip ovpn_management {

    # 'interval' perché si inseriranno
    # dei NET ID
    set allowed_clients {
        type ipv4_addr;
        flags interval;
    }

    # traffico generato dal server verso i client
    chain output {
        type filter hook output priority 0; policy accept;
        ip daddr @allowed_clients meta oifname "ovpn*"
        ct state new,related,established accept
    }
}
```

```

# traffico dai client destinato al server
chain input {
    type filter hook input priority 0; policy accept;
    # consentire traffico dai client registrati
    ip saddr @allowed_clients meta iifname "ovpn*"
        ct state related,established accept
    # tutto il resto del traffico che viene dalla VPN
    # viene droppato
    meta iifname "ovpn*" drop
}

# traffico dai client alle Docker machines
chain forward {
    type filter hook forward priority 0; policy accept;
    # consentire traffico dai client registrati
    ip saddr @allowed_clients meta iifname "ovpn*"
        ct state related,established accept
    # il resto del traffico che viene dalla VPN viene
    # droppato
    meta iifname "ovpn*" drop
}

# masquerading verso la VPN
chain masquerading {
    type nat hook postrouting priority 0; policy accept;
    ip daddr @allowed_clients meta oifname "ovpn*"
        masquerade
}
}

```

Infine, ecco come si presentano i nuovi file per gli hook `client-connect` e `client-disconnect`.

```

# /etc/openvpn/ovpn1/cs/client-up.sh

#!/bin/bash

if [ "$common_name" == "client100" ]; then
    ip route add 192.168.10.0/24 via 10.7.0.1
    nft add element ovpn_management allowed_clients {
        ↪ 192.168.100.0/24 }
fi

# /etc/openvpn/ovpn1/cs/client-down.sh

#!/bin/bash

```

```

if [ "$common_name" == "client100" ]; then
    ip route del 192.168.10.0/24 via 10.7.0.1
    nft delete element ovpn_management allowed_clients {
        ↪ 192.168.100.0/24 }
fi

```

## 5.3 Scenari di attacco

In questa sezione si discutono alcuni scenari di attacco che sfruttano la VPN. Per ciascuno di essi, si utilizza la seguente struttura:

**Scenario** Una breve descrizione dello scenario.

**Livello di difficoltà** Lo sforzo richiesto all'attaccante per portare a termine l'attacco, di cosa deve disporre per avere successo.

**In caso di successo** Che cosa otterrebbe l'attaccante in caso di successo.

**Criticità** Quanto è grave l'attacco.

**Prevenzione** Le principali per mitigare la possibilità che l'attacco si verifichi.

**Ulteriori misure preventive** Ulteriori misure di prevenzione secondaria attuate e attuabili.

**Cosa fare in caso di attacco** Nel caso in cui l'attacco si verifichi, quali contromisure occorre adottare. Oltre a quelle indicate, è sempre necessario verificare *come mai* l'attacco si sia verificato (ovvero: *come ha fatto l'attaccante a riuscirci?*).

Poiché allo stato delle cose non è ancora stato approntato come sarà effettivamente realizzato il VPN client (es: quale versione di Linux? Quali utenti? Quali interfacce? Ecc...), è prematuro analizzare le contromisure da attuare sui client. Inoltre, è comunque possibile che l'attaccante usi un proprio dispositivo client, su cui ovviamente MoonCloud non ha controllo.

E' molto importante notare che la protezione data dalle regole di firewalling è molto efficace, anche nei casi di attacchi con livello di criticità alto.

### 5.3.1 Scenario 1

**Scenario** Attacco (D)DoS verso i VPN server.

**Livello di difficoltà** Basso, è sufficiente disporre di sufficiente potenza di calcolo.

**In caso di successo** L'operatività di MoonCloud sarebbe compressa per tutta la durata dell'attacco.

**Criticità** Alta, anche grazie alla relativa facilità dell'attacco.

**Prevenzione** Sebbene si assume che l'attaccante non disponga di certificati validi, e che quindi la connessione VPN non andrebbe a buon fine, se l'attaccante riuscisse a creare un'adeguata mole di traffico, allora riuscirà a sovraccaricare il server quanto basta per causare un'interruzione di servizio. L'opzione `tls-crypt` potrebbe mitigare gli effetti dell'attacco, oppure richiedere maggiore sforzo all'attaccante. Non sarebbe comunque una soluzione definitiva.

**Ulteriori misure preventive** Non vi sono altre misure di prevenzione.

**Cosa fare in caso di attacco** E' tristemente noto come sia difficile proteggersi da attacchi DoS. Si potrebbero spegnere i server coinvolti nell'attacco (e magari anche quelli non coinvolti per precauzione) fino a che l'ISP non riesce a bloccare il traffico malevolo in anticipo.

### 5.3.2 Scenario 2

**Scenario** Inviare pacchetti malevoli alla rete MoonCloud mediante un VPN client non legittimo.

**Livello di difficoltà** Basso. Si suppone che l'attaccante abbia a disposizione un qualsiasi dispositivo con OpenVPN installato, e che voglia connettersi ad un VPN server e sfruttarlo per inviare pacchetti malevoli alla rete MoonCloud.

**In caso di successo** Non vi sono possibilità di successo, perché non disponendo di un VPN client valido, non riuscirebbe a connettersi.

**Criticità** Bassa a causa dell'impossibilità di successo.

**Prevenzione** OpenVPN rifiuta connessioni da client che non riesce a verificare.

**Ulteriori misure preventive** Nessuna.

**Cosa fare in caso di attacco** Se non si tratta di un attacco DoS, non c'è molto da fare.

### 5.3.3 Scenario 3

**Scenario** Inviare pacchetti malevoli alla rete MoonCloud mediante un VPN client legittimo (in tutto o in parte).

**Livello di difficoltà** Medio. Si suppone che l'attaccante abbia a disposizione un VPN client legittimo, ovvero con un certificato valido. E' possibile che l'attaccante sia riuscito ad ottenere un VPN client *ufficiale*, o che sia riuscito ad ottenere da esso un certificato valido.

**In caso di successo** L'attaccante riuscirebbe a connettersi ai server VPN, ma le sue possibilità si fermerebbero lì grazie alle contromisure.

**Criticità** Media, la rete MoonCloud è intatta e la vulnerabilità è eventualmente sul client VPN.

**Prevenzione** Grazie alle regole di firewalling sui server, se anche l'attaccante ottenesse dei certificati validi, non potrebbe *in ogni caso* inviare pacchetti a MoonCloud, poiché sono consentite solo risposte a richieste dalle Docker machine.

**Ulteriori misure preventive** E' possibile revocare il certificato usato dall'attaccante, limitando del tutto la sua possibilità di azione.

**Cosa fare in caso di attacco** Il servizio MoonCloud\_VPN offre una API dedicata alla revoca dei certificati, questo è il primo passo da seguire.

#### 5.3.4 Scenario 4

**Scenario** Compromissione della chiave private della CA

**Livello di difficoltà** Alto: la chiave privata della CA non è mai distribuita al di fuori della rete MoonCloud, tantomeno viene distribuita ai VPN server.

**In caso di successo** Un attaccante potrebbe generare dei nuovi certificati validi per client VPN. Egli potrebbe quindi connettersi senza alcuna difficoltà ai VPN server.

**Criticità** Molto alta, l'attaccante è per forza di cose riuscito ad introdursi nella rete MoonCloud ed aver ottenuto la chiave privata.

**Prevenzione** La chiave privata è utilizzata *esclusivamente* dal microservizio MoonCloud\_VPN, e sono in approntamento misure di protezione basate su firewall per far sì che solo richieste legittime arrivano al servizio.

**Ulteriori misure preventive** Anche se l'avversario riuscisse a connettersi, non potrebbe comunque modificare le regole di firewalling che consentono alle sole risposte di transitare dai VPN server verso la rete MoonCloud. Per limitare ulteriormente lo spazio di manovra dell'attaccante si può usare l'opzione `tls-verify` per una sorta di *abilitazione esplicita*.

**Cosa fare in caso di attacco** E' chiaro che la compromissione della chiave privata sia uno scenario estremamente grave, pertanto vi sono numerose contromisure attuabili per limitare i danni. Tuttavia, in caso in cui questo si verifichi, occorre spegnare i server VPN, e chiaramente generare delle nuove chiavi.





# Capitolo 6

## MoonCloud\_VPN

*MoonCloud\_VPN* è il nome del microservizio che ho scritto per l'automatizzazione della gestione di tutti gli aspetti legati alla soluzione VPN proposta; questo capitolo è dedicato alla sua descrizione, si mostreranno anche degli esempi di codice. Un elenco delle principali caratteristiche:

- scritto in Python usando il framework `Django`
- in produzione viene deployato in un container `Docker`
- si sfrutta il DBMS `PostgreSQL` ed il Key-Value store `Redis`

### 6.1 Requisiti

Il microservizio implementa delle API REST per le seguenti funzionalità:

**Configurazione VPN server** Creazione di tutti i file di configurazione necessari ad OpenVPN server per funzionare.

**Configurazione device client** Creazione di tutti i file richiesti al device VPN client, compresi quelli necessari ad OpenVPN ed il file per `nftables`.

**Gestione dei certificati** Client e server hanno bisogno di certificati X509 per poter stabilire un collegamento VPN. E' richiesto di creare tali certificati, sia per client sia per server, e di gestirne il loro ciclo di vita, compreso di rinnovo ed eventuale revoca.

**Trasferimento file** Una volta che file di configurazione e certificati sono stati creati, devono essere trasferiti sul VPN server in maniera *sicura*. Allo stesso modo, ogni volta che si crea un nuovo client, occorre aggiornare la configurazione del server a cui si conatterà (si pensi ai file `client-up.sh` e `client-down.sh`). Questo trasferimento deve inoltre preservare l'integrità della configurazione del server.

**Visualizzare informazioni** Devono essere disponibili delle API che ritornino informazioni su specifici client e server, incluse anche informazioni sui certificati.

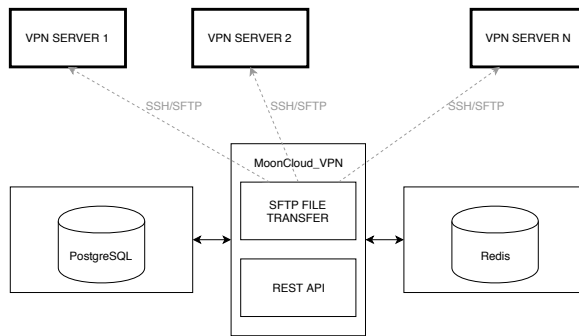


Figura 6.1: Il deployment di MoonCloud\_VPN. I rettangoli con linee in grassetto rappresentano le VM su cui vi sono i VPN server, mentre quelli con contorno normale rappresentano i tre container di cui il deployment si compone.

**Gestione dell'IP mapping** Questa attività si scompone in:

- ogni volta che si crea un nuovo client VPN, allocare ad esso  $n$  reti mappate univoche per il server al quale esso si collegherà
- dato un indirizzo IP originale di un certo client, ritornare l'IP mappato
- *Blacklisting*: deve essere possibile specificare che alcuni indirizzi IP (indirizzi di host o indirizzi di rete) o nomi di dominio<sup>1</sup> non devono essere mai assegnati a reti mappate.

**Revocare client** Per vari ragioni, deve essere necessario poter revocare un client. Questa attività si traduce nell'invalidare il certificato ad esso relativo ed alla cancellazione del mappaggio dal database. La revoca deve essere propagata ad ogni server in modo che esso non riesca per nessun motivo a connettersi nuovamente alla VPN di MoonCloud.

## 6.2 Architettura

L'intero deployment del mio microservizio si compone di un container su cui è in esecuzione Django, cioè il web framework che espone le API REST, di un secondo container su cui è in esecuzione PostgreSQL, di un terzo dedicato a Redis. La figura 6.2 mostra quanto descritto. Non è escluso che in futuro questo servizio venga separato in microservizi ancora più piccoli, ad esempio si sta studiando la separazione della parte di trasferimento file. Realizzarlo non sarebbe difficile poiché le singole componenti sono il più possibile separate ed indipendenti tra loro le singole componenti.

Il microservizio è composto da diversi moduli, ciascuno dei quali ben specializzato per una certa funzionalità, ad esempio: creazione file di configurazione server, creazione file per nftables, trasferimento file, ecc... Il modulo `controllers.py` definisce una serie di funzioni che, tramite `views.py` risponde direttamente alle chiamate alle API del servizio. Si cura di chiamare i metodi delle classi di tutti gli altri moduli.

<sup>1</sup>Si intende l'indirizzo IP corrispondente a tale nome di dominio.

**client** Questo modulo si occupa delle configurazioni relative ai client VPN. In particolare, contiene classi responsabili di creare il file di configurazione principale per un VPN client, creare i file client-specific da trasferire sul server, creare il file di script per nftables dato il mapping da usare per quel client.

**dnmcp** Si occupa in maniera specifica di creare il mapping date le reti originali. Una volta che è stato definito, è responsabile di salvarlo nel database.

**dns** Un modulo specifico dedicato alla risoluzione dei nomi DNS relative al blacklisting. Oltre a risolvere nuovi nomi ricevuti in input, è anche responsabile di aggiornare tutti quelli presenti nel database. La risoluzione viene fatta in maniera asincrona, ovvero si fanno tutte le query necessarie, una dopo l'altra senza aspettare risposta, e solo quando l'ultima è stata completata si aspettano i risultati complessivi. Questo consente di aumentare le performance mediante I/O non bloccante.

**openssl** Una classe preposta alla creazione, revoca ed aggiornamento dei certificati di client e server, e gestione della CRL. Utilizza una libreria Python che a sua volta utilizza OpenSSL come backend.

**server** L'unico compito dell'unica classe di questo modulo è quello di creare il file di configurazione principale per un VPN server, dati una serie di parametri in input.

**trans** Il nome del modulo è l'abbreviazione di "transfer". Si compone di una serie di classi responsabili di trasferire file sui server mediante `sftp` (trasferimento file su un canale sicuro SSH) e di creare la struttura di directory così come assunta nel file di configurazione di OpenVPN (esempio: creare le cartelle `/etc/openvpn/.../ccd/` e `/etc/openvpn/.../certs`).

**workers** Contiene la classe `SSHBackgroundWorker` la quale, mediante due thread separati, si occupa del trasferimento mediante il modulo `trans`. Maggiori dettagli su questa funzionalità verranno fornite in seguito durante il capitolo.

Infine si descrivono i seguenti moduli:

**controllers** Organizzato in diversi file, questo modulo si occupa di *unire* le funzionalità messe a disposizione da tutti gli altri moduli, oltre a creare le istanze dei dati da memorizzare nel database.

**models** Contiene la definizione delle tabelle memorizzate nel DBMS, secondo i cosiddetti "Model" nella terminologia Django<sup>2</sup>.

**views** Definisce le API a cui il microservizio risponde, deserializza e serializza gli input e gli output, fa le chiamate a `controllers`, ne intercetta eventuali errori.

---

<sup>2</sup><https://docs.djangoproject.com/en/2.1/topics/db/models/>

## 6.3 Dettagli

In questa sezione si affrontano in maniera più dettagliata le principali funzionalità offerte dal microservizio. In letteratura vi è un ampio dibattito su quale sia l'effettiva dimensione di un microservizio, e di fatto il risultato è che un esso dovrebbe fare una cosa ed una soltanto. Si potrebbe quindi dire che questo principio non è completamente rispettato in MoonCloud\_VPN. A tale proposito, vanno fatte le seguenti osservazioni:

- il microservizio nella sua interezza non è particolarmente complesso, anche dal punto di vista algoritmico, a parte il trasferimento file ed il mappaggio delle reti il codice è abbastanza semplice. Si tratta di circa 4000 righe di codice Python, escludendo i test ed ulteriori script, che è una dimensione ragionevole.
- Il microservizio svolge una precisa funzionalità, ovvero gestire tutto ciò che riguarda la configurazione di OpenVPN per MoonCloud.
- Separare alcune parti, come ad esempio i moduli che si occupano solo di generare file di configurazione, renderebbe le cose più complicate, perché dovrebbero essere comunque trasferiti localmente una prima volta, per poter quindi essere spostati sui server.

Queste considerazioni hanno portato a lasciare questo microservizio come un *unico* microservizio.

### 6.3.1 Certificate Management

Requisiti fondamentale per il funzionamento di OpenVPN è che il client e server dispongano di certificati X509 firmati da una stessa CA. A tale scopo, si dispone di una CA interna, la cui chiave pubblica e privata è deployata assieme a questo servizio<sup>3</sup>. Attualmente il certificato è self-signed, in futuro potrebbe non essere così.

OpenVPN richiede, sia per client sia per server, che siano settati alcuni attributi specifici nei certificati, da cui poi derivano le direttive `"remote-cert-eku "TLS Web Server/Client Authentication"`. Ogni certificato ha un seriale univoco, il quale viene generato in maniera pseudocasuale usando una funzione preposta dalla libreria che viene utilizzata.

Il microservizio mantiene anche una CRL – *Certificate Revocation List*, la quale contiene l'elenco dei seriali dei certificati revocati, la ragione (nel cui caso è `"unspecified"`), ed è firmata digitalmente dalla chiave privata della CA. Ogni volta che si revoca un nuovo client, occorre creare una nuova CRL inserendo tutti i precedenti certificati revocati aggiungendo quello nuovo. La CRL deve poi essere distribuita a tutti i server, ma questo è compito di un altro modulo.

I certificati sono generati utilizzando l'algoritmo ECDSA, con livello di sicurezza 128 bit per client e server. Secondo le raccomandazioni del NIST, un livello

---

<sup>3</sup>In futuro si potrebbero maggiori di protezione per la chiave privata, oltre alla protezione *fisica* (es: regole di firewall) dell'accesso al servizio.

di sicurezza di 128 è sicuro fino al 2030 ??). Si può passare senza problemi ad un livello di sicurezza maggiore (192 o 256), semplicemente modificando i settaggi del microservizio.

Per il certificato della CA, si utilizza invece una curva ellittica con livello di sicurezza di 256.

Tutto il codice per la gestione di queste funzionalità è implementato nel modulo `openssl` mediante il file `openssl.py`, di cui di seguito si mostra un estratto leggermente modificato. Si può vedere come si crea un certificato ed una CRL vuota (metodo “`create_empty_crl`”, per la quale vi si aggiunge un certificato fake il cui numero seriale vale 1. Il tipo di certificato (client o server), dipende dai parametri con cui si chiama il metodo “`create_keys_and_cert`”. Alcuni metodi sono omessi per brevità, tra essi il costruttore.

```
import cryptography.hazmat.backends as backends
import cryptography.hazmat.primitives.asymmetric.ec as ec
import cryptography.hazmat.primitives.hashes as hashes
import cryptography.hazmat.primitives.serialization as serialization
import cryptography.x509 as x509

class OpenSSL(object):

    def __get_ca_private_key(self):
        if self.__root_ca_key_path is None:
            raise Exception('You should have provided the ca\'s key')

        ca_priv_key_file = open(self.__root_ca_key_path, 'rb')

        ca_private_key = serialization.load_pem_private_key(
            data=ca_priv_key_file.read(),
            backend=backends.default_backend(),
            password=None,
        )

        ca_priv_key_file.close()
        return ca_private_key

    def __get_revoked(self, serial):
        return x509.RevokedCertificateBuilder(
        ).serial_number(
            int(serial)
        ).revocation_date(
            datetime.datetime.today()
        ).add_extension(
            x509.CRLReason(x509.ReasonFlags.unspecified),
            critical=False
        ).build(backend=backends.default_backend())

    def __get_crl_builder(self, issuer_name, crl_number, last_update):
        return x509.CertificateRevocationListBuilder(
        ).issuer_name(
            issuer_name
        ).last_update(
            last_update
        ).next_update(
```

```

        last_update + self.NEXT_UPDATE
    ).add_extension(
        x509.CRLNumber(crl_number),
        critical=False
    )

def create_empty_crl(self):
    """
    Generates a certification revocation list (CRL) with
    a single revoked cert with serial = 1.

    :rtype str the path to the CRL.
    """

    cert = self.__get_ca_cert()
    if self.__crl_path is None or self.__crl_path == '':
        self.__crl_path = os.path.abspath(os.path.join(
            self.__directory, 'crl.pem'))

    crl_builder = self._get_crl_builder(
        issuer_name=cert.issuer,
        crl_number=1,
        last_update=datetime.datetime.today()
    )

    revoked = self._get_revoked(1)
    crl_builder = crl_builder.add_revoked_certificate(revoked)

    crl = crl_builder.sign(
        private_key=self.__get_ca_private_key(),
        algorithm=hashes.SHA512(),
        backend=backends.default_backend()
    )

    self.__write_public_bytes(self.__crl_path, crl)
    return self.__crl_path

def _basic_cert_setup(self, data, cert_type, serial=None):
    """
    Does a basic setup for the certificate.

    - deciding which curve to use
    - settings all of the attributes of the cert
    - generate a serial if it's not given

    :rtype tuple:
    - tuple[0] private_key
    - tuple[1] public_key
    - tuple[2] str serial
    - tuple[3] x509.CertificateBuilder the cert builder
    - tuple[4] datetime.datetime issuing date
    - tuple[4] datetime.datetime expiration date
    """
    common_name = data['CommonName']
    days = data.get('Days', 365)
    email = data['Email']
    organization_name = data['OrganizationName']

```

```

self.__current_security_level =\
    self.__get_current_security_level(cert_type)

curve, curve_str =\
    self.__get_curve(self.__current_security_level)

self.__curve = curve_str

private_key, public_key = self.__gen_keys_pair(curve)

cert_attributes = x509.Name([
    x509.NameAttribute(x509.oid.NameOID.COUNTRY_NAME, 'IT'),
    x509.NameAttribute(x509.oid.NameOID.ORGANIZATION_NAME,
                        organization_name),
    x509.NameAttribute(x509.oid.NameOID.COMMON_NAME,
                        common_name),
    x509.NameAttribute(x509.oid.NameOID.EMAIL_ADDRESS,
                        email),
])

if serial is None:
    serial = gen_random_serial()

issuing_date = datetime.datetime.now(datetime.timezone.utc)
expiration_date = issuing_date + datetime.timedelta(days=days)

builder = x509.CertificateBuilder(
).subject_name(
    cert_attributes
).not_valid_before(
    issuing_date
).not_valid_after(
    expiration_date
).public_key(
    public_key
).serial_number(
    serial
)

return (private_key, public_key, serial, builder,
        issuing_date, expiration_date)

def __gen_keys_pair(self, curve):
    private_key = ec.generate_private_key(
        curve=curve,
        backend=backends.default_backend())
    public_key = private_key.public_key()

def create_keys_and_cert(self, data, cert_type, serial=None):
    """
    Does the following:
    - generates a key pair using ECDSA
    - generates a certificate for that public key
    - save to two files the certificate and the private key

    :param data    dict, keys:

```

```

- 'CommonName' str the 'CommonName' to insert into
  the cert
- 'Email' str the 'email' to insert into the cert
- 'Days' int the days for which the cert is valid
  starting from now
- 'OrganizationName' str the 'OrganizationName' to
  insert into the cert
:param cert_type enum of 'CertType'
:param serial str the optional serial to insert into the
  certificate.
If 'None', a brand new one is generated using
  randomness.

:note 'serial', if is not 'None', should be pseudorandom.

:rtype tuple, as follows:
- tuple[0] = (path-to-cert, path-to-privkey-file)
- tuple[1] = (serial, issuing date, expiration date)

:raises ValueError if 'serial == 1'
"""

if serial is not None and (serial == 1 or serial == '1'):
    raise ValueError('serial cant be = 1')

truncated_name = utils.get_truncated_name(data['CommonName'])

self.__key_file_name = os.path.abspath(
    os.path.join(self.__directory, truncated_name+'.key'))

self.__cert_file_name = os.path.abspath(
    os.path.join(self.__directory, truncated_name+'.crt'))

result = self._basic_cert_setup(data, cert_type, serial)

private_key = result[0]
# public_key = result[1]
serial = result[2]
cert_builder = result[3]
issuing_date = result[4]
expiration_date = result[5]

extensions = None
if cert_type == CertType.server:
    extensions = x509.ExtendedKeyUsage(
        [x509.oid.ExtendedKeyUsageOID.SERVER_AUTH])
elif cert_type == CertType.client:
    extensions = x509.ExtendedKeyUsage(
        [x509.oid.ExtendedKeyUsageOID.CLIENT_AUTH])

cert_builder = cert_builder.add_extension(extensions,
                                          critical=False)

ca_private_key = self.__get_ca_private_key()
ca_cert = self.__get_ca_cert()

cert_builder = cert_builder.issuer_name(ca_cert.subject)

```



```

cert = cert_builder.sign(private_key=ca_private_key,
                        algorithm=hashes.SHA512(),
                        backend=backends.default_backend())

ca_private_key = None

private_key_output = open(self.__key_file_name, 'wb')
private_key_output.write(private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.TraditionalOpenSSL,
    encryption_algorithm=serialization.NoEncryption()
))
private_key_output.close()
private_key = None

self.__write_public_bytes(self.__cert_file_name, cert)

return (self.__cert_file_name,
        self.__key_file_name), (serial,
                                issuing_date,
                                expiration_date)

```

Nel database si memorizzano anche informazioni relative ai certificati, come il CommonName, il seriale, il livello di sicurezza, l'algoritmo di firma. Questa tabella è quindi referenziata dalla relativa tabella che rappresenta i VPN server ed i VPN device client.

Si mantiene anche un tabella con informazioni relative alla CRL, come la data di ultima modifica.

### 6.3.2 IP Mapping

L'IP mapping è una funzionalità fondamentale per poter far sì che un VPN server possa gestire il maggior numero di reti target diverse. Durante l'analisi dei requisiti è stato osservato che gli Execution Manager dedicati all'analisi di reti target sono posti ciascuno in una rete isolata rispetto al resto di MoonCloud, ciò ha portato alle seguenti osservazioni:

- è possibile assegnare più volte la stessa rete mappata a diversi client, a patto che siano collegati a VPN server diversi, e quindi Execution Manager.
- è possibile assegnare *qualsiasi* indirizzo IP, non solo quelli indicati come *privati*, questo perché tali indirizzi sono utilizzati sono nel contesto degli Execution Manager. Per evitare eventuali conflitti, come ad esempio assegnare una rete mappata che contiene l'indirizzo IP di una repository Linux, deve essere possibile poter escludere degli indirizzi, reti, nomi di dominio.

#### Assegnamento reti

Il grafico 6.2 rappresenta, mediante la notazione UML per i diagrammi di sequenza, quali azioni sono svolte per l'assegnamento delle reti, che qui viene descritto a parole.

1. Per ciascuna rete in input da mappare, si crea un corrispondente oggetto (detto “rete canonica”) che ha come subnet mask una subnet mask *classful* in grado di contenere tutti gli indirizzi della rete in input. Ad esempio, se in input vi è una rete con subnet mask a 26 bit, il corrispettivo oggetto è una rete con lo stesso NET ID e con subnet mask a 24 bit.
2. Si raggruppano tutte le reti canoniche per subnet mask.
3. Per ciascun gruppo, si richiede al database un numero di reti mappate pari al numero di reti che costituiscono ciascun gruppo.
4. Si utilizza una funzione (il DBMS PostgreSQL consente di farlo) salvata nel database per questa operazione, poiché si devono effettuare numerose query. Facendo tutto con una funzione nel database, si richiede meno I/O. La funzione garantisce che le reti mappate che vengono infine ritornate non siano nella blacklist. Si noti che grazie ai tipi di dati `INET` e `CIDR`, certe interrogazioni sono molto efficienti (ad esempio esiste un operatore per valutare se un indirizzo IP è contenuto in una rete).
5. Si *allineano* le reti in input con le reti mappate, e le si salvano nel database.
6. Si effettua un secondo allineamento tra le reti mappate e quelli in input, ritornando infine una lista di oggetti che contengono tutte le informazioni per creare il file di script per `nftables`.

Si noti che nel database si salvano, di fatto, delle coppie Rete Originale – Rete Mappata. Si può vedere un indirizzo IP originale nel come l’offset tra esso e il NET ID, ed allo stesso modo, il corrispondente indirizzo mappato si calcola sommando l’offset dell’IP originale al NET ID mappato.

La creazione del mapping è abbastanza complessa, si inserisce come codice d’esempio la creazione del file di script per `nftables`.

```
import ipaddress
from io import StringIO

import cconf.dnmcp.ip_utils as ip_utils

from . import abstract_tables

COMMENT = """\t# map 'mapping' is applied in postrouting for
\t# packets target_net -> mooncloud_net, it changes the
\t# source addresses to the ones expected by mooncloud.

\t# map 'remapping' is applied in prerouting for
\t# packets mooncloud_net -> target_net, it changes the
\t# dest addresses from the mapped ones known by mooncloud
\t# to real ones.

\t# since OpenVPN server does NAT to the VPN, the only
\t# address known here is the VPN subnet."""

class NFTables(abstract_tables.AbstractTables):
    """
```

*Class that creates the mapping file for 'nftables'.*

*The 'shebang' is set to '#!/usr/sbin/nft -f'.*  
*"""*

```
def __init__(self, directory, data):
    super().__init__(directory, data, 'nftables.nft')

def _mapping_str(self):
    nftables_initial = StringIO()
    nftables_mapping = StringIO()
    nftables_remapping = StringIO()
    nftables_masquerading = StringIO()

    final_rules = StringIO()

    nftables_initial.write('#!/usr/sbin/nft -f\n')
    nftables_initial.write('\n')

    nftables_initial.write('# creating the new table\n')
    nftables_initial.write('table ip ovpn_nat {\n')

    nftables_mapping.write('\t# map old -> new\n')
    nftables_mapping.write('\tmap mapping {\n')
    nftables_mapping.write('\t\ttype ipv4_addr: ipv4_addr\n')

    nftables_remapping.write('\t# map new -> old\n')
    nftables_remapping.write('\tmap remapping {\n')
    nftables_remapping.write('\t\ttype ipv4_addr: ipv4_addr\n')

    first = True

    # -j MASQUERADE on everything that comes from the VPN
    for address in self._internal_nets_cidr:
        rule = (
            '\t\tip saddr ' + self._vpn_net_cidr + ' ip daddr '
            + address + ' masquerade'
        )
        nftables_masquerading.write(rule+'\n')

    # building the two maps
    for mapping in self._mappings:

        old_net = mapping.old_network
        new_net = mapping.new_network
        count = mapping.counter
        starting_old_net_address =
→ mapping.old_network_starting_address
        starting_new_net_address =
→ mapping.new_network_starting_address

        if isinstance(old_net, str):
            old_net = ip_utils.CustomIPv4Network(old_net)

        if isinstance(new_net, str):
            new_net = ip_utils.CustomIPv4Network(new_net)
```

```

if isinstance(starting_old_net_address, str):
    starting_old_net_address = \
        ipaddress.IPv4Address(starting_old_net_address)

if isinstance(starting_new_net_address, str):
    starting_new_net_address = \
        ipaddress.IPv4Address(starting_new_net_address)

ending_old_address = starting_old_net_address + count - 1
current_old_address = starting_old_net_address
current_remapped_address = starting_new_net_address
i = 1

mapping_element = ''
remapping_element = ''
while current_old_address <= ending_old_address:

    new_addr = current_remapped_address.exploded
    original_addr = current_old_address.exploded

    prefix = '\t\t\t\t\t'
    if first:
        prefix = '\t\t\t\t\t'
        first = False

    if i % 2 == 0 and prefix != '\t\t\t\t\t':
        prefix = ' '

    end = ', '
    if current_old_address == ending_old_address:
        end = '}'

    current_remapping_element = (prefix + new_addr +
                                ' : ' + original_addr + end)
    current_mapping_element = (
        prefix + original_addr + ' : ' + new_addr + end)

    if i % 2 == 0:
        # new line
        mapping_element = (mapping_element +
                           current_mapping_element + '\n')
        remapping_element = (remapping_element +
                              current_remapping_element +
                              '\n')

    nftables_mapping.write(mapping_element)
    nftables_remapping.write(remapping_element)
else:
    # else stay on this line
    mapping_element = current_mapping_element
    remapping_element = current_remapping_element

    # shift to next elements
    mapping_element = current_mapping_element
    remapping_element = current_remapping_element

    current_remapped_address = current_remapped_address + 1

```

```

        current_old_address = current_old_address + 1

        i += 1

        nftables_mapping.write('\t}\n')
        nftables_remapping.write('\t}\n')

    final_rules.write(nftables_initial.getvalue()+'\n')

    final_rules.write(nftables_mapping.getvalue()+'\n')

    final_rules.write(nftables_remapping.getvalue()+'\n')

    # writing the comment
    final_rules.write(COMMENT+'\n\n')

    # declaring chains
    prerouting_chain = (
        '\tchain prerouting {\n' +
        '\t\ttype nat hook prerouting priority 0; policy accept;\n'
↪ +
        '\t\tip saddr ' + self._vpn_net_cidr + ' dnat to ip daddr
↪ map @remapping\n' +
        '\t}\n'
    )

    postrouting_chain = (
        '\tchain postrouting {\n' +
        '\t\ttype nat hook postrouting priority 0; policy
↪ accept;\n' +
        nftables_masquerading.getvalue() +
        '\t\tip daddr ' + self._vpn_net_cidr + ' snat to ip saddr
↪ map @mapping\n' +
        '\t}\n'
    )

    final_rules.write('\t# chains\n')
    final_rules.write(prerouting_chain)
    final_rules.write(postrouting_chain)

    # the last newline is important
    # nft will signal an error if it's not present.
    final_rules.write('}\n')

    return final_rules.getvalue()

```

### Ritornare indirizzo IP mappato

Quando un client specifica un target per un'analisi, egli lo fa inserendo l'indirizzo IP originale, per lui la questione dell'IP mapping è completamente trasparente. Ciò significa che occorre una funzione che, dato un indirizzo IP originale di un certo VPN client, ritorni quello mappato, poiché esso è l'unico noto a MoonCloud. Tale funzione viene quindi implementata dal mio microservizio, ed è molto performante. I passi da compiere sono:

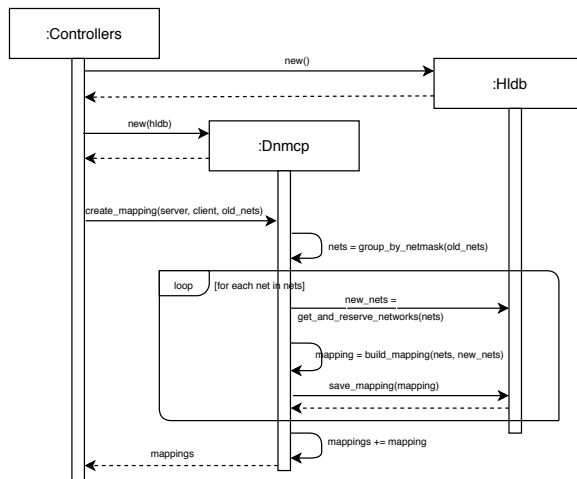


Figura 6.2: Diagramma di interazione che mostra le chiamate tra le vari classi che collaborano per assegnare nuove reti mappate all’aggiunta di un nuovo client (altre azioni, come la creazione dei file di configurazione vengono omesse). La classe Hldb si occupa dell’interazione con il database, mentre DNMCP compie operazioni ad un più alto livello.

- fare una query al database e ritornare la rete mappata cui corrisponde la rete originale a cui appartiene l’indirizzo IP richiesto in input. Anche in questo caso si sfruttano gli operatori specifici messi a disposizione dal DBMS.
- Il corrispondente IP mappato si calcola con i seguenti step:
  - applicare la subnet mask al NET ID originale ed all’IP originale, la subnet mask è memorizzata nel database.
  - Applicare la subnet mask al NET ID mappato o al primo indirizzo IP della rete mappata in caso non sia il NET ID (ad esempio: il mapping parte da 192.168.100.10 anziché 192.168.100.0).
  - Calcolare l’offset dell’indirizzo IP originale sottraendo l’IP originale con la subnet mask applicata al NET ID originale con la subnet mask applicata.
  - Sommare l’offset all’indirizzo IP mappato di partenza con la subnet mask applicata.

Le operazioni potrebbero essere effettuate anche senza l’applicazione della subnet mask, si tenga comunque conto che applicare una subnet mask ad indirizzo IP equivale ad un AND logico, quindi un’istruzione in ogni caso molto veloce.

### 6.3.3 Trasferimento file

Vi sono diverse situazioni che richiedono il trasferimento di file verso i VPN server di MoonCloud. Qualsiasi sia il caso, esso viene effettuato lungo il protocollo SSH, mediante il quale si possono sia inviare comandi remoti o trasferire file con il sottosistema SFTP – Secure File Transfer Protocol. Si sfruttano delle librerie

Python a tale scopo. L'autenticazione viene fatta mediante chiave pubblica. Il trasferimento è richiesto nelle seguenti situazioni:

**Creazione di un nuovo server** Si trasferiscono sul server il file di configurazione principale di OpenVPN, la relativa chiave privata e certificato, la chiave pubblica della CA, la CRL. Vengono trasferiti anche due script:

- `move-files-server.sh`: dopo aver trasferito i file appena elencati in una cartella sul server, viene lanciato questo script, il quale legge il file di configurazione principale e da esso deduce la struttura di directory e file richiesta, e quindi la crea. Poiché le operazioni che lo script deve compiere non sono poche, si è preferito scegliere questa strada anziché invocare direttamente *n* comandi lungo la connessione SSH per ridurre l'I/O e quindi aumentare le prestazioni.
- `deleteclient.py`. Quando si revoca un client, esso deve essere *totalmente eliminato*, ciò si traduce anche nell'eliminare ogni riferimento ad esso dai file `client-up.sh` e `client-down.sh`, quindi eliminare il corpo degli `if` all'interno dei due file. Questa operazione viene svolta direttamente sui server, mediante l'invocazione di questo script. Analogamente a quanto detto per lo script precedente, e a maggior ragione in questo caso, eseguire `deleteclient.py` direttamente sul server riduce di molto l'I/O: se non si fosse scelta questa opzione sarebbe stato necessario trasferire i due file, modificarli, e di nuovo caricarli sul server.

**Creazione di un nuovo client** E' necessario creare un file con lo stesso nome del `CommonName` presente nel certificato del client, il cui contenuto è la direttiva `"iroute"`<sup>4</sup>. Esso deve essere quindi trasferito in una cartella specifica nel VPN server. Oltre a tale file, è necessario modificare gli script `client-up.sh` e `client-down.sh`, aggiungendo un nuovo ramo `if` specifico per questo client.

**Rinnovo certificato server** L'unico trasferimento da effettuare è il nuovo certificato e chiave pubblica, i quali devono essere spostati nella directory specificata, ancora una volta, nel file di configurazione di OpenVPN nel server. Rimane comunque la necessità di riavviare il server affinché questi cambiamenti abbiano effetto.

**Rinnovo certificato client** E' necessario revocare il certificato precedente, quindi propagare la nuova CRL ad ogni server di MoonCloud.

**Revoca certificato client** Similmente al punto precedente, per ciò che concerne il trasferimento file, l'unica azione da effettuare è propagare la nuova CRL ai VPN server.

**Refresh CRL** Quando si crea una CRL si specifica la data di prossimo aggiornamento, ed essa viene controllata da OpenVPN server: se risulta scaduta non

---

<sup>4</sup>Si rimanda al capitolo dedicato alle configurazioni di OpenVPN.

si accettano nuove connessioni. Per questa ragione è importante aggiornarla, ovvero creare una nuova CRL che contenga gli stessi certificati revocati della precedente ma con data di prossimo aggiornamento posticipata di un anno.

Il trasferimento dei file verso i server è una parte fondamentale del microservizio, pertanto deve essere il più possibile robusto. Il modo in cui viene effettuato è stato chiamato *asynchronous transferring*, perché è svolto da due thread specifici a questo scopo, separati dagli altri thread di esecuzione, implementati dalla classe `SSHBackgroundWorker`. Il modulo `controllers` organizza i trasferimenti da effettuare e crea degli oggetti (*dizionari* Python) e li deposita su una delle due code condivise con i thread di trasferimento. La computazione di `SSHBackgroundWorker` è composta da due loop (uno per ogni thread), dentro il quale si estraggono elementi dalla coda: ciascun elemento estratto contiene tutte le informazioni che servono per trasferirlo sul server (o sui server) correttamente. I trasferimenti quindi sono eseguiti serialmente, nell'ordine con cui sono stati inseriti in una delle due code (*FIFO*). Ogni volta che si deposita un nuovo job sulla coda, si riceve un `WorkID`, il quale è generato in maniera pseudocasuale nell'idea di identificare univocamente ogni job.

Lo stato della computazione di un lavoro è mantenuto in memoria da Redis. Tra le varie informazioni che si memorizzano vi sono:

- ID del lavoro (è la Key)
- stato di avanzamento: non iniziato, in corso, concluso, concluso con errore
- tipo (es: *trasferimento server, rinnovo certificato, ecc...*)
- ulteriori informazioni, come una descrizione dell'errore in caso si verifichi.

Quando un thread inizia la computazione per un job modifica lo stato da `Pending` a `Working`, infine ad `Executed` quando è finito.

Inizialmente si utilizzava, al posto di Redis, un dizionario acceduto concorrentemente dai thread, tuttavia questo presupponeva un elevato uso di meccanismi di lock. Si è deciso quindi di passare a Redis, il quale, in quanto database di tipo Key-Value, è molto efficiente, inoltre tutte le operazioni svolte da esso sono atomiche per definizione. Inoltre, per svolgere operazioni complesse in esso, è possibile definire degli script Lua, e richiamarne quindi la loro esecuzione; più avanti se ne mostra un esempio.

Una volta che il job è stato depositato in una coda dal `controllers`, si ritorna immediatamente la risposta HTTP, tale risposta contiene anche lo `WorkID`. E' responsabilità del chiamante fare quindi del *polling* al microservizio per verificare se il task si è concluso. Tali richieste si traducono in una interrogazione a Redis.

Quando si richiede lo stato di un lavoro ed esso è stato completato, viene rimosso da Redis.

La ragione per cui il trasferimento segue questo paradigma non è prestazionale, bensì è per garantire l'integrità dei file sui server. Si pensi ad un semplice esempio, per cui si fanno due richieste a questo microservizio nello stesso momento, ed entrambe richiedono di aggiungere un nuovo client allo stesso server.



Si supponga che vi siano due thread, ciascuno che gestisce una delle due risposte. Essi quindi accedono al file `client-down.sh` sul server, lo modificano aggiungendo informazioni relative solo al proprio client, quindi il primo thread trasferisce il file, immediatamente dopo il secondo trasferisce la propria versione, ed ecco che la modifica fatta dal primo viene sovrascritta.

Un altro problema a cui si può pensare è il seguente: si fa una richiesta di revoca di un certificato, la CRL viene quindi modificata ed inizia la propagazione verso tutti i server. Poi si fa una richiesta di refresh, quindi la si ricrea ed inizia la distribuzione... Qua c'è il problema. Se si suppone che in totale vi siano 100 server e che la seconda richiesta sia arrivata mentre si concludeva il trasferimento verso il 50esimo, a quel punto si modifica la CRL e quella trasferita al 51esimo è diversa dalle precedenti.

Un'esecuzione seriale mette al sicuro da questi problemi. Anche con quando il numero di server si farà potenzialmente alto, un thread è stato ritenuto sufficiente perché i file trasferiti sono sempre testuali e relativamente brevi. L'utilizzo di certificati su curva ellittica ha l'effetto di avere anche tali file di dimensioni contenute. In futuro si può prevedere, dopo un accurato studio, di aumentare il numero di thread a disposizione. Per ciò che concerne il problema appena descritto relativo alla CRL, la stessa modifica alla CRL viene schedulata in maniera seriale, non solo il trasferimento.

Inizialmente tutte le operazioni venivano eseguite da un unico thread, successivamente se ne è aggiunto un secondo.

- `Main_worker_thread` si occupa del trasferimento di nuovi server, nuovi client e, parzialmente, della revoca di un client chiamando lo script `deleteclient.py`. Si noti che questo thread è l'unica flusso di computazione che può modificare i file `client-up.sh` e `client-down.sh`.
- `CRL_worker_thread` è responsabile dell'aggiornamento e della propagazione della CRL ai diversi server. Anche in questo caso, vi è un unico flusso che modifica un file, per cui l'integrità è ancora garantita.

Di seguito si mostra un estratto di `SSHBackgroundWorker`, in particolare si mostrano i due loop eseguiti dai due thread.

```
class SSHBackgroundWorker(object):

    def __init__(self):

        self.__main_queue = queue.Queue()
        self.__main_conn = conns.ConnManager()
        self.__main_worker_thread = threading.Thread(
            target=self.__main_loop_func)

        self.__crl_queue = queue.Queue()
        self.__crl_conn = conns.ConnManager()
        self.__crl_worker_thread = threading.Thread(
            target=self.__crl_loop_func)

        self.__thread_number = 2

        self.__redis_conn = redis_conn.RedisConn()
```

```

        threads_number=self.__thread_number)

def __main_loop_func(self):
    loop = True
    while loop is True:
        work = self.__main_queue.get()
        if work is not None:

            work_type = work['work_type']
            work_id = work['work_id']

            result = None

            if work_type == status.WorkType.transfer_client.value:
                result = self.__transfer_client(work)
            elif work_type ==
↪ status.WorkType.transfer_server.value:
                result = self.__transfer_server(work)
            elif (work_type == status.WorkType.renew_server.value):
                result = self.__renew_server_cert(work)
            elif work_type == status.WorkType.revoke_client.value:
                result = self.__deleteclient(work)

            if result is not None:
                self.__work_post(work_id, result)

        else:
            loop = False

def __crl_loop_func(self):
    loop = True
    while loop is True:
        work = self.__crl_queue.get()
        if work is not None:

            work_type = work['work_type']
            work_id = work['work_id']

            if status.is_crl_str(work_type):
                result = self.__update_crl(work)
                self.__work_post(work_id, result)
            else:
                loop = False

def __work_post(self, work_id, result):

    if result['is_error']:
        if isinstance(result, tuple):
            error_str = str(result[1])
        else:
            error_str = result
        self.__redis_conn.set_work_status(
            work_id, status.WorkStatus.error,
            error_str)

```

Il prossimo codice presentato è uno script Lua che viene eseguito su Redis. E' utilizzato per ritornare, in una sola query a tale database, l'intera lista dei job.

Senza tale script, sarebbe stato necessario fare una prima query per ottenere la lista degli WorkID, poi, per ciascuno di essi, richiedere l'oggetto completo.

```
-- argument passed to the script
-- "true" if the script has to remove
-- completed works.
local remove_if_done = tostring(KEYS[1])

-- list of WorkIDs
local keys = redis.call("KEYS", "*")

local result = {}

-- iterating over the list
for _, key in ipairs(keys) do
    -- get the full object
    local value = redis.call("HGETALL", key)
    local current_work = {}

    -- parsing the result
    for index = 1, #value, 2 do
        local field_name = value[index]
        local field_value = value[index + 1]

        current_work[field_name] = field_value

        -- if have to remove, and it's finished then
        -- delete it
        if remove_if_done == "true" then
            if field_name == "Status" and field_value == "Executed"
↪ then
                redis.call("DEL", key)
            end
        end
        result[key] = current_work
    end

-- encoding the result
return cjson.encode(result)
```

### 6.3.4 Registrazione server

Registare un nuovo server significa che è stata già deployata in MoonCloud una VM dedicata esclusivamente ad essere un VPN server, ed esso è raggiungibile anche dal Docker container su cui si trova il microservizio MoonCloud\_VPN. Viene messa a disposizione una API per questo scopo; i passi per creare un server sono:

1. verificare che la chiave SSH fornita sia valida
2. creare una nuova istanza nel database per il nuovo server
3. creare i certificati ed i file di configurazione
4. registrare nel database le informazioni relative al nuovo certificato

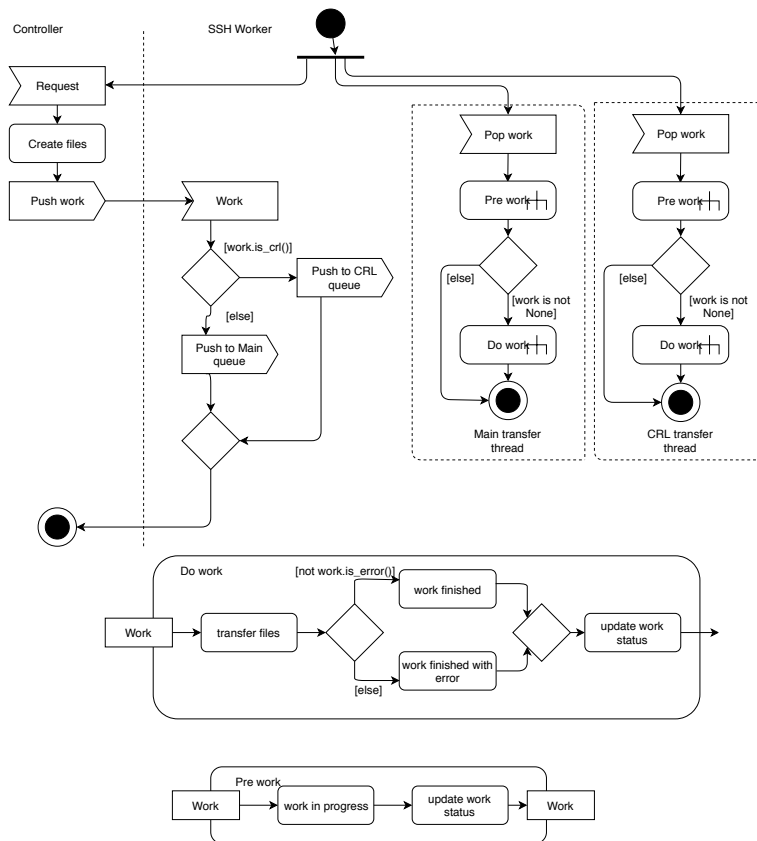


Figura 6.3: Diagramma di attività: trasferimento file. Il presente diagramma UML mostra cosa succede quando è necessario trasferire dei file. Ad una generica richiesta arrivata a `controllers`, esso richiede di aggiungere un lavoro. Si decide quindi su quale coda depositarlo in base al tipo, quindi si ritorna al chiamante. Parallelamente, i due thread di trasferire svolgono il lavoro.

5. mettere sulla coda di trasferimento un nuovo job, ovvero quello di trasferire sul nuovo server tutti i file necessari.

L'estratto di codice che si presenta ora è una versione leggermente modificata di ciò che controllers fa.

```
@transaction.atomic
def create_server(input_data):
    """
    The function used to create a new server.

    :param input_data: dict
    :rtype tuple: (directory-in-wich-files-will-be, work_id,
                  server.id, server.conf)
    """

    is_key_valid = sshclient.is_key_valid(
        private_key=input_data['SSHPrivateKey'],
        is_from_file=False,
        private_key_passphrase=None,
        key_algo=input_data['SSHKeyAlgorithm']
    )

    if not is_key_valid:
        raise paramiko.ssh_exception.SSHException(WRONG_SSH_KEY)

    directory, work_id = misc_ops.create_temp_directory_work_id()

    chacha_supported = is_chacha_supported()

    try:
        server_vpn_net = ipaddress.IPv4Network(
            input_data['VPNVirtualNetID'] +
            '/' + input_data['VPNVirtualMask']
        )

        server = models.Server(
            public_host_name=input_data['publicHostName'],
            public_port=input_data['publicPort'],
            internal_ip=input_data['internalIP'],
            vpn_net=server_vpn_net,
            ssh_username=input_data['SSHUsername'],
            ssh_port=input_data['SSHPort'],
            ssh_private_key=input_data['SSHPrivateKey'],
            ssh_key_algorithm=input_data['SSHKeyAlgorithm']
        )

        create_server_in = {
            'ServerName': input_data['publicHostName'],
            'SupportChaCha': chacha_supported,
            'ServerPort': input_data['publicPort'],
            'VPNNet': {
                'NetID': input_data['VPNVirtualNetID'],
                'Mask': input_data['VPNVirtualMask'],
            }
        }
    }
```

```

# creating the main configuration file.
server_conf = createserver.ServerConf(
    directory,
    create_server_in
)

# dict that contains the paths used by the server.
server_dst_paths = server_conf.get_paths()

server.private_key_path = server_dst_paths['key_path']
server.public_key_path = server_dst_paths['cert_path']

# effectively creates the file.
server_conf.create_file()

cert_res = cert_ops.generate_server_cert(
    directory,
    input_data['publicHostName'], email=input_data['email'])

creation_info = cert_res[0]
keys_info = cert_res[1]

server_cert_paths = creation_info[0]

serial = creation_info[1][0]
issuing_date = creation_info[1][1]
expiration_date = creation_info[1][2]

security_level = keys_info[0]
curve = keys_info[1]

# certificate model
cert = models.CertInfo(
    email=input_data['email'],
    security_level=security_level,
    algorithm=curve,
    certificate_serial=serial,
    issuing_date=issuing_date,
    expiration_date=expiration_date,
    common_name=input_data['publicHostName']
)

cert.save()

server.base_per_client_dir_path = server_dst_paths[
    'base_ccd_path']
server.client_up_script_file_path = server_dst_paths[
    'client_up_path']
server.client_down_script_file_path = server_dst_paths[
    'client_down_path']

server.crl_file_path = server_dst_paths['crl_path']

server.vpn_nic = server_conf.get_device_name()

server.certificate = cert
server.save()

```

```

work = misc_ops.get_create_server_work(
    directory=directory,
    server=server,
    server_cert_paths=server_cert_paths,
    server_conf=server_conf,
    server_dst_paths=server_dst_paths,
    transferCA=input_data['transferCA']
)
global ssh_worker
misc_ops.add_host(ssh_worker, server)

ssh_worker.put_work(work, work_id=work_id)
return (directory, work_id, server.id, server_conf)

except Exception as e:
    __shutil_rmtree(directory)
    raise e

```

Alla fine, la API ritorna al chiamante l'ID del server che lo identifica univocamente nel database ed il `WorkID` relativo al trasferimento dei file. E' responsabilità del chiamante fare ulteriori richieste per verificare lo stato del trasferimento. Per farlo viene messa a disposizione una API specifica.

### 6.3.5 Registrazione client

Una API REST è predisposta alla creazione di un nuovo VPN device client. Tra gli input che occorre fornire, è particolarmente importante specificare l'ID del server a cui il client si conatterà e l'elenco delle reti che il client sarà in grado di raggiungere una volta portato nella rete target. Si compiono i seguenti step:

1. fare un refresh della lista di nomi di dominio da escludere dal mapping
2. creare una nuova istanza nel database per il device client
3. creare i certificati ed i file di configurazione
4. definire il mapping
5. creare il file di script per nftables
6. mettere sulla coda di trasferimento un nuovo job di tipo "Trasferire Client".

Si ritorna al chiamante l'ID del client ed un file zippato codificato in formato base64 che contiene:

- file di configurazione principale per OpenVPN
- certificato e chiave privata del VPN client
- chiave pubblica della CA

Anche in questo caso si mostra un esempio.

```

@transaction.atomic
def create_client(input_data):
    """
    Creates a new client.

    That means:
    - creating config files
    - creating a new pair of cryptographic keys
    - registering a new mapping
    - moving necessary to the server which this client is
      registered to.

    The configuration files are immediately returned as a
    zipped-then-base64ed file, you'll get a work id too, necessary
    to handle the moving of files to the server.
    """
    company = input_data['company']
    identifier = input_data['identifier']

    # first thing to do is to update dns
    dns_worker = dnsworker.DNSWorker()

    event_loop, requests = dns_worker.update_all_no_wait()

    full_name = utils.get_full_name(company, identifier)

    directory, _ = misc_ops.create_temp_directory_work_id()

    chacha_supported = is_chacha_supported()

    try:

        # getting the related server instance
        server = models.Server.objects.get(pk=input_data['idServer'])

        cert_res = cert_ops.generate_client_cert(
            directory, full_name,
            company, input_data['email'])

        creation_info = cert_res[0]
        keys_info = cert_res[1]

        serial = creation_info[1][0]
        issuing_date = creation_info[1][1]
        expiration_date = creation_info[1][2]

        # keys_info = client_openssl.keys_info
        security_level = keys_info[0]
        curve = keys_info[1]

        cert = models.CertInfo(
            email=input_data['email'],
            security_level=security_level,
            algorithm=curve,
            certificate_serial=serial,
            issuing_date=issuing_date,
            expiration_date=expiration_date,

```



```

        common_name=full_name,
    )

    cert.save()

    pair = creation_info[0]

    # creating the client model instance
    client = models.Client(
        # common_name=full_name,
        company=company,
        identifier=identifier,
        certificate=cert
    )

    client.save()

    pair = creation_info[0]

    create_client_in = {
        'SupportChacha': chacha_supported,
        'ClientName': full_name,
        'ServerName': server.public_host_name,
        'ServerPort': server.public_port,
        'ServerCertName': server.certificate.common_name,
    }

    # copying the move-files.sh into the directory
    # in which we currently are.

    # this can be used on the client-side to move OpenVPN
    # files to the correct locations.
    move_script_file_name = os.path.join(directory,
↪ 'move-files.sh')

    shutil.copyfile(settings.MOVE_SCRIPT_FILE_CLIENT,
                    move_script_file_name)

    # we need the ca.crt too
    dest_ca = os.path.abspath(os.path.join(directory, 'ca.crt'))
    shutil.copyfile(settings.ROOT_CA_CERT, dest_ca)

    # getting the mapper instance.
    db = hlddb.HLDb()
    dnmcp = dnmcp4.DNMCP4(database_conn=db)

    nets_to_remap = input_data['internalNets']

    # now waiting the dns resolver
    dns_worker.wait_multi(event_loop, requests)

    mappings = dnmcp.create_mapping(
        server=server,
        client=client,
        old_nets=nets_to_remap)

    add_client_in = {

```

```

        'ServerVirtualIP': server.get_server_vpn_ip_address(),
        'ClientName': full_name,
        'InternalNets': [remapped.old_network for remapped in
↪ mappings],
        'ServerVirtualNICName': server.vpn_nic,
    }

    a_client = addclient.AddClient(
        directory,
        add_client_in
    )

    a_client.create_file()

    work = misc_ops.get_create_client_work(
        directory=directory,
        server=server,
        add_client=a_client,
        client_full_name=full_name
    )

    global ssh_worker
    ssh_worker.put_work(
        work_id=client.id,
        work=work
    )

    c_client = createclient.ClientConf(
        directory,
        create_client_in
    )

    c_client.create_file()

    # passing the mapping to the *Tables handler.
    xptables_remapping = mappings

    xtables_in = {
        'ClientName': full_name,
        'VPNNet': {
            'NetID': server.vpn_net.exploded,
            'Mask': server.vpn_net.netmask.exploded,
        },
        'InternalNets': input_data['internalNets'],
        'Remapping': xptables_remapping
    }

    client_xtables = nftables.NFTables(directory, xtables_in)

    client_xtables.create_file()

    zip_file = zipper.Zipper(
        directory=directory,
        zipfile_name=full_name+'.zip'
    )

    files_to_add = [

```

```

        client_xtables.file_name,
        c_client.file_name,
        pair[0],
        pair[1],
        dest_ca,
        move_script_file_name,
    ]

    for file_to_add in files_to_add:
        zip_file.add_file(file_to_add)

    zip_file.finalize()

    return (directory, zip_file.zip_abs_path, client.id)
except Exception as e:
    __shtuil_rmtree(directory)
    raise e

```

## 6.4 API

In questa sezione si descrivono le API REST messe a disposizione da MoonCloud\_VPN. Input ed output sono descritti senza Specificare l'esatta sintassi, poiché è fuori dallo scopo di questo documenti fornire una documentazione delle API. Tutti gli input e gli output sono in formato JSON.

### Creazione server POST /server

- Descrizione: mediante questa API si creano tutte le configurazioni necessarie per un nuovo VPN server. Tutti i file che servono vengono trasferiti sul server mediante SSH.
- Input: un oggetto con i seguenti campi:
  - nome DNS da inserire nel certificato ed ulteriori informazioni (es: per il campo Email)
  - porta di ascolto OpenVPN
  - NET ID e subnet mask del tunnel VPN
  - credenziali SSH, compresa la porta di ascolto
- Output: l'ID del server assegnato dal database ed il WorkID del lavoro di trasferimento.

### Stato creazione server GET /server/creation/work-id>

- Descrizione: viene utilizzata per controllare se il trasferimento dei file per la creazione di un nuovo server è stato completato.
- Input: nessuno
- Output: un oggetto JSON che riporta lo stato del trasferimento, il tipo di job, in questo caso di tipo TransferServer, il WorkID.

### Dettagli server GET /server/<server-id>

- Descrizione: ritorna tutte le informazioni memorizzate relative ad un certo server.
- Input: nessuno
- Output: oggetto JSON che riporta le seguenti informazioni:
  - ID
  - nome DNS e porta OpenVPN di ascolto
  - indirizzo IP interno mediante il quale è raggiungibile
  - alcuni path della configurazione di OpenVPN
  - NET ID e subnet mask del tunnel VPN
  - informazioni sul certificato, tra cui data di firma, data di scadenza, numero seriale ed algoritmo di firma.

**Rinnovo certificato server** PUT /server/<server-id>/renew

- Descrizione: i certificati hanno una scadenza, ed una volta scaduti non sono più validi e quindi non è più possibile effettuare il collegamento VPN, pertanto è necessario rinnovarli periodicamente.
- Input: nessuno
- Output: il WorkID relativo al trasferimento del certificato e della chiave privata sul server in questione.

**Stato rinnovo certificato server** GET /server/renew/<work-id>

- Descrizione: utilizzata per verificare lo stato del trasferimento verso il server del nuovo certificato e chiave privata.
- Input: nessuno
- Output: stato del trasferimento, tipo, WorkID.

**Creazione client** POST /client

- Descrizione: si chiama questa API per creare un nuovo client. Il micro-servizio crea tutti i file di configurazione, registra un mapping, trasferisce sul server designato i file necessari per poter stabilire un collegamento con esso.
- Input: un oggetto JSON che deve contenere i seguenti campi:
  - ID server a cui il client si connetterà
  - nome dell'azienda cliente a cui il device andrà
  - nome mnemonico con cui identificare il device (es: officel); assieme al nome dell'azienda andrà a comporre il CommonName del certificato
  - email da inserire nel certificato
  - lista di reti raggiungibil dal device

- Output: un oggetto JSON costituito da due campi, l'ID del device client all'interno del database, uno zip file in formato base64 contenente tutti i file che devono essere poi trasferiti sul dispositivo, unitamente ad uno script che automatizza la creazione della struttura di directory assunta da OpenVPN.

**Stato creazione client** GET /client/creation/<client-id>

- Descrizione: analogamente ad ogni altra API relativa ad ottenere lo stato di un trasferimento, questa viene utilizzata per verificare quando i file relativi ad un nuovo client sono stati spostati sul server.
- Input: nessuno a parte la URI in sè; si noti che solo in questo caso il WorkID equivale all'ID del client ritornato al momento della creazione.
- Output: informazioni sullo stato del trasferimento.

**Dettagli client** GET /client/<client-id>

- Descrizione: mediante questa API si ottengono informazioni relative ad un certo VPN client.
- Input: nessuno
- Output: un oggetto JSON con vari campi tra cui:
  - ID
  - nome azienda e nome mnemonico
  - informazioni relative al certificato del client, tra cui data di firma, data di scadenza, algoritmo, livello di sicurezza.

**Rinnovo certificato client** PUT /client/<client-id>/renew

- Descrizione: come per il certificato del server, anche quello del client deve essere sempre valido, altrimenti il server rifiuterà la connessione VPN. Questa API serve a creare un nuovo certificato valido e contemporaneamente a revocare il precedente, distribuendo la nuova CRL a tutti i server.
- Input: nessuno
- Output: lo WorkID del trasferimento ed uno zip file in base64 contenente la nuova chiave privata e certificato.

**Stato rinnovo certificato client** GET /client/renew/<work-id>

- Descrizione: viene utilizzata per verificare lo stato del trasferimento della CRL verso tutti i server in seguito al rinnovo di un certificato client.
- Input: nessuno.
- Output: informazioni sullo stato del trasferimento.

**Revoca client** DELETE /client/<client-id>/

- Descrizione: per numerose ragioni, tra cui il semplice fatto di non servire più, è necessario dover cancellare un VPN client. Tra le azioni coinvolte, oltre a cancellare vari record dal database, occorre revocare il certificato usato da tale client, e propagare la nuova CRL a tutti i server.
- Input: nessuno
- Output: lo `WorkID` relativo alla propagazione della CRL.

**Stato revoca client** GET `/revocation/<work-id>`

- Descrizione: si usa questa API per verificare lo stato del trasferimento della CRL verso i VPN server in seguito alla cancellazione di un client.
- Input: nessuno
- Output: informazioni sullo stato del trasferimento.

**Ottenimento IP mappato** GET `/client/<client-id>/mapping`

- Descrizione: questa API implementa una delle funzionalità più importanti, ovvero fornire la versione mappata di un indirizzo IP originale per un certo VPN client.
- Input: il body della richiesta prevede un oggetto JSON contenente come unico campo l'IP richiesto.
- Output: un oggetto JSON composto dall'IP originale e dall'IP mappato.

**Aggiunta di un'esclusione** POST `/blacklist/add`

- Descrizione: è necessario poter definire delle reti o indirizzi IP (o nomi di dominio tradotti poi in IP) che non devono essere mai utilizzati come reti mappate. Mediante questa API è possibile aggiungere nuove esclusioni alla blacklist.
- Input: un oggetto JSON con diverse combinazioni, tra cui un singolo indirizzo IP, una rete (NET ID e subnet mask), un nome di dominio. Vi è anche un campo `isInternal` per specificare se quell'esclusione è relativa a MoonCloud (ad esempio, si vuole escludere l'indirizzo IP che contiene la repository Docker).
- Output: un oggetto JSON con gli stessi campi dell'input ed in più l'ID assegnato dal database a quella particolare esclusione.

**Elenco blacklist** GET `/blacklist/all`

- Descrizione: si utilizza questa API per visualizzare l'intera blacklist.
- Input: nessuno
- Output: si ritorna un array di oggetti JSON in cui ciascun oggetto rappresenta un'esclusione, tra i vari campi si mostra anche l'ID.

**Aggiornare blacklist** PUT `/blacklist/update`

- **Descrizione:** i nomi di dominio salvati nella blacklist devono essere aggiornati per riflettere i cambiamenti nell'associazione indirizzo IP – nome DNS. Si noti che questo aggiornamento è fatto anche ogni volta che si crea un nuovo client, per cui si può presumere che quest'API non sarà chiamata spesso.
- **Input:** nessuno
- **Output:** un oggetto JSON con i seguenti campi:
  - una lista di dei nomi DNS aggiornati
  - una lista dei nomi DNS che non sono stati aggiornati a causa di errori
  - una descrizione degli errori incontrati durante la risoluzione.

**Vedere specifiche esclusioni** GET /blacklist/dns/internal|external|all

- **Descrizione:** questa API è stata pensata appositamente per visualizzare solo le esclusioni che hanno un nome DNS registrato. Le si possono ritornare tutte ("all"), solo quelle esterne ("external"), oppure solo quelle interne ("internal").
- **Input:** nessuno
- **Output:** una lista di oggetti JSON, per ciascuno si mostra il nome DNS, se interna o no, e l'ID.

**Dettagli esclusione** GET /blacklist/<blacklist-id>

- **Descrizione:** utilizzata per visualizzare una particolare esclusione dalla blacklist, dato il suo ID.
- **Input:** nessuno
- **Output:** un oggetto JSON con tutte le informazioni relative all'esclusione.

**Cancellare esclusione** DELETE /blacklist/<blacklist-id>

- **Descrizione:** mediante quest'API si cancella una particolare esclusione.
- **Input:** nessuno
- **Output:** nessuno

**Specificare il supporto a ChaCha20** PUT /chachasupport/true|false

- **Descrizione:** l'algoritmo di cifratura ChaCha20 ?? è supportato solo in alcune versioni di OpenVPN a seconda di come sia stato compilato. Questa API presuppone che, ad esempio, da un certo punto in poi tale algoritmo sia sempre supportato sui nuovi client e server, e che si possa specificare tale supporto mediante questa API senza dover riavviare il microservizio (poiché il valore di default è specificato nei settaggi).
- **Input:** nessuno
- **Output:** nessuno

**Refresh CRL** PUT /crl

- Descrizione: tra i vari campi che si inseriscono in una CRL vi è “NextUpdate”, ovvero per quando ci si aspetta il prossimo aggiornamento. OpenVPN, nel controllo della CRL, verifica anche il valore di questo campo, e se la CRL non è aggiornata il server rifiuterà nuove connessioni, pertanto è importante mantenerla aggiornata. In questo caso si genera una nuova CRL con gli stessi certificati revocati della precedente, ma con data di NextUpdate posticipata di un anno. Dopo aver generato una nuova CRL, viene distribuita a tutti i server.
- Input: nessun
- Output: lo WorkID relativo alla distribuzione della CRL a tutti i VPN server di MoonCloud.

**Stato refresh CRL** GET /crl/refresh/<work-id>

- Descrizione: si usa l’API in questione per vedere lo stato del trasferimento della CRL ai server in seguito ad un refresh.
- Input: nessuno
- Output: nessuno

**Ultimi aggiornamenti alla CRL** GET /crl

- Descrizione: questa API ritorna una lista degli ultimi dieci aggiornamenti fatti alla CRL. I dati vengono prelevati da un’apposita tabella dal database, la quale è aggiornata solo dopo che la CRL modificata è stata distribuita a tutti i server.
- Input: nessuno
- Output: un array JSON in cui ciascun oggetto rappresenta un aggiornamento. Per ciascun item dell’array i campi sono la data di aggiornamento e la ragione (“refresh” o “revocation” di un client).

**Far ripartire un job** PUT /<work-id>

- Descrizione: vi sono molte ragioni per cui un work di trasferimento può terminare con un errore, tra cui un problema di connettività. Per situazioni come queste, in cui per problemi che poi sono stati risolti non è stato possibile completare un trasferimento, si è pensata questa API. Dopo aver verificato che un trasferimento non è andato a buon fine, e dopo aver risolto l’errore (se possibile), si può far ripartire il job. Esso sarà rimesso sulla coda come lavoro più recente.
- Input: nessuno
- Output: nessuno

**Elenco job** GET /all-works



- Descrizione: oltre a fare del polling per ogni singolo job, può essere più veloce utilizzare questa API per vedere l'elenco di tutti i job in corso, compresi quelli terminati di cui non si è ancora verificato lo stato, quelli in corso, quelli conclusi con errore.
- Input: nessuno
- Output: un array JSON contenente oggetti ciascuno nell'esatto formato ritornato dalle API relative a vedere lo stato di un job. Tutti i job completati e ritornati sono rimossi, pertanto non sarà più possibile vedere il loro stato in un secondo momento.



# Bibliografia

- [1] M. Anisetti, C. A. Ardagna, E. Damiani, and F. Gaudenzi, "A semi-automatic and trustworthy scheme for continuous cloud service certification," *IEEE TRANSACTIONS ON SERVICES COMPUTING*, 2017.
- [2] M. Anisetti, C. A. Ardagna, E. Damiani, N. El Ioini, and F. Gaudenzi, "Modeling time, probability, and configuration constraints for continuous cloud service certification," *COMPUTERS & SECURITY*, vol. 72, pp. 234–254, 2018.
- [3] OpenVPN, "Faq general," 0.
- [4] Spiegel, "Intro to the vpn exploitation process," 0.
- [5] Y. Rekhter, R. G. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear, "Address allocation for private internets," BCP 5, RFC Editor, February 1996. <http://www.rfc-editor.org/rfc/rfc1918.txt>.
- [6] T. Jonsson, "Latency and throughput comparison between iptables and nftables at different frame and ruleset sizes," 0.
- [7] OpenVPN, "Are there any known security vulnerabilities with openvpn?," 0.
- [8] L. Daniel, "Inferring openvpn state machines using protocol state fuzzing," 0.
- [9] T. Novickis, "Protocol state fuzzing of an openvpn," 0.
- [10] K. Bhargavan and G. Leurent, "On the practical (in-)security of 64-bit block ciphers – collision attacks on http over tls and openvpn," 0.