LES GUIDES DE
GNU LINUX MAGAZINE / FRANCE
HORS-SÉRIE N°92
France MÉTRO. : 12.90 € — CH : 18,00 CHF — BEL/PORT.CONT : 13,90 € — DOM TOM : 13,90 € — CAN : 18,00 $ CAD

VPN

LE GUIDE INDISPENSABLE POUR INSTALLER ET CONFIGURER VOTRE VPN

DÉBUTEZ avec le protocole TCP/IP, les nouveautés d'IPv6 et le fonctionnement d'un VPN

OPENVPN Configurez-le finement et installez des clients y compris sur des systèmes non libres

ALLEZ PLUS LOIN en développant vos plugins OpenVPN et en y intégrant un firewall

IPSEC Apprenez à configurer l'« autre » solution de tunneling

Édité par Les Éditions Diamond
L 15066 - 92 H - F: 12,90 € - RD
www.ed-diamond.com

Retrouvez toutes nos publications
LES ÉDITIONS DIAMOND
sur **www.ed-diamond.com**

GNU/Linux Magazine Hors-Série
est édité par **Les Éditions Diamond**

10, Place de la Cathédrale - 68000 Colmar - France

Tél. : 03 67 10 00 20 / **Fax :** 03 67 10 00 21

E-mail : cial@ed-diamond.com
lecteurs@gnulinuxmag.com

Service commercial : abo@gnulinuxmag.com

Sites : https://www.gnulinuxmag.com
https://www.ed-diamond.com

Directeur de publication : Arnaud Metzler

Chef des rédactions : Denis Bodor

Rédacteur en chef : Tristan Colombo

Responsable Service Infographie : Kathrin Scali

Responsable publicité : Tél. : 03 67 10 00 27

Service abonnement : Tél. : 03 67 10 00 20

Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

Distribution France :
(uniquement pour les dépositaires de presse)

MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04

Service des ventes :
Distri-médias : Tél. : 05 34 52 34 01

IMPRIMÉ en Allemagne - PRINTED in Germany

Dépôt légal : A parution

N° ISSN : 0183-0864

Commission Paritaire : K78 976

Périodicité : Bimestrielle

Prix de vente : 12,90 Euros

Les articles non signés contenus dans ce numéro ont été rédigés par les membres de l'équipe rédactionnelle des Éditions Diamond.

PRÉFACE

Au cours des trois dernières décennies, la progression de l'informatique et surtout la nécessité grandissante de devoir garantir une interconnexion des machines ont donné lieu à de nombreuses évolutions logicielles comme matérielles. Les premiers réseaux locaux étaient relativement lents ; avec 1 Mbps pour du 3+Open, 4Mbps pour du Token-Ring voire 10Mbps pour de l'Ethernet, nous sommes bien loin des 10Gbps qui commencent à être disponibles aux particuliers. De plus ces réseaux transportaient simultanément plusieurs protocoles, **NetBIOS**, **IPX/SPX** ou **TCP/IP** pour ne nommer que les plus répandus. À cette époque, le rôle des administrateurs systèmes était surtout de faire en sorte que tout cela fonctionne bien en local et le moins mal possible quand il fallait interconnecter des bâtiments ; la sécurité des données ne venait qu'en dernière position, et encore... Depuis, tout ceci a progressé et les priorités ont bien changé. Les professionnels ont depuis longtemps mis en place des tunnels chiffrés pour interconnecter les plateformes et pour relier les postes itinérants aux services centraux.

Ce qui, dans un monde parfait, aurait dû rester dans le domaine professionnel s'est vu inviter chez de plus en plus de particuliers suite aux révélations d'Edward Snowden ou à la mise en place d'Hadopi ;-) Maintenant, tout un chacun qui ne souhaite pas prendre le risque de voir sa vie privée dévoilée au grand jour ou servir de monnaie d'échange pour les géants d'Internet se voit contraint de chiffrer ses données tant sur ses disques durs que sur ses câbles réseau et je ne parle même pas des transmissions sans fil. Cependant, si le chiffrement des disques se fait assez facilement, monter un tunnel chiffré fiable entre son domicile et ses serveurs en l'air ou ses appareils mobiles reste difficile à appréhender pour le commun des mortels, même versé dans les arcanes de l'informatique.

Le but de ce guide est de vous proposer deux moyens fiables et éprouvés de parvenir à vos fins : **IPSec** et **OpenVPN**. Le premier pourrait être décrit comme la solution officielle, reposant sur des protocoles bien documentés dans des RFCs, mais tellement omnipotent qu'il en devient presque anxiogène, du moins à première vue. Le second ressemble plus à la solution du geek, c'est-à-dire facile à mettre en place dans une version basique, mais fonctionnelle et possédant d'énormes capacités d'amélioration, ayant cependant le défaut d'utiliser un protocole assez peu documenté dès que l'on désire creuser profond. C'est pour ces raisons qu'OpenVPN sera notre principal « client » que nous décortiquerons autant que faire se peut. IPSec sera aussi présenté et documenté bien entendu ne serait-ce que pour montrer que sa mise en œuvre n'est pas si compliquée que cela. De plus, c'est le seul à permettre une intégration simple avec le monde des routeurs propriétaires.

Donc retroussez vos manches, faites chauffer les VMs, il va y avoir du monde dans les tuyaux...

Cédric PELLERIN

Sommaire

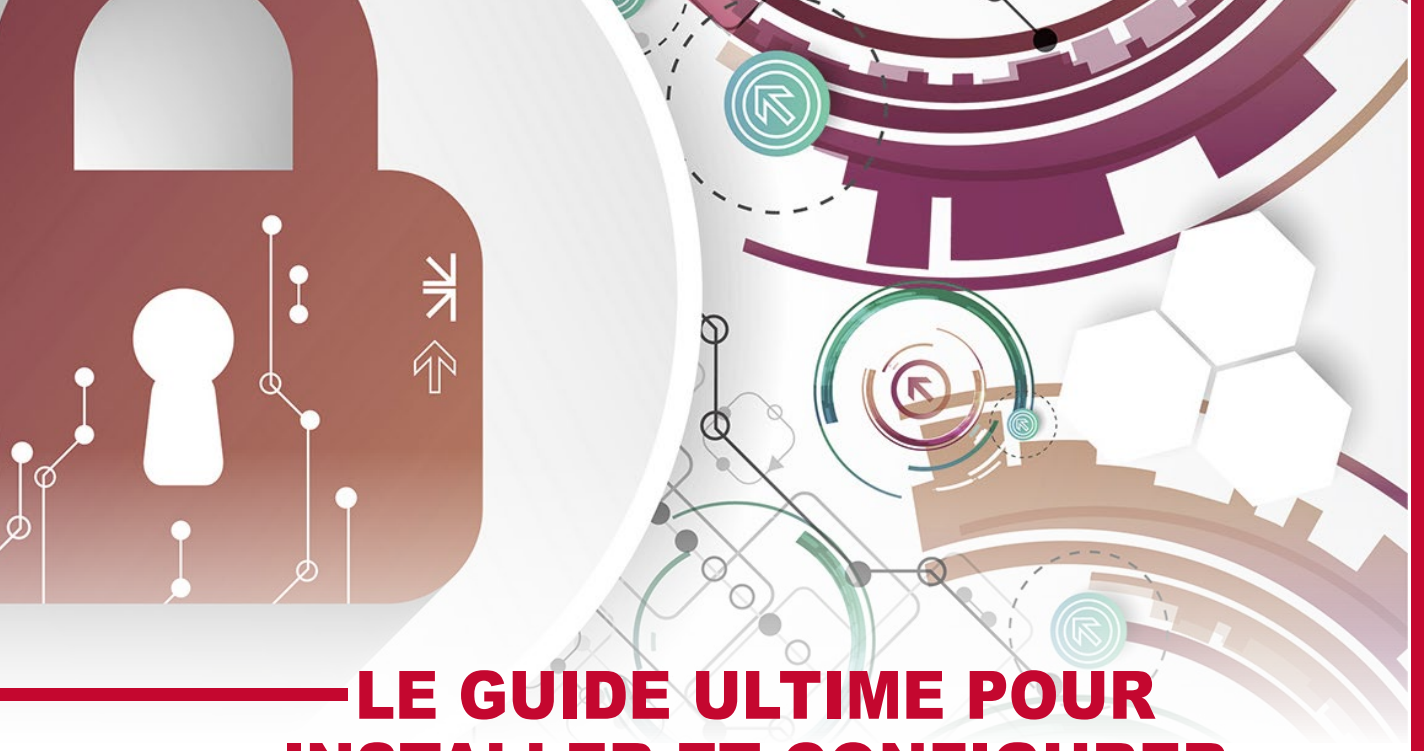
GNU/Linux Magazine
Hors-Série
N°92

DÉBUTEZ

avec le protocole TCP/IP, les nouveautés d'IPv6 et le fonctionnement d'un VPN

OPENVPN

Configurez-le finement et installez des clients y compris sur des systèmes non libres



LE GUIDE ULTIME POUR INSTALLER ET CONFIGURER VOTRE VPN

IPSEC

Apprenez à configurer l'« autre » solution de tunneling

ALLEZ PLUS LOIN AVEC OPENVPN

en développant vos plugins et en intégrant un firewall à OpenVPN

DÉBUTEZ

1 DÉBUTEZ

À découvrir dans cette partie...

Présentation de TCP/IP

Pour comprendre comment fonctionne un VPN, nous allons commencer par partir des éléments les plus bas du fonctionnement d'un réseau (plus particulièrement le transfert des données) et aborder TCP/IP, IPv4 et les nouveautés apportées par IPv6. p. 08

À la découverte des VPNs

Les données transitant par un VPN ne doivent pas pouvoir être altérées ni lues par un tiers. Nous allons découvrir dans cet article les bases du fonctionnement d'OpenVPN et d'IPSec. p. 20

Comment choisir son VPN ?

Voici venu le moment de faire votre choix : quel VPN allez-vous utiliser ? Pour vous aider dans cette démarche, cet article résume les points forts et les points faibles des principaux protocoles. p. 30

PRÉSENTATION DE TCP/IP

Cédric PELLERIN

Quand on désire bien comprendre comment fonctionne quelque chose, il est souvent bon de descendre aussi bas que possible afin de ne laisser échapper aucun détail. Nous n'irons pas jusque-là dans cet article, mais nous allons tracer le début de la route.

1. TCP/IP V4

Toute personne ou presque qui utilise un ordinateur de nos jours a entendu parler de l' « adresse IP ». Parmi ces personnes, certaines sont au courant qu'il s'agit d'une série de quatre octets séparés par un point. Mais au-delà, c'est le grand trou noir. Mettons donc un peu de lumière là-dedans et examinons tout ça.

1.1 Un peu d'histoire

Élaboré depuis 1973 à l'université de Stanford, **TCP/IP** subit ses premiers tests grandeur réelle en 1975 en transportant des données entre Stanford et l'University College London (UCL). La première interconnexion de réseaux eut lieu en 1977 entre les USA, l'Angleterre et la Norvège. De nombreuses retouches furent apportées au protocole jusqu'au 1er janvier 1983, date officielle de sortie de la version définitive. En 1982, il fut adopté par les militaires américains avant de devenir en 1985 le protocole officiel de ce qui allait devenir Internet.

1.2 Un protocole en « couches »

L'idée à la base de ce découpage est que chaque couche fasse uniquement ce pour quoi elle a été conçue et ensuite passe la main à la couche immédiatement supérieure (en réception) ou inférieure (à l'émission) comme montré en figure 1.

Dans le cas de TCP/IP, nous pouvons énumérer les cinq couches suivantes :

Niveau	Nom	Exemples	Interconnexion via
5	Application	HTTP, FTP, Telnet...	
4	Transport	TCP, UDP...	N° de port réseau
3	Réseau	IP	Adresse IP
2	Liaison	Ethernet, Token Ring...	Adresse MAC
1	Physique	Ligne téléphonique, ADSL...	

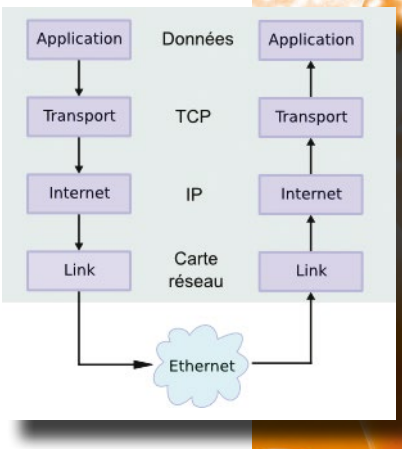


Fig. 1 : Comment les données traversent les couches du modèle OSI.

De la couche **Application** proviennent les données à transférer et ensuite chaque couche rajoute son en-tête avec ses informations propres (par exemple, comme on peut le voir en figure 2, l'adresse IP de destination est l'une de ces informations rajoutées par la couche **Réseau**).

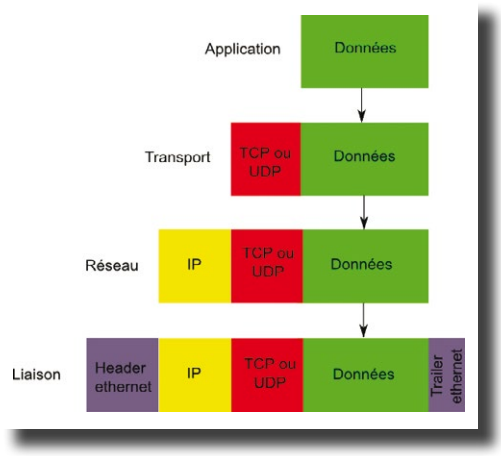


Fig. 2 : Mise en forme des données.

Une fois toutes les couches descendues, la carte réseau transforme tout ce petit monde en impulsions électriques sur le câble réseau jusqu'à la carte réseau de destination. Les couches sont alors remontées les unes après les autres, les en-têtes enlevés après vérification et les données parviennent à la couche **Application** où elles sont traitées.

Le protocole connu sous le nom de TCP/IP est en fait une suite de protocoles qui ont chacun leur utilité et qui interviennent en général au niveau 3 ou 4. Parmi ceux-ci on trouve **ICMP** (pour les **ping** entre autres), les protocoles de routage comme **BGP** ou **OSPF**, **ARP** pour *Address Resolution Protocol* et de nombreux autres.

1.3 Encapsulation et en-têtes

Comme nous venons de le voir, au niveau liaison le paquet complet est composé d'un genre de petit train composé par :

- le *header* Ethernet en tête ;
- le *header* IP ;
- le *header* TCP ou UDP ;
- les données « utiles » ;
- le *trailer* Ethernet.

La partie intéressante de tout cela in fine ce sont les données. Cependant pour les transmettre sans erreur, il faut leur ajouter un genre de métadonnées indiquant entre autres d'où elles viennent et où elles vont, un peu comme une enveloppe (*header*) dans laquelle vous mettez la lettre (données). En TCP/IP c'est un petit peu plus compliqué, car chaque couche rajoute son propre *header*, un peu comme les sur-enveloppes parfois utilisées pour les inscriptions en fac.

Nous en sommes donc à la couche **Transport** qui a ajouté le *header* TCP aux données. Nous continuons à descendre les couches et nous passons par la couche **Réseau** qui va, elle, considérer le bloc « header TCP » + données comme son *payload* et y rajouter son en-tête IP. Dans le jargon réseau, le *payload* est en gros tout ce qui n'est pas *header* et *trailer* (quand il existe) du protocole en cours dans un paquet. Puis nous finissons par la couche **Liaison** qui va finir en ajoutant le *header* et le *trailer* Ethernet.

En résumé, les données sont encapsulées dans une trame TCP avec ajout d'un en-tête TCP. La trame TCP est encapsulée dans une trame IP avec ajout d'un en-tête IP. La trame IP est encapsulée dans une trame Ethernet avec ajout d'un *header* et d'un *trailer* Ethernet.

À propos d'en-tête, regardons d'un peu plus près comment sont constitués les *headers* TCP et IP, on verra ensuite les différences, quand elles existent, avec IPv6.

1.3.1 En-tête TCP

Une « trame » ou « segment » TCP est constitué du *header* et des données. Le *header* est constitué de 20 octets auxquels on peut rajouter des options. Il se présente sous la forme suivante (voir figure 3) :

Offsets Octet		0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size														
16	128	Checksum																Urgent pointer (if URG set)															
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) ...																															

Fig. 3 : En-tête TCP.

- « Source port », « Destination port » : On retrouve nos deux ports de source et de destination, 16 bits chacun.
- « Sequence number » : le numéro de séquence. Cette valeur sur 32 bits sert lors d'un transfert de données (*flag* **SYN** à **0**). Dans ce cas, il contient le numéro du premier octet de la zone de données du segment. Un segment transporte en général plusieurs octets de données qui sont tous numérotés. La zone « Sequence number » contient le numéro du premier de ces octets. La valeur dans cette zone n'est pas remise à zéro à chaque début de transmission. Dans un segment numéro zéro, la zone « Sequence Number »

peut parfaitement contenir un nombre nettement supérieur. Ce qui va compter pour déterminer le bon numéro de séquence sera la différence entre la valeur en cours et celle du segment numéro zéro.

- « Acknowledgement number » : contient le numéro du premier octet de données attendu dans la réponse. Le même principe de relativité que pour la zone précédente est appliqué. Cela semble compliqué, ça l'est un peu, mais on va y revenir dans la suite de l'exemple.
- « Data offset » : 4 bits qui permettent de décaler le début des données par rapport à la fin normale de l'en-tête afin d'y positionner des options supplémentaires. Il y est indiqué le nombre de mots de 32 bits qui composent l'en-tête et les éventuelles options.
- « Flags » sur les 12 bits restants dont voici la signification :
  - **Reserved** : réservé, toujours à **0** ;
  - **NS** (souvent noté **ECN**), **CWR** et **ECE** servent à gérer les congestions. Il s'agit de moments où le lien réseau est saturé et il faut que les stations soit ralentissent, soit bloquent pour un temps les nouvelles connexions. Afin que cela se fasse en bonne harmonie, les RFC 3168 et 3540 ont rajouté ces trois *flags*.
  - **URG** (« URGent ») à **1** signale que le champ « Urgent pointer » est significatif ;
  - **ACK** (« ACKnowledge ») à **1** signale que le champ « Acknowledgement » doit être pris en considération. On verra que c'est le cas dans 95 % des échanges ;
  - **PSH** (« PuSH ») à **1** demande explicitement à ce que les données en cache soient envoyées à l'application. En gros, on vide le cache de données vers la couche 5 ;
  - **RST** (« ReSeT ») à **1** demande un reset de connexions ;
  - **SYN** (« SYNchronize ») est utilisé pour démarrer une connexion ;
  - **FIN** (« FINalize ») sert à terminer une connexion.
- « Window Size » sur 16 bits permet de changer la taille de la fenêtre de réception. En théorie, à chaque segment reçu, le récepteur doit envoyer un ACK à l'émetteur. Cela sécurise la connexion, mais ralentit le débit. Dans un souci de pouvoir augmenter la vitesse, on peut avec ce champ signifier à l'émetteur qu'il ne recevra un ACK que tous les N octets de donnée. Les 16 bits permettent de monter à une fenêtre de 64Kio. Pour augmenter encore cette fenêtre, on peut positionner une option – dans la fameuse zone optionnelle entre l'offset 160 et le début des données – afin de multiplier la taille de cette fenêtre et de l'augmenter jusqu'à 1Gio. Cette action peut avoir un effet de bord non nul. En effet, l'émetteur va envoyer des octets jusqu'à atteindre la taille de cette fenêtre afin de recevoir un ACK. S'il a besoin d'envoyer 312 octets et que la fenêtre est à 32Kio, il va devoir envoyer 32455 octets de « bourrage ». Il faut donc utiliser cette possibilité avec modération.
- « Checksum » : Comme son nom l'indique, il permet d'envoyer une somme de contrôle concernant l'en-tête et les données.
- « Urgent » permet de désigner le début d'une zone de données urgentes – si le *flag* URG est mis à **1** – dans la zone de données du segment. Ce pointeur est un offset par rapport au numéro de séquence.

Enfin nous avons éventuellement la zone des options qui sera terminée par un bourrage à coup de zéros afin de finir sur des frontières de 32 bits.

Voilà pour l'en-tête TCP. Passons maintenant à IP.

1.3.2 En-tête IPv4

L'en-tête IP ressemble pas mal à l'en-tête TCP (voir figure 4, page suivante).

Il tient lui aussi sur 20 octets, plus un champ « Options » facultatif. Les divers champs sont les suivants :

- « Version » : Version d'IP donc 4 en général. L'en-tête IPv6 est différent.

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags			Fragment Offset											
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															

Fig. 4 : En-tête IP.

- « IHL » pour *Internet Header Length*. Indique la taille du *header* en mots de 32 bits et permet de rajouter les options. Même mode de fonctionnement que le champ « Data offset » du *header* TCP.
- « DSCP » pour *Differentiated Services Code Point* : Zone utilisée par les applications ayant besoin de temps réel, comme la VoIP par exemple. Pour plus d'informations, voir la RFC 2474.
- « ECN » pour *Explicit Congestion Notification*. Permet de gérer les problèmes de congestion aussi au niveau IP. Les deux interlocuteurs doivent être capables de l'utiliser et d'accord pour le faire.
- « Total Length » indique la taille totale du paquet, *header* plus données. Sachant qu'un en-tête fait minimum 20 octets, la taille maximum des données en IP sera de 65515 octets, cependant la plus petite taille qui doit être supportée par un équipement est de 576 octets (raisons historiques), *header* compris. Cet aspect des choses peut mener à des besoins de fragmenter les datagrammes IP afin qu'ils passent les routeurs.
- « Identification » permet d'identifier de façon univoque un paquet IP. À chaque envoi de datagramme en provenance du même émetteur, ce champ sera incrémenté. Il est très utile lors du réassemblage des paquets fragmentés.
- « Flags » sur trois bits :
  - Bit **0** non utilisé, toujours à **0** ;
  - Bit **1** : Don't fragment (**DF**) ;
  - Bit **2** : More fragments (**MF**) sous-entendu « are coming ».

Si le bit **DF** est mis à **1** et qu'un équipement requiert une fragmentation, le paquet est purement et simplement passé à la trappe. Ce *flag* est utilisé principalement par des outils comme **traceroute**.

Pour les paquets non fragmentés, le bit **MF** est à zéro ainsi que le champ « Fragment Offset ». Pour les paquets fragmentés, tous les fragments ont le bit **MF** à **1** sauf le dernier. La différence est faite grâce au champ « Fragment Offset » qui n'est pas à zéro dans ce cas.

- « Fragment Offset » indique l'offset du fragment en cours par rapport au premier octet du premier fragment.
- « TTL » pour *Time To Live*. Ce champ permet d'éviter qu'un paquet mal routé tourne sur le réseau pour l'éternité. La valeur du TTL est décrémentée par chaque routeur que le paquet traverse. Quand la valeur tombe à zéro le paquet est *droppé* et le routeur avertit l'émetteur via un message ICMP de type « Time Exceeded ».
- « Protocol » : Dans cette zone, on trouve le numéro du protocole sous-jacent tel que défini par l' « Internet Assigned Numbers Authority ». Par exemple, TCP aura le numéro **0x06**, ICMP le numéro **0x01**, etc. La liste complète est disponible sur [1].
- « Header Checksum » : somme de contrôle du *header* qui est contrôlée et recalculée par chaque routeur. En effet, le fait de décrémenter le TTL change le *checksum*. Si la vérification est négative, le paquet est rejeté.
- « Source Address » et « Destination Address » sont les adresses IP de l'émetteur et du récepteur. Elles sont susceptibles de changer en cas de translation d'adresse par un routeur.

Ensuite viennent les options, si elles existent, puis les données. Ces dernières sont constituées dans le cadre de notre exemple, par l'en-tête TCP plus les données réelles. On a là la première vraie encapsulation.

Nous pourrions continuer avec la description de la couche Ethernet, mais ce serait aller bien loin pour une simple introduction. Nous renvoyons les lecteurs intéressés à *Open Silicium n° 13* dans lequel tout est expliqué en détail.

2. LE PROTOCOLE DU FUTUR PROCHE : IPV6

Depuis quelques années, la raréfaction des adresses publiques IPv4 incite les organismes officiels comme le **RIPE** à tirer la sonnette d'alarme et à avertir tout le monde que la migration vers IPv6 va devoir s'effectuer rapidement. Cette nouvelle norme introduit tout un tas d'éléments nouveaux dont nous allons passer les principaux en revue ici même.

2.1 Les améliorations apportées par IPv6

2.1.1 Plusieurs adresses IP par interfaces

Cela est déjà possible en IPv4, mais devient une généralité en IPv6. Le support natif de ces multiples adresses permet de faciliter les migrations, les renumérotations de réseau et l'hébergement de plusieurs services sur une seule et même machine, chaque service pouvant avoir sa propre adresse IP publique.

2.1.2 Adresse de lien local unique

Les adresses de lien local permettent l'auto-configuration des adresses réseau de toutes les machines. Ces adresses ne sont pas routables et permettent la découverte de l'environnement réseau immédiat.

2.1.3 Des en-têtes IP simplifiés

Comparativement aux en-têtes IPv4, les en-têtes IPv6 sont nettement plus simples : leur taille est fixe et alignée sur 64 bits afin de faciliter le traitement sur des processeurs modernes. Les traitements supplémentaires reposent sur des extensions qui peuvent être vues comme des protocoles de niveau « trois et demi ».

Les champs liés à la fonction de fragmentation ont disparu, en effet un algorithme de découverte du MTU de bout en bout (RFC 1191) doit éviter toute fragmentation. Si celle-ci devait être mise en place pour une raison ou une autre, une extension est prévue. Cela soulage fortement les routeurs qui se contentent de transmettre les paquets. Si le paquet s'avérait malgré tout trop gros pour un routeur, celui-ci renvoie simplement un message ICMPv6 de type « Packet Too Big » et l'émetteur devra redécouper le paquet et le renvoyer.

L'en-tête IPv6 est constituée ainsi :

Version (4 bits)	Classe de trafic (8 bits)	Indicateur de flux (20 bits)
Longueur des données (16 bits)	Prochain en-tête (8 bits)	Sauts max (8 bits)
Adresse source (128 bits)		
Adresse destination (128 bits)		

- Le champ « Version » sera toujours à **6**, car on est en IPv6 ;
- Le champ « Classe de Trafic » définit la priorité du datagramme ;
- Le champ « Indicateur de flux » est à la disposition de l'émetteur pour nommer des séquences de paquets auxquels un traitement spécial doit être réservé (QoS, temps « réel », etc.) ;
- La taille des données ne concerne que les données et non plus les données + l'en-tête comme en IPv4. Cependant si des extensions sont utilisées, elles seront considérées comme des données et donc incluses dans le calcul de la longueur ;

- Le champ « Prochain en-tête » identifie le type de données qui suit l'en-tête. Les valeurs utilisées sont identiques à celles du champ « Protocole » en IPv4 auxquelles ont été rajoutées des valeurs spécifiques aux extensions IPv6 ;
- Le champ « TTL » d'IPv4 a été renommé « Hop Limit » – « Sauts max » ici en français – et fonctionne de la même manière. Il faut noter qu'en cas d'usage d'une adresse non routable, ce champ est à **255** et non à **1** comme en IPv4. Si par hasard un routeur laissait passer par erreur, le suivant repèrerait aussitôt une adresse de lien local avec un Hop Limit à **254** et cette incohérence suffirait pour que le paquet soit ignoré.
- Les deux champs d'adresse source et destination sont trop évidents pour qu'on en parle plus longuement.

Dans le cas où des extensions sont présentes, le premier octet de chacune d'elle indique le « Prochain en-tête » un peu à la manière d'une liste chaînée.

2.1.4 Autoconfiguration stateless

L'un des aspects frappants d'IPv6 est sa capacité à insérer les machines dans le réseau sans aucune configuration manuelle ni serveur DHCP. Certes dans ce cas la configuration est minimale, mais elle est parfaitement fonctionnelle.

2.2 Les adresses IP

2.2.1 Notions de base

Ce qui frappe le plus quand on regarde IPv6 pour la première fois est le format des adresses IP. Les adresses IPv4 étaient constituées de 32 bits – 4294967296 adresses possibles – notées par quatre octets écrits en décimal séparés par un point. En IPv6, nous sommes passés à 128 bits – soit 340282366920938463463374607431768211456 (environ 3,4 x 10^{38}) adresses possibles, ce qui permettrait de donner une adresse IP à plusieurs centaines de machines au mètre carré sur toute la planète – qui sont notées en huit mots de 16 bits en hexadécimal séparés par le signe deux-points, par exemple `fe80:cafe:0000:dead:0000:0000:acf1:0001`. Afin de condenser un peu l'écriture il a été convenu que les zéros non significatifs peuvent être omis à condition de ne pas en enlever plus de trois. Cela donne pour l'adresse ci-dessus : `fe80:cafe:0:dead:0:0:acf1:1`.

Quand un ou plusieurs mots successifs sont à zéro, il est possible de les omettre totalement et de remplacer la suite de zéros par une paire de deux-points (`::`) mais une fois seulement. On peut donc condenser l'adresse ci-dessus en `fe80:cafe:0:dead::acf1:1`, mais pas en `fe80:cafe::dead::acf1:1`, car il est impossible dans ce cas de savoir combien de zéros sont remplacés par les deux doubles deux-points.

En IPv4, nous avons certaines plages d'adresses réservées aux usages privés, une autre pour le *loopback*, une pour le *failover* DHCP, etc. En IPv6, toutes les plages sont affectées ou il est prévu qu'elles le soient un jour. Il n'y a plus de notions d'adresses privées ni de NAT, car le principe d'IPv6 est justement de donner au moins une adresse publique à chaque équipement. Pour le moment, les plages à retenir sont les suivantes :

- `::1` (`0000:0000:0000:0000:0000:0000:0000:0001`) est l'adresse de *loopback* équivalente au `127.0.0.1` d'IPv4 ;
- `2000::/3` Global unicast. Ce sont les adresses routables sur Internet, l'équivalent des adresses publiques en IPv4 ;
- `fe80::/10` est une nouveauté d'IPv6 nommée « adresse de lien local » qui permet d'autoconfigurer les interfaces réseau dès le démarrage de la machine et ce, même sans lien réseau fonctionnel. Nous allons y revenir très vite.
- `ff00::/8` regroupe les adresses de *multicast*. Il est à noter que le *broadcast* n'existe plus en IPv6, il est remplacé par du *multicast*, qui sature moins les réseaux s'ils sont équipés de *switches*, ce qui est le cas maintenant de 99 % des réseaux.

IPv6 connaît trois types d'adresses :

- Adresses *multicast* qui désignent un groupe d'interfaces. Lorsqu'un paquet est à destination d'une telle adresse, il est remis à tous les membres du groupe.
- Adresses *unicast* qui désignent simplement une interface unique. En clair, on s'adresse à un seul interlocuteur, il s'agit donc de l'adresse IP d'une station par exemple.
- Adresses *anycast* qui sont très proches du *multicast* à cette différence que le paquet est remis à un seul des membres du groupe, en général le plus proche au sens métrique du terme.

De nombreuses autres plages sont réservées par l'IETF, d'autres sont encore non attribuées, mais l'espace disponible avec le `2000::/3` (de `2000::` à `3fff::`) laisse une sérieuse marge de manœuvre.

Contrairement à l'habitude en IPv4, une carte réseau supportera très souvent plusieurs adresses IPv6 en même temps. En général, nous aurons l'adresse de lien local en même temps qu'une adresse routable (ou *global unicast address*). Il ne faut donc pas s'en étonner.

La question qui se pose maintenant concerne les ports TCP. En effet, quand on veut se connecter à un port TCP on sépare l'adresse IP du numéro de port par le signe deux-points. Comment fait-on en IPv6 ? La réponse est « simple », on encadre l'adresse IPv6 avec des crochets. Exemples :

Terminal

```
$ telnet [fe80:cafe:0:dead::acf1:1]:25
$ wget [fe80:cafe:0:dead::acf1:1]:8080/mapage.html
```

2.2.2 Les sous-réseaux

La notion de sous-réseaux est familière à tout administrateur système qui se respecte. Elle consiste à redécouper la plage d'adresses afin de libérer quelques bits de plus pour la numérotation des réseaux ou pour celle des stations en fonction des besoins. Pour mémoire, dans un masque de sous-réseau, les bits à `1` définissent la plage réservée aux réseaux et ceux à `0` la plage réservée aux stations. Un réseau en `/80` est donc plus petit (comprendre moins de stations) qu'un autre en `/64`. En IPv6, le *subnetting* existe aussi bien entendu, mais les mécanismes de configuration automatiques évolués (DHCPv6 et *Router Advertisement* entre autres) sur lesquels nous reviendrons en détail bientôt ne fonctionneront pas pour un masque de sous-réseau supérieur à `/64`.

En clair, les bonnes pratiques spécifient qu'une adresse IPv6 est constituée de 64 bits pour le préfixe plus les divers *subnettings* et de 64 bits pour les stations. La gestion des sous-réseaux doit donc se faire dans les derniers bits des 64 premiers. C'est pour cela que tout bon *provider* fournira un `/48` ou au pire un `/56` afin de nous permettre de gérer nos propres sous-réseaux. Un cas particulier est le fournisseur **Free** qui fournit officiellement un `/64` à ses abonnés. L'idée derrière cette façon de faire est de simplifier le travail des clients en laissant à la freebox le soin de servir de routeur en IPv6 même lorsqu'elle est configurée en pont. On en reparlera plus loin plus en détail.

Étant donné le nombre d'octets d'une adresse IPv6, les masques de sous-réseaux sont notés uniquement par la notation CIDR (*Classless Inter-Domain Routing*) déjà bien utilisée en IPv4.

2.2.3 Le routage

Lors d'une étude superficielle d'IPv6 on est amené à se dire qu'il va falloir mettre un *firewall* sur toutes les machines, car elles sont accessibles directement depuis Internet. C'est oublier un peu vite que la sortie sur le grand *ternet* se fait via un routeur unique – en général, le

routeur par défaut – et que pour accéder à une machine il faut pouvoir lui envoyer des paquets, mais aussi en recevoir d'elle. D'autre part, connaître l'adresse IP d'une station ou d'un serveur c'est bien, mais s'il n'est pas possible d'en déduire l'adresse MAC correspondante personne n'y accèdera jamais. Or seul le routeur local dispose de la table de transcription IP <-> MAC indispensable à Ethernet. En conséquence, le *firewall* sera posé sur le routeur connecté à Internet et ce sera largement suffisant.

2.2.4 Le NDP ou la récupération des adresses MAC sans ARP

Le protocole ARP a été banni d'IPv6 au profit de NDP qui signifie *Neighbor Discovery Protocol* (RFC 2461). Au lieu d'utiliser un protocole de plus comme en IPv4, IPv6 utilise ICMPv6 en *multicast* pour envoyer une requête du type *Neighbor Solicitation*. L'adresse *multicast* utilisée est au format `ff02::1:ff00:0:<24 derniers bits de l'adresse IPv6 cible>` et s'appelle **solicited-nodes**.

Comme toutes les machines sont abonnées à ce groupe *multicast*, les quelques machines du réseau dont les 24 derniers bits sont identiques recevront ce message et y répondront par un message ICMPv6 de type *Neighbor Advertisement* qui permettra au demandeur de déduire l'adresse MAC recherchée de l'adresse source et ainsi de compléter sa table des *Neighbors*.

2.3 Modes de configuration

Il existe deux modes de configuration automatique d'un réseau en IPv6. Le premier mode nommé *stateless* ou SLAAC pour *StateLess Address AutoConfiguration* permet d'obtenir un réseau fonctionnel sans aucune intervention. La création des adresses est automatique, aussi bien pour les adresses de lien local (adresses non routables, repérables à leur préfixe en `fe80::/10`) que pour les adresses routables. L'établissement de ces dernières fait appel aux messages de type *Router Advertisement*. Il est aussi possible de fournir les adresses des serveurs DNS par cette méthode.

Le mode *stateful* nécessite un serveur DHCPv6 et se rapproche beaucoup du fonctionnement habituel en IPv4 avec un DHCP. La grosse différence consiste dans le fait que les adresses des routeurs ne sont pas fournies par le serveur DHCP, mais par les routeurs eux-mêmes.

Bien entendu, il est toujours possible de figer les adresses IP, les routes, les DNS, etc., à la main via les fichiers de configuration habituels. Cette méthode est même fortement conseillée pour les serveurs et c'est la seule qui fonctionnera si vous décidez de *subnetter* avec un masque supérieur à `/64`.

Il est possible de cumuler les trois méthodes en parallèle sur le même lien local, en décidant par exemple que tous les serveurs auront une IP fixe écrite en dur dans le fichier de configuration, que les stations connues seront fournies en DHCPv6 et que toutes les machines inconnues qui se connectent verront leur IP fournie par le routeur. C'est une méthode qui peut paraître un peu bizarre, mais on verra que le client DHCP a besoin d'être configuré a minima en IPv6 et que cette méthode permettra beaucoup de subtilités.

2.3.1 Mode Statless (SLAAC)

2.3.1.1 Création automatique d'adresses

Lors de l'élaboration d'IPv6, un leitmotiv a guidé une bonne partie des travaux. Il s'agissait d'automatiser au maximum la connexion d'une station au réseau. On devait pouvoir brancher l'équipement, le mettre en marche et il était connecté avec un accès direct aux services nécessaires. Certes tout ceci est possible assez facilement avec DHCP, mais le développement de ce dernier n'a commencé qu'en 1993, soit à peu près en même temps que les premiers balbutiements d'IPV6 (nommé **IPng**, pour « IP next generation », à l'époque). L'un des éléments de la solution a été d'écrire un protocole de création automatique d'adresses IP. Pour ce faire, on utilise l'adresse MAC des cartes, adresse censée être unique, on l'étend de 48 à 64 bits, on y rajoute un préfixe et l'adresse est créée.

La fabrication automatique d'une adresse se fait de la façon suivante. Supposons une station dont la carte réseau possède l'adresse MAC `00:1c:c4:ad:24:24`, cela fait 48 bits sur les 64 dont nous avons besoin. Pour passer de 48 à 64 bits, on va couper l'adresse MAC en deux, prendre la partie « vendeur » (les trois premiers octets), y concaténer la valeur `0xfffe`, puis rajouter la partie « numéro de série ». Ce qui va nous donner `00:1c:c4:ff:fe:ad:24:24`. Cette façon de faire répond à la norme IEEE EUI-64 utilisée aussi dans les réseaux IEEE 1394 (*firewire*).

Cependant ce serait trop simple d'en rester là. En effet, il peut être utile de savoir si l'adresse ainsi fabriquée est universelle (unique sur Internet) ou pas et s'il s'agit d'une adresse de groupe (*multicast* par exemple) ou pas. Pour ce faire, les deux bits de poids faible du premier octet de l'adresse MAC ont été nommés et utilisés respectivement ainsi :

- « u » pour universel (bit 7). Ce bit devrait être à zéro si l'adresse est universelle, mais pour des raisons de lisibilité, la norme IPv6 a décidé de l'inverser et de le mettre à un pour désigner une adresse universelle. Cela permet de le laisser à zéro pour les adresses purement locales.
- « g » pour groupe (bit 8), à zéro s'il s'agit d'une adresse individuelle - comprendre « normale ».

Donc, revenons à notre carte réseau d'adresse MAC `00:1c:c4:ad:24:24`.

1. On insère le mot `0xfffe` au milieu, ce qui va donner `00:1c:c4:ff:fe:ad:24:24` ;
2. On met à 1 le bit `u` (septième bit du premier octet), car une adresse basée sur une adresse MAC est censée être unique, cela donne `02:1c:c4:ff:fe:ad:24:24` ;
3. On rajoute un préfixe, par exemple – cas d'une adresse de lien local – `fe80::/10` pour obtenir in fine l'adresse IPv6 `fe80::21c:c4ff:fead:2424`.

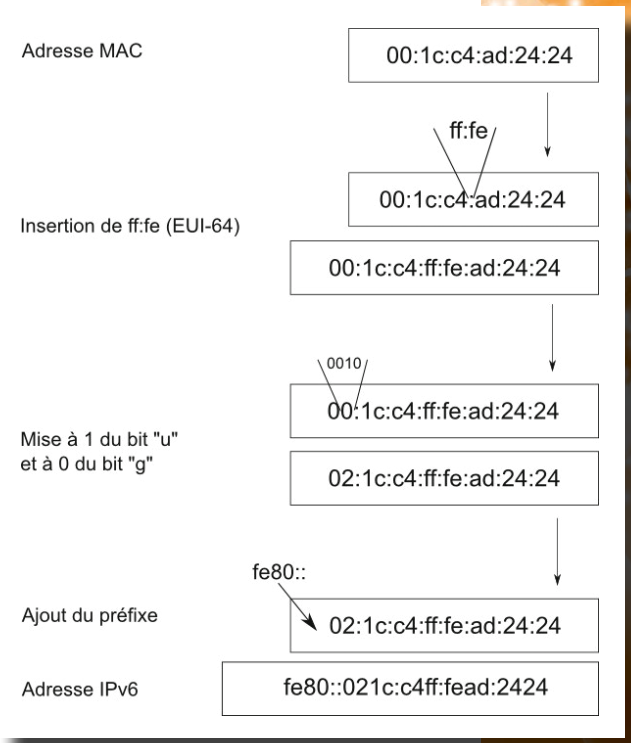


Fig. 5 : Création automatique d'une adresse IP à partir de l'adresse MAC.

NOTE

ATTENTION : Une adresse de lien local – de même qu'une adresse *multicast* – n'indique pas intrinsèquement l'interface de sortie, car elles partagent toutes le même préfixe en `fe80::/10`, il faut donc spécifier de manière explicite sur quelle interface doivent être émis les paquets. Par exemple, sous GNU/Linux :

Terminal

```
# ping6 -I eth0 fe80::216:3eff:fefe:7fc9
```

Une autre syntaxe existe utilisée sous *BSD, MacOS et Windows :

Terminal

```
# ping6 fe80::216:3eff:fefe:7fc9%en0
```

Il est bien entendu que, normalement, deux interfaces distinctes ne sont jamais sur le même sous-réseau. En conséquence, ce problème ne se présente pas avec les adresses routables.

Voilà comment sont créées automatiquement les adresses. Le même principe est utilisé pour toutes les créations automatiques d'adresses, routables ou non.

Un résumé en image est visible figure 5, page précédente.

Un mécanisme de détection d'adresse dupliquée (DAD pour *Duplicate Address Detection*) est mis en œuvre à chaque fois pour valider l'unicité de l'adresse créée sur le lien. En cas de duplication, l'adresse n'est pas créée et le mécanisme ressort en erreur.

2.3.1.2 Le Router Advertisement

Le *Router Advertisement* est un système de diffusion d'ICMPv6 fournissant les informations de routage – adresse du routeur par défaut par exemple – mais aussi le préfixe d'adresses routables pour les configurations de type SLAAC ainsi que l'adresse des serveurs DNS.

Lors de l'envoi du préfixe, les messages *Router Advertisement* (RA) fournissent aussi une durée de vie conseillée (*preferred lifetime*) et une durée de validité (*valid lifetime*). Ce mécanisme équivaut au système de bail du DHCP. En règle générale, la durée de vie conseillée n'expirera pas en raison des annonces régulières faites par les routeurs. Si pour une raison ou une autre les baux ne sont pas renouvelés, une adresse qui dépasse sa durée de vie conseillée devient dépréciée (*deprecated*), mais continue d'exister. Les nouvelles sessions devront éviter de l'utiliser tandis que les sessions en cours ne seront pas interrompues. Dans le cas où la durée de validité est dépassée, l'adresse est supprimée de l'interface et les sessions en cours sur cette adresse sont coupées. Ce mécanisme est très intéressant, car il permet de faire face à de nombreux problèmes qui sont épineux en IPv4 comme la renumérotation d'un réseau par exemple. En IPv6, il suffit de changer le préfixe sur le routeur et petit à petit toutes les machines configurées en SLAAC vont migrer. Une fois la migration des stations faite, on supprime l'ancienne route sur le routeur et le tour est joué.

Les deux durées de vie sont bien entendu réglables et peuvent être infinies.

2.3.2 Mode stateful (DHCPv6)

Ce mode utilise un DHCP à peu près comme en IPv4 et on retrouve assez facilement ses petits.

Le DHCPv6 est très semblable au DHCPv4 vu de l'extérieur, mais en interne pas mal de changements ont eu lieu. Tout d'abord, le DHCPv6 utilise l'*anycast* et non plus le *broadcast* qui n'existe pas en IPv6 et les messages ont été remaniés. Par exemple, un client enverra un message ICMPv6 de type SOLICIT (équivalent au message DHCPDISCOVER en v4) sur l'adresse *anycast* du routeur du sous-réseau qui renverra un message de type ADVERTISE (anciennement DHCPOFFER). De même qu'en v4 le serveur DHCP peut renvoyer le nom de domaine, l'adresse des serveurs DNS, mais jamais l'adresse du routeur par défaut, cela reste l'apanage de chaque routeur de s'annoncer sur le réseau via les messages *Router Advertisement*.

CONCLUSION

Ce rapide survol des protocoles IPv4 et IPv6 est utile pour bien comprendre ce qui se passe lorsque l'on monte un tunnel, mais aussi pour savoir ce qui se passe dans les « tuyaux ». Bientôt IPv6 aura remplacé totalement IPv4 et de nombreuses notions vues ici seront indispensables à maîtriser pour quiconque voudra monter son propre réseau. Nous avons ici volontairement passé sous silence les aspects pratiques, mais tout ceci est disponible dans votre revue préférée (numéros 136 à 140, 182 et 184 entre autres). Vous disposez là de toutes les informations nécessaires pour vous lancer, alors n'attendez pas :) ▉

RÉFÉRENCE

[1] **http://en.wikipedia.org/wiki/List_of_IP_protocol_numbers**

À LA DÉCOUVERTE DES VPNS

Cédric PELLERIN

Lorsque l'on veut transporter des données de façon sécurisée, il faut s'assurer de deux choses : d'une part, que les données reçues n'ont pas été altérées – volontairement ou non ; d'autre part, qu'elles ne puissent être lues par des tiers, on parle alors de chiffrement. En fonction des choix effectués, nous verrons que sur la totalité d'une trame des morceaux plus ou moins grands sont chiffrés et/ou authentifiés.

1. OPENVPN

1.1 Considérations préliminaires

OpenVPN est un VPN SSL qui n'est en rien compatible avec **L2TP**, **IPSec**, **PPTP** ou autres. La plupart des protocoles classiques sont prévus pour être implémentés au niveau noyau alors qu'OpenVPN tourne à 100 % en espace utilisateur. Les paquets rentrent et sortent via les interfaces spécifiques **`tun`** ou **`tap`** et sont traités en dehors du *kernel*. Cette option permet d'avoir un logiciel facilement portable du moment que le système d'exploitation supporte au moins l'un des deux types d'interfaces spécifiques. On pourrait légitimement craindre une vitesse moindre par rapport aux implémentations de VPN dans le noyau, mais tous les tests disponibles prouvent que la différence oscille entre le nul et le totalement négligeable.

1.2 Le header

Étant donné qu'OpenVPN ne fait l'objet d'aucune RFC, une description claire de son protocole est compliquée à trouver. On sait cependant qu'il utilise TLS pour le chiffrement et qu'il peut fonctionner sur UDP ou TCP. Par ailleurs, on sait aussi qu'il empaquette les données chiffrées dans des trames IP, ce qui le rend très difficile à détecter.

Sachant cela, et en examinant les trames avec **Wireshark**, on peut en déduire le format d'une trame OpenVPN en prenant l'exemple des options TLS par défaut :

<table>
<tr><td colspan="4">En-tête IP</td></tr>
<tr><td colspan="4">En-tête UDP</td></tr>
<tr><td>Packet tag
(1 octet)</td><td>HMAC-SHA1 signature
(20 octets)</td><td>Vecteur d'initialisation
(16 octets)</td><td>N° de séquence
(4 octets)</td></tr>
<tr><td colspan="4">En-têtes d'origine (IP + UDP/TCP)
(28 octets)</td></tr>
<tr><td colspan="4">Données</td></tr>
</table>

En examinant ce schéma, on s'aperçoit qu'OpenVPN rajoute un « overhead » d'environ 69 octets. Je dis « environ », car bien entendu cela dépend fortement du chiffrement et du *digest* utilisés. Il est à noter que les nombreuses options d'OpenVPN peuvent entraîner des changements notables dans ce schéma qui est donc à prendre à titre indicatif.

1.3 Zoom sur les devices tun et tap

Les *devices* **`tun`** (pour tunnel) simulent un périphérique réseau opérant au niveau 3 du modèle OSI, c'est-à-dire le niveau « réseau », par exemple IP dans le cadre d'un réseau TCP/IP.

Les *devices* **`tap`** (pour *tapping*, dans le sens d'écoutes téléphoniques) simulent eux un périphérique de niveau 2 – Ethernet par exemple – qui sont prévus pour capturer et renvoyer tout le trafic réseau qui passe par eux.

Les paquets envoyés par le système d'exploitation via un périphérique **`tun`** ou **`tap`** sont remis au programme tournant en espace utilisateur qui est connecté à ce *device*. Ce même programme peut bien entendu envoyer lui aussi des données vers les périphériques **`tun`**/**`tap`** et dans ce cas les données envoyées sont injectées directement dans la pile réseau.

OpenVPN utilise les *devices* **`tun`** lorsqu'il travaille en mode tunnel et les *devices* **`tap`** lorsqu'il travaille en mode *bridge*. La description de ces deux modes de fonctionnement est faite dans les chapitres consacrés à OpenVPN.

Comme on le voit, nul besoin ici de grands développements de protocoles plus ou moins alambiqués pour créer une solution de VPN fiable, performante et surtout très souple. L'usage de SSL/TLS pour le chiffrement, d'algorithmes HMAC (pour *Hash-based Messages Authentication Code*) bien connus, de l'encapsulation pour le *tunneling* et des interfaces étudiées ci-dessus permettent de faire en sorte que tout le code d'OpenVPN tourne en espace utilisateur et soit facilement portable.

1.4 Les flux de données et de contrôle

OpenVPN fonctionne en multiplexant les tunnels sur un seul port TCP ou UDP, mais aussi en entrelaçant des trames de données et des trames de contrôle. L'information sur le type de paquet se trouve dans le premier octet nommé plus haut « packet tag ». Cet octet contient deux informations, le « packet opcode » qui tient sur les 5 bits de poids fort et le « key ID » sur les 3 bits de poids faible. Dans le cas où OpenVPN est en mode serveur TLS - le plus fréquent - si l'opcode indique que le paquet contient des données le *key ID* est utilisé pour trouver l'état TLS local auquel il est associé. Si l'état en question indique que la communication est active, authentifiée et que le client est bien celui attendu, alors seulement les informations nécessaires au déchiffrement sont chargées.

Si l'opcode signale au contraire une trame de contrôle, alors le paquet est authentifié puis en fonction de l'opcode et du session ID l'une des actions suivantes est effectuée :

- un *hard reset* du serveur ;
- un *hard reset* du client ;
- un *soft reset* du client.

Le session ID est constitué par les 8 octets suivants le *packet* tag. Il est lu directement dans le cas d'une trame de contrôle.

Ces actions peuvent entraîner une renégociation de la clé de session pour le canal de données en cas de besoin.

1.5 Conclusion

Comme on le voit, le mode de fonctionnement d'OpenVPN est fort simple et basé principalement sur des protocoles éprouvés et surtout bien tenus à jour. Le code source est assez facilement lisible et commenté de partout, ce qui permet un debug relativement facile. Son approche n'est certes pas très académique, mais il est un peu aux VPNs ce que Linux est aux systèmes d'exploitation, un produit de geek, pensé pour des geeks, mais dont la popularité et le déploiement dépassent plus que largement du cadre prévu initialement.

2. IPSEC

2.1 Protocoles d'authentification et de chiffrement

IPSec utilise deux protocoles pour authentifier et chiffrer. Le premier s'appelle **AH** pour *Authentication Header* (authentification), le second **ESP** pour *Encapsulating Security Payload* (chiffrement).

Le protocole AH fournit un mécanisme dédié uniquement à l'authentification. Il permet de vérifier l'intégrité et l'origine des données et possède un système optionnel contre le rejeu de paquets. L'intégrité des données est assurée par l'utilisation d'un « digest » qui est généré par un algorithme tel que MD5 ou SHA. Un *digest* est une suite de caractères plus ou moins longue qui possède la propriété de changer énormément d'un message à l'autre même si la différence entre les deux messages est minime. Par exemple, tapez ces deux commandes dans un terminal :

Terminal

```
$ echo "salut" | md5sum
6aba532a54c9eb9aa30496fa7f22734d  -
$ echo "salut." | md5sum
5db5d85d93efeeddd6584496e5032fc7  -
$ echo "salut " | md5sum
e01042aec91a877459c0176b713dba28  -
```

Vous voyez la différence entre les trois sommes md5 alors que les modifications du message initial sont presque invisibles. C'est le principe du *digest*.

L'authentification de la provenance des données est, elle, assurée par l'utilisation d'une clé secrète partagée pour créer le *digest*. Enfin, la protection contre le rejeu consiste en la présence d'un champ pour un identifiant de séquence dans l'entête de AH.

Le rejeu est un système assez simple dans lequel l'attaquant, en écoutant la communication, vole le *digest* du mot de passe de l'un des interlocuteurs. Il lui est simple ensuite d'usurper son identité, car il lui suffit de présenter le *digest* volé à la troisième personne pour que le mot de passe soit reconnu. Si on ajoute un identifiant unique de séquence dans l'en-tête (ou ailleurs), le rejeu est impossible, car le *digest* présenté par l'attaquant sera bon, mais pas l'identifiant. C'est ainsi que procède AH.

AH authentifie le *payload* et les entêtes à l'exception des champs pouvant légitimement être modifiés au cours de la transmission comme le champ du TTL par exemple.

Le protocole ESP quant à lui fournit la couche de confidentialité des données, c'est-à-dire le chiffrement, et l'authentification. ESP peut être utilisé soit pour le chiffrement seul, soit pour l'authentification seule, soit les deux. Cependant, contrairement à AH, l'authentification ne porte que sur la partie datagramme IP du paquet et non l'ensemble du paquet, en-têtes compris. Les algorithmes d'authentification sont les mêmes que ceux utilisés par AH.

Ces deux protocoles peuvent être utilisés seuls ou ensemble pour protéger un paquet IP.

2.2 Architecture et méthodes d'implémentation

Ayant été conçu pour cela, IPSec peut facilement être implémenté dans une pile IPv6 de façon à rendre son fonctionnement parfaitement transparent. Cependant, le portage en IPv4 ne permet pas ce mode de fonctionnement, car il faudrait revoir entièrement la pile IP, ce qui ne serait pas franchement pratique. Pour intégrer IPSec dans une pile IPv4, il existe deux méthodes. La première, nommée **BITS** pour « Bump In The Stack », fonctionne en rajoutant une couche IPSec entre la couche IP et la couche liaison (Ethernet, ppp, token ring...). Elle a l'avantage d'être facile à intégrer, mais l'inconvénient de dupliquer l'effort à plusieurs endroits (voir figure 1).

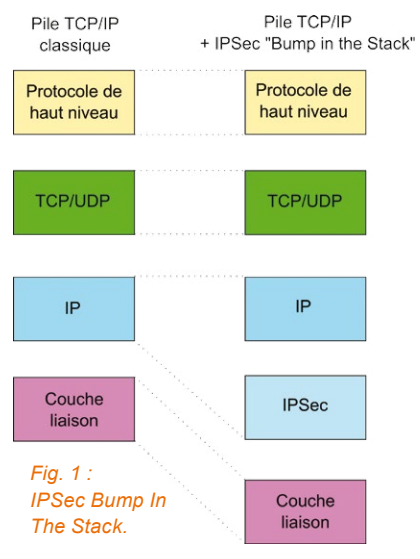


Fig. 1 : IPSec Bump In The Stack.

L'autre méthode, nommée « Bump in the Wire » ou **BITW**, consiste simplement à rajouter des machines sur le trajet des paquets qui vont les chiffrer au passage. On peut voir ça tout simplement comme des routeurs chiffreurs.

Hormis ces méthodes d'implémentation dans le réseau, IPSec possède aussi deux modes de fonctionnement : le mode **Transport** et le mode **Tunnel**. En mode Transport, comme son nom le suggère, IPSec protège le message venant de la couche Transport (TCP, UDP, etc.). Dans ce cas, le message est authentifié et/ou chiffré par AH et ESP et les *headers* appropriés sont rajoutés avant les en-têtes TCP ou UDP. L'en-tête IP est, lui, rajouté ensuite (voir figure 2, page suivante).

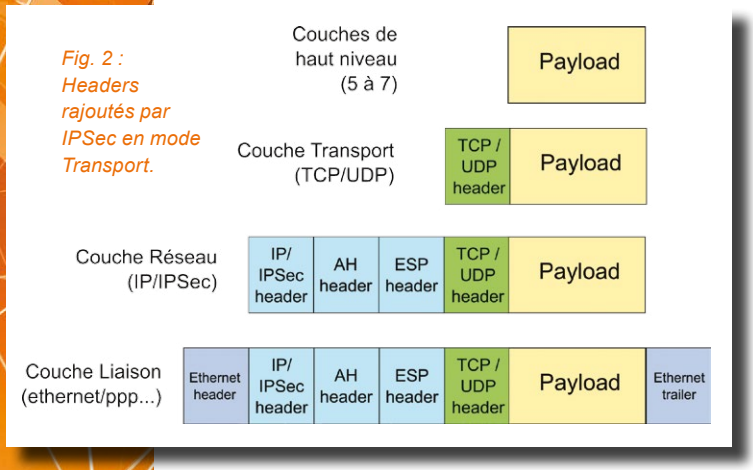


Fig. 2 : Headers rajoutés par IPSec en mode Transport.

De par leur fonctionnement, le mode Transport est la plupart du temps utilisé dans les implémentations IP intégrant la couche IPSec, IPv6 en général, tandis que le mode Tunnel se retrouve dans les implémentations de type « Bump in the Stack » et « Bump in the Wire ».

En mode Tunnel, IPSec est utilisé pour protéger une trame IP complète, *header* IP compris. Dans ce mode, le *header* IPSec est rajouté à la trame IP normale et un nouveau *header* IP est rajouté. On fait de l'encapsulation IP dans IP. L'avantage de ce mode est que la totalité de la trame IP d'origine est sécurisée (voir figure 3).

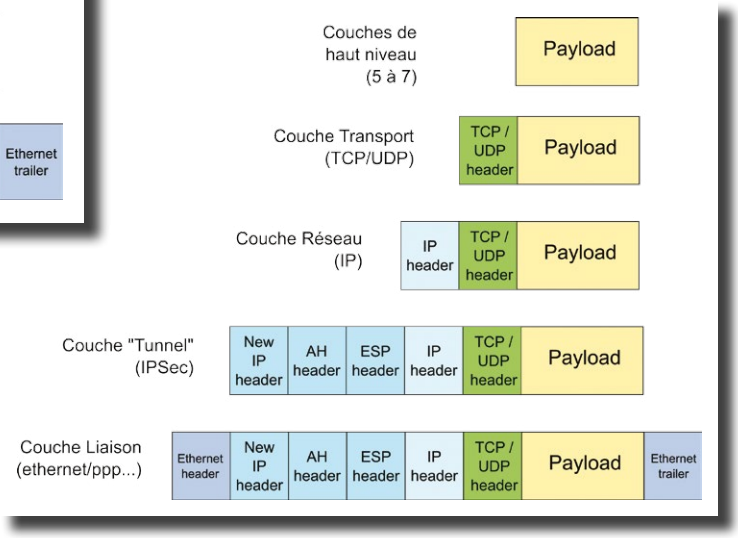


Fig. 3 : Headers rajoutés par IPSec en mode Tunnel.

2.3 Security Associations, Security Policies, Selectors et Security Parameter Index

Au moins là, on est presque sûr qu'on est en train de parler de sécurité... Ces notions sont importantes à comprendre avant d'aborder IPSec plus en profondeur, elles sont là pour contrôler les échanges entre les machines, la façon dont IPSec travaille et s'assure que les trames entrantes et sortantes sont convenablement traitées.

Les échanges sur un réseau classique n'ont pas tous besoin du même niveau de confidentialité. Si vous « pinguez » une machine, vous n'avez cure que la NSA le sache, en revanche si vous envoyez une copie de votre relevé de compte, vous préférez que cela reste aussi confidentiel que possible. Au niveau d'IPSec, c'est le rôle de ces *Security* que de savoir si la trame nécessite qu'on rajoute un bon nombre d'octets pour la sécuriser ou non.

Pour gérer ces complexes subtilités, IPSec est équipé d'un système puissant et flexible lui permettant de savoir comment il doit traiter les différents types de trames. Afin de comprendre comment tout ceci fonctionne, il nous faut d'abord définir deux concepts majeurs :

- Les **Security Policies** (**SP**) ou Politiques de Sécurité : il s'agit de règles intégrées dans l'implémentation d'IPSec qui indiquent comment traiter les différentes trames reçues par le système. Par exemple, ces règles sont utilisées pour décider si tel paquet a besoin d'être traité par IPSec, si on utilise AH et ESP, etc. Si une sécurisation est demandée, ces règles donnent aussi des indications générales sur comment sécuriser. Les Politiques de Sécurité d'une machine sont stockées dans une base de données appelée **Security Policy Database** ou **SPD**. Il existe une SPD par interface réseau sur lesquelles IPSec est activé et il peut même y en avoir deux : une pour le trafic entrant et une pour le trafic sortant. Les SPD doivent être consultées pour tout le trafic, y compris celui qui n'est pas IPSec.
- Les **Security Associations** (**SA**) ou Associations de Sécurité : il s'agit là d'informations de sécurité décrivant chacune un type particulier de connexion sécurisée entre plusieurs machines. Elles précisent les

divers mécanismes de sécurité à mettre en œuvre pour tel ou tel type de connexion. Les Associations de Sécurité d'une machine sont stockées dans une base de données appelée **Security Associations Database** ou **SAD**. Les SA sont partagées entre toutes les machines ayant besoin de communiquer entre elles.

La principale différence entre une SP et une SA est que la SP porte sur les aspects généraux tandis qu'une SA s'occupe du détail de chaque type de connexion. Afin de déterminer comment traiter une trame, IPSec va premièrement interroger sa SPD. Il se peut, en fonction du type de trame, que cette dernière fasse référence à la SAD qui sera à ce moment-là interrogée, la SA extraite et utilisée par IPSec.

Tout ceci est beau et bon, mais la question se pose : comment IPSec peut déterminer la SA à utiliser pour la trame qui vient d'arriver ? C'est là qu'interviennent les **Selectors**. Il s'agit de structures comprenant un certain nombre de critères (adresse source, adresse de destination, port, etc.) et un pointeur sur une SA. Les critères peuvent être multiples et plus ou moins précis, par exemple on peut parfaitement donner une plage d'adresses comme critère 1 et un port précis comme critère 2. À partir de ces sélecteurs, IPSec est donc en mesure de déterminer quelle SA appliquer. Cependant, il faut savoir que les SA sont unidirectionnelles et totalement dépendantes de la machine sur laquelle elles sont installées. En clair, si la machine A communique avec la machine B, on aura une SA sur A pour le trafic de A vers B sortant de A, une autre sur A pour le trafic de B vers A entrant sur A, encore une autre sur B pour le trafic de A vers B entrant sur B et enfin une sur B pour le trafic de B vers A sortant de B. Donc on aura quatre SA pour un seul trafic bidirectionnel.

Pour les désigner, les SA n'ont pas de nom, mais sont définies par trois paramètres appelés triplet :

- Le **Security Parameter Index** ou **SPI** : il s'agit d'un nombre sur 32 bits qui va identifier de manière unique la SA utilisée. Ce SPI est transporté dans les en-têtes AH et ESP afin d'être disponible à l'arrivée pour y être utilisé dans le but de trouver la SA idoine.
- **Adresse IP de destination** : c'est l'adresse de destination de la machine pour laquelle la SA est établie.
- Le **Security Protocol Identifier** : indique si l'association est pour AH ou pour ESP. Si les deux sont utilisés conjointement, ils auront chacun leur propre SA.

Les associations de sécurité sont donc utilisées pour chiffrer et déchiffrer les messages. Elles doivent donc être connues de tous les périphériques concernés. C'est là le travail du protocole nommé **Internet Key Exchange** ou **IKE** que nous verrons plus bas.

2.4 Les en-têtes

2.4.1 L'en-tête AH

Comme nous l'avons vu, AH s'occupe de fournir un mécanisme d'authentification pour tout ou partie des trames IP en rajoutant un *header* qui est calculé en fonction de la valeur des différents éléments de la trame. Ce protocole fonctionne presque comme les CRC que l'on utilise pour les détections d'erreur à ceci près que l'algorithme est spécifique au hachage des données et qu'on y associe une clé de chiffrement connue uniquement de l'émetteur et du récepteur. Une SA est mise en place entre les deux interlocuteurs afin de préciser comment effectuer les opérations d'authentification, ainsi seuls les deux protagonistes seront capables de s'y retrouver. Sur la source, AH effectue les calculs et met le résultat nommé **Integrity Check Value** ou **ICV** dans un en-tête spécial. À l'autre bout, le destinataire effectue les mêmes calculs en utilisant la clé qu'ils partagent, ce qui lui permet de voir immédiatement si la trame a été altérée ou non.

Il est très important de noter que AH n'effectue aucun chiffrement et que les données ne sont nullement modifiées par son action. Il se contente juste de permettre la vérification de leur intégrité.

La méthode de calcul du *header* AH est similaire pour IPv4 et pour IPv6 ; la principale différence réside dans le mécanisme utilisé pour insérer l'en-tête dans la trame et pour lier les en-têtes entre eux.

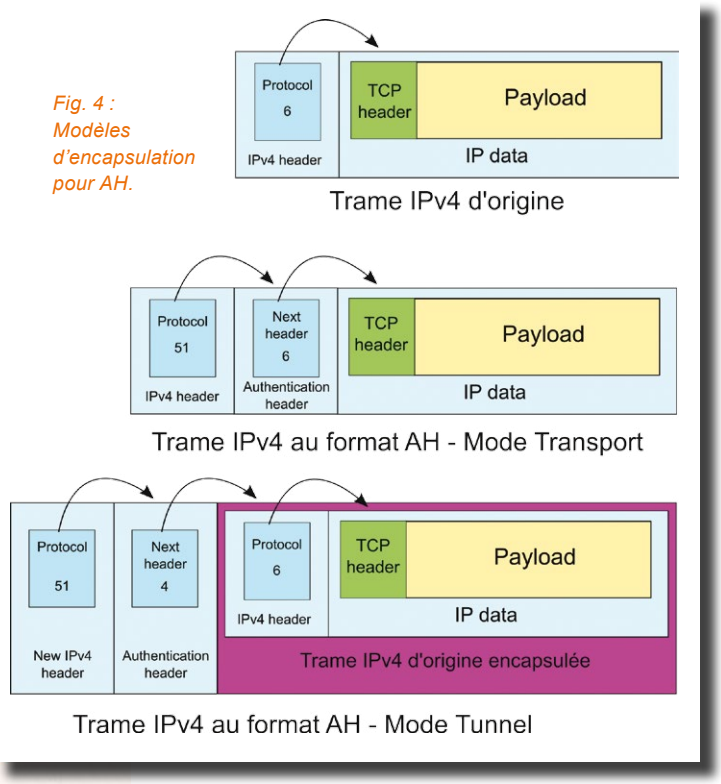


Fig. 4 : Modèles d'encapsulation pour AH.

En IPv6, il existe un mécanisme d'extensions prévu de base dans le protocole et AH en fait bon usage (voir l'en-tête IPv6 dans le chapitre sur la théorie TCP/IP). Dans les deux modes de fonctionnement d'IPSec (tunnel et transport), tous les champs sont authentifiés.

Pour IPv4, l'astuce consiste à utiliser le champ « Protocole » dans l'en-tête. Ce champ désigne le protocole encapsulé dans le *payload* IP. Il va être recopié dans l'en-tête AH et remplacé par le numéro désignant le protocole AH lui-même, soit **51**. C'est nettement plus parlant avec le petit dessin de la figure 4.

On remarque bien ici la notion de « tunnel » où la trame d'origine est presque camouflée au sein de la nouvelle trame.

Nous pouvons maintenant voir comment est formé le fameux *header* AH (pléonasme) :

Prochain en-tête (8 bits)	Taille des données (8 bits)	Réservé
Security Parameter Index		
Numéro de séquence		
Integrity Check Value		

La taille de l'ICV est variable en fonction de l'algorithme choisi pour ce faire et de la taille des différentes trames possibles, mais elle doit être un multiple de 32 bits. De même, la taille totale du *header* doit être un multiple de 32 bits en IPv4 et de 64 bits en IPv6. Pour atteindre cet objectif, il est possible de rajouter des octets de bourrage (*padding*) au champ ICV.

Champ	Taille (en octets)	Description
Prochain en-tête	1	Contient le n° de protocole du prochain header après AH. Est utilisé pour lier les headers entre eux.
Taille des données	1	Taille du header d'authentification, pas du payload.
Réservé	2	Non utilisé, mis à zéro.
Security Parameter Index	4	Valeur sur 32 bits qui, combinée avec l'adresse de destination et le type de protocole de sécurité, permet d'identifier l'association de sécurité (SA) à utiliser pour cette trame.
Numéro de séquence	4	Compteur initialisé à zéro quand une association de sécurité est formée entre deux appareils puis incrémenté pour chaque trame envoyée en utilisant ce SA. Son unicité identifie chaque trame pour un SA donné et fournit une protection contre les attaques par rejeu.
Integrity Check Value	Variable	Contient le résultat de l'algorithme de hachage utilisé par AH.

2.4.2 L'en-tête ESP

Autant le protocole AH sait assurer l'intégrité des données, autant il ne peut rien faire pour les rendre confidentielles. Ce travail est à la charge d'un autre protocole nommé **Encapsulating Security Payload** ou **ESP**. Avec ESP, un algorithme de chiffrement mélange les données de la trame avec une clé pour les encoder. Le résultat est réarrangé pour être transmis au destinataire qui pourra les décoder. ESP dispose aussi de son propre système d'authentification ou peut être utilisé en collaboration avec AH.

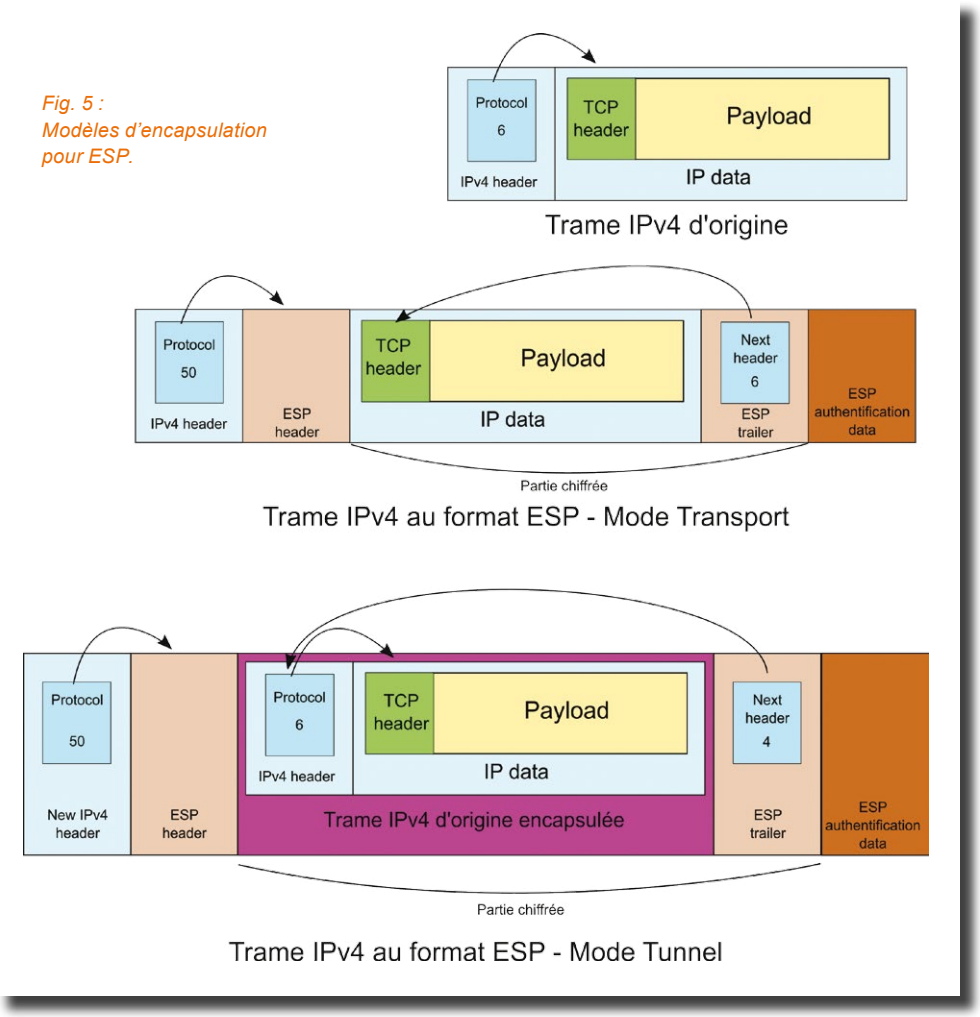


Fig. 5 : Modèles d'encapsulation pour ESP.

Contrairement à AH qui se contente de rajouter un en-tête, ESP va rajouter trois composants :

- Un *header* qui contient deux champs, le *Security Parameter Index* et le numéro de séquence (**Sequence Number**). Ces informations sont placées avant les données chiffrées.
- Un *trailer* qui regroupe les octets de *padding* pour l'alignement et le champ « Prochain en-tête » pour ESP.
- Une zone nommée Données d'Authentification ESP (**ESP Authentification Data**) qui contient l'ICV calculé de manière similaire à AH. Ce champ n'est présent que lorsque la fonctionnalité optionnelle d'authentification d'ESP est activée.

Il y a deux bonnes raisons à ce que ces informations soient réparties en trois blocs. La première est que certains algorithmes de chiffrement exigent de travailler sur des blocs de données de taille précise et donc les octets de bourrage doivent être positionnés après les données. La seconde est que le contenu du champ ESP Authentification Data contrôle l'intégrité des données chiffrées, y compris le *header* et le *trailer*. Il doit donc se situer en dehors complètement.

Pour s'insérer dans des trames IP, ESP travaille en gros comme AH. En IPv6, il profite du mécanisme d'extensions intégré au protocole et en IPv4, il utilise le même « subterfuge » que AH à ceci près qu'il rajoute un *header* et un *trailer* et que le pointeur vers le *header* TCP (en mode transport) ou IP (en mode tunnel) est situé dans le *trailer*.

Comme on le voit sur la figure 5, le *trailer* ESP est ajouté à la trame avant de réaliser le chiffrement. Par conséquent le *trailer* est lui aussi chiffré, seul le *header* ESP – et bien sûr le nouveau *header* IP – restent en clair.

Le *header* et le *trailer* ESP sont relativement simples à comprendre et sont constitués ainsi :

Header	Security Parameter Index (SPI)		
	Numéro de séquence		
Payload	Données du payload ESP		
Trailer	Bourrage	Taille du bourrage	En-tête suivant
Données d'authentification ESP			

Les noms des divers champs parlent plus ou moins d'eux-mêmes, mais voici quelques précisions utiles :

Section	Champ	Taille (octets)	Description
Header	SPI	4	Valeur sur 32 bits qui, combinée avec l'adresse de destination et le type de protocole de sécurité, permet d'identifier l'association de sécurité (SA) à utiliser pour cette trame.
	Numéro de séquence	4	Compteur initialisé à zéro quand une association de sécurité est formée entre deux appareils puis incrémentée pour chaque trame envoyée en utilisant ce SA. Son unicité identifie chaque trame pour un SA donné et fournit une protection contre les attaques par rejeu.
Payload	Données du payload	Variable	Toutes les données chiffrées constituées par les messages des couches plus hautes ou par une trame IP encapsulée. Peut aussi inclure des données spécifiques comme un vecteur d'initialisation requis par certaines méthodes de chiffrement.
Trailer	Bourrage	Variable (0 à 255)	Octets rajoutés en fonction des besoins de l'algorithme de chiffrement ou d'alignement.
	Taille du bourrage	1	Nombre d'octets servant de bourrage.
	En-tête suivant	1	Contient le numéro de protocole du prochain en-tête dans la trame. Il s'agit en général d'un en-tête TCP ou UDP en mode transport ou IP en mode tunnel.
Données d'authentification		Variable	Contient l'ICV issu de l'algorithme d'authentification optionnel.

2.5 Internet Key Exchange

Plus connu sous le nom de IKE, ce protocole permet d'échanger les SA entre deux machines de manière aussi sécurisée que possible. Défini dans la RFC 2409 pour la version 1 et dans la RFC 4306 pour la version 2, IKE est l'un des protocoles les plus compliqués à comprendre dans tous ceux utilisés par IPSec, en effet il faut un bon bagage en mathématiques et en cryptographie pour espérer s'en sortir. Étant donné les très nombreuses différences entre IKEv1 et IKEv2, nous allons nous concentrer sur ce dernier.

Le protocole IKEv2 établit et maintient dynamiquement un état partagé entre l'émetteur et le récepteur d'une trame IP. Il effectue l'authentification mutuelle entre les deux parties et établit l'association de sécurisation (SA) pour IKE. Celui-ci, que l'on va appeler IKE-SA pour le distinguer des autres SA, utilise des informations secrètes partagées qu'il stocke pour remplir deux fonctions différentes :

- ⇨ Établir les « CHILD-SA » pour ESP et AH (les SA dont nous avons parlé plus haut) ;
- ⇨ Définir les algorithmes de chiffrement qui seront utilisés par les SA.

IKEv2 est un protocole qui fonctionne par paires de requête/réponse qui sont appelées échanges. Le demandeur porte la responsabilité de garantir la fiabilité. Si une réponse n'est pas reçue, le demandeur peut soit renvoyer la requête, soit fermer la connexion. IKEv2 dispose de quatre types d'échanges :

- **IKE_SA_INIT** : Il s'agit du premier échange qui établit le IKE-SA. Il doit absolument être accompli avant de commencer n'importe quel autre échange. Il assure trois fonctions dans la mise en place de l'IKE-SA :
  - la négociation des paramètres de sécurité pour l'IKE-SA ;
  - l'envoi des nonces ;
  - l'envoi des données Diffie-Hellman.

Les nonces (*number used once*) sont des nombres arbitraires destinés à être utilisés une seule fois. Il s'agit souvent d'un nombre aléatoire ou pseudo-aléatoire servant à garantir que les anciennes communications ne peuvent pas être réutilisées dans des attaques par rejeu.

- **IKE_AUTH** : Ce deuxième échange doit lui aussi absolument être terminé avant de poursuivre plus avant. Il assure trois fonctions essentielles :
  - l'envoi des identités ;
  - la preuve de la connaissance des secrets liés aux protagonistes ;
  - l'établissement de la première – et en général unique – CHILD-SA pour AH et/ou ESP.
- **CREATE_CHILD_SA** : Cet échange est simplement utilisé pour créer d'autres CHILD-SA en fonction des besoins.
- **INFORMATIONAL** : Échange de maintenance concernant les SA. Parmi ses fonctions, nous pouvons noter :
  - Suppression des SA quand besoin il y a ;
  - Rapport d'erreurs ;
  - Vérification des durées de vie des SA ;
  - etc.

Une fois les deux premiers échanges obligatoires effectués dans l'ordre, tous les autres peuvent intervenir dans n'importe quel ordre.

IKEv2 utilise un grand nombre de protocoles de chiffrement pour accomplir ses missions. Il est basé sur le protocole de gestion de clés Diffie-Hellman.

Il resterait encore beaucoup à dire à ce sujet, mais cela nécessiterait de trop rentrer dans les détails. Je ne peux qu'encourager le lecteur intéressé et courageux à aller lire les RFC indiquées.

CONCLUSION

Les quelques pages que vous venez de lire ne reflètent même pas la partie émergée de l'iceberg. J'ai essayé d'être le plus clair possible en passant sous silence bon nombre de détails dont l'explication nous aurait emmenés beaucoup trop loin. La cryptographie est une science à part entière avec son vocabulaire, ses codes et ses gourous (ceux qui méditent et les autres...;-)) Si vous vous sentez attirés par ce monde-là, potassez vos mathématiques et lisez beaucoup ; étudiez la documentation, mais aussi le code existant afin de comprendre comment les algorithmes ont été implémentés. Bonne découverte... ▉

COMMENT CHOISIR SON VPN ?

Cédric PELLERIN

Lorsque l'on cherche une solution de VPN en dehors des logiciels propriétaires, on s'aperçoit rapidement que le choix se résume à OpenVPN ou IPSec. Pour éviter de sortir le dé à douze faces ou de jouer ça à la roulette russe, voici quelques rapides informations et commentaires qui pourront vous aider.

1. LES DIVERS TYPES DE VPN

Plusieurs classifications existent, mais en général on ne cite que les VPN SSL et **IPSec**. C'est un peu court, car si l'on en croit la définition d'un VPN SSL il s'agit d'un système sans client basé sur un navigateur web qui est le seul autorisé à accéder aux applications. Cependant, il existe au moins deux autres types de tunnels VPN qui se rapprochent beaucoup des VPN SSL, car ils utilisent un chiffrement basé sur SSL tout en permettant de faire passer toute sorte de trafic :

- le tunnel **OpenSSH** classique ;
- **OpenVPN**.

Cependant, d'autres protocoles de tunnels chiffrés existent aussi bien entendu comme **DTLS** chez **Cisco**, **MPPE** et **SSTP** chez **Microsoft**, etc.

Dans la série des VPN non chiffrés, on trouve **L2TP** que l'on utilise souvent dans IPSec afin de rajouter une couche de sécurité. Ce protocole possède le gros avantage de travailler au niveau 2 de la couche OSI (couche Liaison) et ainsi d'être capable de transporter n'importe quel protocole réseau de TCP/IP à IPX/SPX en passant par NetBIOS et autres. C'est pourquoi on retrouve souvent des montages IPSec incluant L2TP pour ne pas se retrouver limité. D'autres protocoles de *tunneling* niveau 2 existent comme EtherIP ou IPLS, mais leur diffusion est beaucoup plus confidentielle.

2. QUEL VPN CHOISIR ?

Si l'on passe en revue les protocoles disponibles aisément sur toutes les plateformes grand public ou presque, on en trouve trois principaux : OpenVPN, IPSec et **PPTP**. Ils ont chacun leurs avantages et leurs inconvénients que nous allons voir plus en détail.

2.1 OpenVPN

OpenVPN est un protocole non standardisé – comprendre qu'il n'a pas fait l'objet d'une RFC – et donc sujet à changements sans avertissement. Cependant, il existe depuis 2001 et le code de la version communautaire est *open source* (GPLv2 et AGPL). Il existe aussi une version propriétaire qui inclut une interface graphique et quelques fonctionnalités supplémentaires. Sur le plan technique, les caractéristiques majeures sont les suivantes :

- fonctionne sur toutes les plateformes grand public du marché : Windows, **GNU/Linux**, ***BSD**, **OSX**, **Android**, **iOS**, mais n'est inclus en standard que dans Linux et *BSD ;
- peut travailler en TCP ou en UDP ;
- supporte IPv6 ;
- peut se connecter sur un port au choix (facilite le passage des *firewalls*) ;
- sait faire du tunnel en couche 2 via les interfaces TAP et le *bridging* ;
- supporte un très grand nombre de suites cryptographiques, dont les plus récentes ;
- aucune vulnérabilité majeure connue à ce jour ;
- configuration simple en quelques minutes quand on est habitué ;

- connexion très stable même en wifi ou en réseau cellulaire. Pour renforcer encore la fiabilité, il est possible de passer en TCP au prix d'une légère perte de vitesse.

2.2 IPSec

IPSec est un protocole de l'IETF ayant fait l'objet de nombreuses RFC. Il devait être obligatoirement inclus dans les piles IPv6 pour être conforme à la norme jusqu'à ce que la RFC 6434 le rende optionnel, car de nombreuses techniques de sécurisation existent aujourd'hui sans qu'aucune ne sorte du lot. Ses principales caractéristiques techniques sont :

- disponible sur toutes les plateformes grand public, mais aussi professionnelles comme Cisco ;
- utilise obligatoirement les ports UDP **500** et **1701**. Peut aussi nécessiter l'ouverture du port UDP **4500** pour passer les NAT ;
- supporte IPv6 ;
- aucune vulnérabilité connue à ce jour. Cependant des rumeurs persistantes font état du fait que la NSA saurait utiliser le protocole IKE (*Internet Keys Exchange*) pour décrypter des trafics IPSec. Il faut aussi noter que l'utilisation d'IPSec avec des *pre-shared keys* publiques est vulnérable à une attaque de type *Man In The Middle* ;
- configuration assez complexe, beaucoup de notions à appréhender avant de pouvoir monter la première connexion.

2.3 PPTP

PPTP est un protocole très basique de VPN. Par défaut, il n'est pas chiffré et se repose sur PPP pour cela. Or PPP utilise le protocole MPPE de Microsoft pour le chiffrement et celui-ci est obsolète depuis des années (RC4 128 bits...). Les caractéristiques à retenir sont :

- disponible absolument partout depuis Windows 95 ;
- protocole de chiffrement obsolète et vulnérable ;
- configuration extrêmement simple.

CONCLUSION

Comme on vient de le voir, des trois solutions disponibles simplement en environnement grand public, seuls IPSec et OpenVPN sont utilisables actuellement. Notre étude va donc se concentrer sur ces deux environnements – car parler de protocole serait trop restrictif – avec une certaine emphase sur OpenVPN qui est le plus flexible des deux. IPSec est à connaître, car il s'agit de la solution « officielle » et que face à des routeurs de type Cisco il n'existe pas d'autre solution.

En ce qui concerne les tunnels OpenSSH que nous avons évoqués, ils ne seront pas abordés car, même s'ils fonctionnent bien, ils ne sont pas prévus pour être utilisés autrement que ponctuellement. ■

OPENVPN

2 OPENVPN

À découvrir dans cette partie...

Créez une configuration OpenVPN simple

Pour pouvoir aborder OpenVPN, il faut commencer par être capable de mettre en place une configuration simple tout en s'assurant qu'elle soit tout de même robuste. p. 36

Utilisez les options avancées d'OpenVPN

Il est temps de passer à une configuration fine d'OpenVPN ! Cet article vous permettra d'augmenter significativement le niveau de sécurité de votre installation et de maîtriser totalement tous les paramètres de configuration. p. 56

Configurez les postes clients y compris sur des systèmes non libres

Les postes clients ne sont pas tous forcément sous GNU/Linux et il faut donc être en mesure de configurer les postes clients sur des systèmes non libres tels que Windows, MacOS ou encore Android. p. 84

CRÉEZ UNE CONFIGURATION OPENVPN SIMPLE

Cédric PELLERIN

OpenVPN est une solution de tunneling qui gagne de plus en plus de terrain chaque année et qui permet des configurations impossibles avec d'autres solutions concurrentes. Nous allons voir dans cette première partie comment établir une connexion déjà robuste avec des options simples.

L'objectif recherché est de connecter un client du genre nomade avec un réseau d'entreprise de la manière la plus simple possible tout en étant à haut niveau de sécurité. Nous allons commencer par une configuration minimaliste que nous étofferons petit à petit.

Notre serveur VPN sera à l'adresse **192.168.1.99** et il aura une deuxième interface vers le LAN en **172.31.0.1**. Il portera le doux nom d'**ovpngw**.

Nous aurons un serveur intranet sur le LAN uniquement en **172.31.0.10** dont la passerelle par défaut sera notre serveur VPN, soit **172.31.0.1**. Son petit nom sera **intranet**.

Une troisième VM simulera une station mobile dont le nom sera **roadwarrior**.

Le but de la manœuvre sera de pouvoir connecter la station mobile aux applications hébergées sur le serveur de façon la plus transparente possible.

1. CONFIGURATION

1.1 Le serveur

1.1.1 Génération des certificats en utilisant easy-rsa

La manière la plus simple de générer des certificats serveur et clients consiste à utiliser un utilitaire nommé **easy-rsa** dont le paquet éponyme est normalement une dépendance du paquet **openvpn**. La version d'easy-rsa utilisée ici est la v2. Attention, des différences non triviales existent entre cette version et les autres. Ce paquet met bizarrement les utilitaires dans **/usr/share/easy-rsa**. Personnellement, je recopie le tout dans **/root**, je trouve cela plus simple, mais vous pouvez aussi bien le mettre ailleurs si vous préférez.

Commençons par configurer easy-rsa. Il suffit d'éditer le fichier **easy-rsa/vars** et de renseigner les dernières lignes :

Fichier

```
# Taille de la clé. On peut monter à 4096
export KEY_SIZE=2048
# Validité en jours
export CA_EXPIRE=3650
export KEY_EXPIRE=3650
# À renseigner en fonction de votre situation
export KEY_COUNTRY="US"
export KEY_PROVINCE="CA"
export KEY_CITY="SanFrancisco"
export KEY_ORG="Fort-Funston"
export KEY_EMAIL="me@myhost.mydomain"
export KEY_OU="MyOrganizationalUnit"
# Ici mettre le sujet que l'on veut pour le certificat au format X509
export KEY_NAME="EasyRSA"
```

Une fois les modifications faites, ne pas oublier de les faire prendre en compte par le système avec :

Terminal

```
# cd easy-rsa
# source ./vars
```

La première chose à faire maintenant est de générer les clés et le certificat pour l'autorité de certification racine à moins que vous ayez déjà votre propre CA. On commence par tout nettoyer avec le script **clean-all**. Attention toutes les clés et certificats existants seront

supprimés. Cette commande est à lancer à la première utilisation d'easy-rsa ou quand vous voulez tout reprendre de zéro. Ensuite, il faut lancer la commande `build-ca` qui va vous demander de confirmer les informations saisies dans le fichier `vars`. C'est le moment de les changer si besoin est :

Terminal

```
# ./clean-all
# ./build-ca
Generating a 2048 bit RSA private key
..............................+++
.........................................+++
writing new private key to 'ca.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [FR]:
State or Province Name (full name) [Bretagne]:
Locality Name (eg, city) [Rennes]:
Organization Name (eg, company) [GLMF]:
Organizational Unit Name (eg, section) [BE]:
Common Name (eg, your name or your server's hostname) [GLMF CA]:
Name [EasyRSA]:
Email Address [cpellerin@bs-tech.fr] :
```

On trouve alors dans `easy-rsa/keys` les fichiers suivants :

- ⇨ `ca.crt` qui est le certificat du root CA ;
- ⇨ `ca.key` qui est la paire de clés du root CA ;
- ⇨ `index.txt` la « base de données » d'OpenSSL pour les certificats ;
- ⇨ `serial` contient le numéro de série du dernier certificat généré.

Une fois la CA en place, on peut générer les clés et certificats serveur :

Terminal

```
# ./build-key-server ovpngw
Generating a 2048 bit RSA private key
..............+++
............+++
writing new private key to 'ovpngw.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [FR]:
State or Province Name (full name) [Bretagne]:
Locality Name (eg, city) [Rennes]:
Organization Name (eg, company) [GLMF]:
Organizational Unit Name (eg, section) [BE]:
Common Name (eg, your name or your server's hostname) [ovpngw]:
Name [EasyRSA]:
Email Address [cpellerin@bs-tech.fr]:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

```
Using configuration from /usr/share/easy-rsa/openssl-1.0.0.cnf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
countryName           :PRINTABLE:'FR'
stateOrProvinceName   :PRINTABLE:'Bretagne'
localityName          :PRINTABLE:'Rennes'
organizationName      :PRINTABLE:'GLMF'
organizationalUnitName:PRINTABLE:'BE'
commonName            :PRINTABLE:'ovpngw'
name                  :PRINTABLE:'EasyRSA'
emailAddress          :IA5STRING:'cpellerin@bs-tech.fr'
Certificate is to be certified until Apr 30 15:49:07 2027 GMT (3650 days)
Sign the certificate? [y/n]:y

1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

Puis même chose pour le client avec la commande **`build-key <nom_du_client>`**. Le déroulé est identique à celui ci-dessus, inutile de se répéter à ceci près qu'il faut bien mettre un *Common Name* correspondant au nom que l'on veut donner à la machine cliente vue d'OpenVPN. Chaque poste client devra avoir un *Common Name* unique.

Une fois ceci fait, nous pouvons passer à la partie qui va nous permettre d'aller prendre un chocolat ou autre tout en tapant une belote : la génération du fichier **`dhparam`** pour l'échange de clés utilisant l'algorithme Diffie-Hellman. Il suffit de lancer la commande :

Terminal

```
# ./build-dh
```

... et d'attendre patiemment. En fonction de l'entropie de votre machine, la génération ira plus ou moins vite, mais elle peut prendre plusieurs minutes même sur une machine récente.

Tout ceci terminé, il faut recopier les fichiers **`easy-rsa/keys/dh2048.pem`**, **`easy-rsa/keys/ca.crt`**, **`easy-rsa/keys/ovpngw.crt`** et **`easy-rsa/keys/ovpngw.key`** dans le répertoire **`/etc/openvpn/keys`** du serveur (à créer s'il n'existe pas).

1.1.2 Le fichier de configuration

Le fichier de configuration de base est court. Nous allons l'expliquer point par point, car celui du client que nous verrons plus bas est fort proche. Vous êtes libres de choisir son nom mais, au moins sous **Debian**, l'extension doit être **`.conf`** si vous souhaitez que ce VPN soit lancé automatiquement au démarrage. Ici nous allons l'appeler **`server.conf`** :

Fichier

```
 1 mode server
 2 proto udp
 3
 4 dev tun
 5 topology subnet
 6 ca keys/ca.crt
 7 cert keys/ovpngw.crt
 8 key keys/ovpngw.key
 9 dh keys/dh2048.pem
10 server 10.50.0.0 255.255.255.0
11 keepalive 10 120
12
13 comp-lzo
14 persist-key
15 persist-tun
16 verb 3
```

Prenons-le ligne par ligne :

- 1 : on indique à OpenVPN qu'il va agir en tant que serveur ;
- 2 : le protocole utilisé sera l'UDP, qui est le choix par défaut ;
- 4 : le *device* utilisé ici sera de type `tun`, c'est-à-dire que l'on va monter un tunnel au niveau 3 de la *stack* OSI. On pourra donc encapsuler de l'IP. Pour permettre la transmission d'autres protocoles comme IPX/SPX ou NetBIOS, il faudrait encapsuler au niveau 2 et utiliser un *device* de type `tap` que nous verrons ultérieurement.
- 5 : la topologie est fixée à `subnet` pour se simplifier la vie. En effet, celle par défaut, nommée `net30` est utilisée pour garantir une rétrocompatibilité avec les machines sous Windows, mais son utilisation est nettement plus complexe et n'apporte rien.
- 6 à 9 : on indique où trouver respectivement le certificat du CA, le certificat et la paire de clés du serveur et enfin le fichier de paramètres pour l'échange Diffie-Hellman. Les chemins peuvent être soit absolus, soit relatifs à l'emplacement du fichier de configuration, c'est-à-dire `/etc/openvpn` pour nous.
- 10 : cette simple directive est en fait une sorte de macro qui va remplacer une série de paramètres :
  - passer en mode serveur sécurisé via TLS (directive `tls-server`) ;
  - envoyer la topologie choisie au client (directive `push "topology <topologie serveur>"`) ;
  - positionner correctement les paramètres réseau en fonction des choix effectués concernant le *device* et la topologie.

Dans notre cas, nous indiquons que le *pool* d'adresses IP à utiliser est `10.50.0.0/24` ;

- 11 : encore une macro concernant le système de maintien de connexion et de vérification de la présence des clients en l'absence de trafic. Le premier paramètre de la directive `keepalive` indique le nombre de secondes à attendre après une détection de trafic nul avant d'envoyer un paquet de type `ping` au client. Le second paramètre indique au bout de combien de secondes relancer la connexion sans réception de trafic ni de ping. Il est à noter que ces paquets de ping ne renvoient pas d'écho, le serveur doit *pinger* le client, mais le client doit lui aussi *pinger* le serveur sinon la connexion sera relancée. La macro `keepalive` se charge de positionner les paramètres côté serveur et côté client via deux `push` ;
- 13 : on demande une compression utilisant l'algorithme LZO ;
- 14 : l'option `persist-key` permet d'éviter une renégociation des clés lors d'un redémarrage de connexion. Cette option est surtout importante quand on va demander à OpenVPN de fonctionner sous un autre utilisateur que root. En effet, dans ce cas-là, une fois la diminution de privilèges effectuée, il n'a plus accès aux clés qui sont en `0600` et appartiennent à root ;
- 15 : on demande là de ne pas fermer puis réouvrir le *device* `tun` (ou `tap`) à chaque redémarrage d'une connexion. L'effet de bord est que les éventuels scripts de démarrage/fermetures ne seront pas réexécutés ;
- 16 : pour finir, on passe la verbosité à `3` sur une échelle de `0` à `11` afin de voir un peu ce qui se passe. Elle est à `1` par défaut.

Voilà pour une configuration serveur minimale, mais complètement fonctionnelle à poser dans `/etc/openvpn`.

1.1.3 Le forward de paquets

Comme toute passerelle qui se respecte, un serveur VPN est censé transmettre les paquets d'un réseau à un autre et réciproquement. Nous avons donc besoin d'expliquer à notre machine `ovpngw` ce qu'il faut faire en effectuant la commande suivante :

Terminal

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

Et pour rendre cette modification pérenne, il faut aller modifier le fichier **/etc/sysctl.conf** en décommentant la ligne :

Fichier

```
net.ipv4.ip_forward=1
```

Une option serveur qui peut être intéressante s'appelle **redirect-gateway def1**. Elle ordonne au serveur OpenVPN de se comporter comme passerelle par défaut pour tous les clients. Attention cependant, l'usage de cette directive ne va pas remplacer la route par défaut sur les stations, mais ajouter une route en **0.0.0.0/0** au-dessus de celle par défaut. Ça fonctionne plutôt bien, mais il faut en mesurer les conséquences sur des clients fixes déjà reliés à un réseau local, ne pas oublier de leur pousser les routes locales. D'autre part, il faut penser à pousser aussi la route pour recontacter le serveur OpenVPN en cas de plantage, car il se peut dans ce cas-là que la route par défaut poussée par le serveur reste en place alors que le VPN est tombé.

La solution de secours pour les postes dont on ne veut surtout pas modifier la route par défaut est une combinaison de **route-noexec** et **route-up** à poser dans la configuration client. Le premier interdit au client de tenir compte de toutes les routes poussées tandis que le second exécute un script local censé définir les routes juste après l'authentification de la connexion ou un peu après en fonction de la valeur de la directive **route-delay**.

La directive **redirect-gateway** prend aussi d'autres paramètres intéressants décris dans la documentation officielle **[1]**.

1.2 Les clients

La configuration des clients est plus simple que celle du serveur, car toute la partie PKI est déjà faite. Il ne nous reste qu'à écrire un fichier de configuration compatible avec celui du serveur. Le voici in extenso, nous ne commenterons que les parties qui diffèrent de la configuration serveur :

Fichier

```
 1 client
 2 dev tun
 3 proto udp
 4
 5 remote 192.168.1.99 1194
 6 nobind
 7
 8 persist-key
 9 persist-tun
10
11 ca /etc/openvpn/keys/ca.crt
12 cert /etc/openvpn/keys/roadwarrior1.crt
13 key /etc/openvpn/keys/roadwarrior1.key
14
15 comp-lzo
16 verb 3
```

En ligne 1, on retrouve la directive **client** pour faire pendant à la directive **server** de notre passerelle VPN. L'adresse IP du serveur est indiquée en ligne 5 avec l'option **nobind** en ligne 7 pour éviter qu'OpenVPN n'accapare le port **1194** côté client et nous laisser la

possibilité de pouvoir lancer plusieurs tunnels sur le client si nous le désirons. Les autres options sont identiques à celles du serveur hormis bien sûr le certificat et la clé du client.

Pour configurer d'autres clients, il suffira de recopier ce fichier de configuration, créer un autre certificat et une autre paire de clés sur le serveur, adapter les lignes 12 et 13 et poser le tout sur le nouveau client. Inutile de dire que tout ceci se *package* fort bien tant en DEB qu'en RPM et pour le déploiement, vous avez le choix entre **Ansible**, **Puppet**, **Chef**, etc.

2. DÉMARRER OPENVPN

Une fois que l'on a bien vérifié et revérifié que les fichiers de PKI sont sur les bonnes machines à la bonne place et que les fichiers de configuration y font référence correctement – première cause de soucis – nous pouvons démarrer le serveur à la main histoire de pouvoir consulter son log :

Terminal

```
# cd /etc/openvpn
# openvpn --config ovpngw.conf
Wed May  3 15:23:50 2017 OpenVPN 2.3.4 i586-pc-linux-gnu [SSL (OpenSSL)] [LZO] [EPOLL]
[PKCS11] [MH] [IPv6] built on Nov 19 2015
Wed May  3 15:23:50 2017 library versions: OpenSSL 1.0.1t  3 May 2016, LZO 2.08
Wed May  3 15:23:50 2017 NOTE: your local LAN uses the extremely common subnet address
192.168.0.x or 192.168.1.x.  Be aware that this might create routing conflicts if you
connect to the VPN server from public locations such as internet cafes that use the
same subnet.
Wed May  3 15:23:50 2017 Diffie-Hellman initialized with 2048 bit key
Wed May  3 15:23:50 2017 Socket Buffers: R=[163840->131072] S=[163840->131072]
Wed May  3 15:23:50 2017 TUN/TAP device tun0 opened
Wed May  3 15:23:50 2017 TUN/TAP TX queue length set to 100
Wed May  3 15:23:50 2017 do_ifconfig, tt->ipv6=0, tt->did_ifconfig_ipv6_setup=0
Wed May  3 15:23:50 2017 /sbin/ip link set dev tun0 up mtu 1500
Wed May  3 15:23:50 2017 /sbin/ip addr add dev tun0 10.50.0.1/24 broadcast 10.50.0.255
Wed May  3 15:23:50 2017 UDPv4 link local (bound): [undef]
Wed May  3 15:23:50 2017 UDPv4 link remote: [undef]
Wed May  3 15:23:50 2017 MULTI: multi_init called, r=256 v=256
Wed May  3 15:23:50 2017 IFCONFIG POOL: base=10.50.0.2 size=252, ipv6=0
Wed May  3 15:23:50 2017 Initialization Sequence Completed
```

La ligne importante est la dernière. Si vous obtenez autre chose que « Initialization Sequence Completed », c'est qu'il y a un problème.

La verbosité (directive `verb`) étant à `3`, nous n'avons pas énormément d'informations, mais elles suffisent pour valider les grandes lignes qui sont l'usage de l'interface `tun0` sur laquelle le serveur s'octroie l'adresse `10.50.0.1`, c'est-à-dire la première du *pool* autorisé. On a même droit à un *warning* concernant l'utilisation d'adresses privées pour le réseau sous-jacent. Là on est dans le cadre d'un test donc on le sait, mais ce message peut éviter des problèmes en production.

Passons maintenant à la machine cliente et lançons aussi OpenVPN :

Terminal

```
# cd /etc/openvpn
# openvpn --config roadwarrior.conf
Wed May  3 15:36:06 2017 OpenVPN 2.3.4 x86_64-pc-linux-gnu [SSL (OpenSSL)] [LZO] [EPOLL]
[PKCS11] [MH] [IPv6] built on Nov 12 2015
Wed May  3 15:36:06 2017 library versions: OpenSSL 1.0.1t  3 May 2016, LZO 2.08
Wed May  3 15:36:06 2017 WARNING: No server certificate verification method has been
enabled.  See http://openvpn.net/howto.html#mitm for more info.
Wed May  3 15:36:06 2017 Socket Buffers: R=[212992->131072] S=[212992->131072]
```

```
Wed May  3 15:36:06 2017 UDPv4 link local: [undef]
Wed May  3 15:36:06 2017 UDPv4 link remote: [AF_INET]192.168.1.99:1194
Wed May  3 15:36:06 2017 TLS: Initial packet from [AF_INET]192.168.1.99:1194, sid=ac3f5744
4d8ee1dd
Wed May  3 15:36:06 2017 VERIFY OK: depth=1, C=FR, ST=Bretagne, L=Rennes, O=GLMF, OU=BE,
CN=GLMF CA, name=EasyRSA, emailAddress=cpellerin@bs-tech.fr
Wed May  3 15:36:06 2017 VERIFY OK: depth=0, C=FR, ST=Bretagne, L=Rennes, O=GLMF, OU=BE,
CN=ovpngw, name=EasyRSA, emailAddress=cpellerin@bs-tech.fr
Wed May  3 15:36:06 2017 Data Channel Encrypt: Cipher 'BF-CBC' initialized with 128 bit key
Wed May  3 15:36:06 2017 Data Channel Encrypt: Using 160 bit message hash 'SHA1' for HMAC
authentication
Wed May  3 15:36:06 2017 Data Channel Decrypt: Cipher 'BF-CBC' initialized with 128 bit key
Wed May  3 15:36:06 2017 Data Channel Decrypt: Using 160 bit message hash 'SHA1' for HMAC
authentication
Wed May  3 15:36:06 2017 Control Channel: TLSv1, cipher TLSv1/SSLv3 DHE-RSA-AES256-SHA, 2048
bit RSA
Wed May  3 15:36:06 2017 [ovpngw] Peer Connection Initiated with [AF_INET]192.168.1.99:1194
Wed May  3 15:36:08 2017 SENT CONTROL [ovpngw]: 'PUSH_REQUEST' (status=1)
Wed May  3 15:36:08 2017 PUSH: Received control message: 'PUSH_REPLY,route-gateway
10.50.0.1,topology subnet,ping 10,ping-restart 120,ifconfig 10.50.0.2 255.255.255.0'
Wed May  3 15:36:08 2017 OPTIONS IMPORT: timers and/or timeouts modified
Wed May  3 15:36:08 2017 OPTIONS IMPORT: --ifconfig/up options modified
Wed May  3 15:36:08 2017 OPTIONS IMPORT: route-related options modified
Wed May  3 15:36:08 2017 TUN/TAP device tun0 opened
Wed May  3 15:36:08 2017 TUN/TAP TX queue length set to 100
Wed May  3 15:36:08 2017 do_ifconfig, tt->ipv6=0, tt->did_ifconfig_ipv6_setup=0
Wed May  3 15:36:08 2017 /sbin/ip link set dev tun0 up mtu 1500
Wed May  3 15:36:08 2017 /sbin/ip addr add dev tun0 10.50.0.2/24 broadcast 10.50.0.255
Wed May  3 15:36:08 2017 Initialization Sequence Completed
```

Ce log est tout de suite plus fourni. En effet, nous avons la trace résumée de la connexion avec le serveur comportant les certificats utilisés. Nous remarquons plusieurs choses importantes :

- la validation des deux certificats avec leur contenu (les deux **VERIFY OK**) ;
- en dessous, nous avons le cipher utilisé pour le chiffrement : BF-CBC 128 bits, on peut faire mieux... et l'algorithme d'authentification : SHA-1. Là ce n'est carrément pas bon, le SHA-1 étant considéré comme peu fiable. Mais nous avons laissé les valeurs par défaut, on va améliorer ça très vite ;
- la ligne suivante nous informe du cipher utilisé lui pour le canal de contrôle – multiplexé avec les data – il s'agit d'un cipher AES, extension de TLSv1.0, nommé DHE-RSA-AES256-SHA et ce en 2048 bits ;
- plus bas, nous avons la liste des informations reçues du serveur via un **push**. Ici nous trouvons toutes les informations concernant le réseau ainsi que les **ping** et **ping-restart** qui correspondent à ce qu'a envoyé la macro **keepalive** du serveur ;
- le *device* utilisé côté client est aussi **tun0**, on le voit à plusieurs reprises ;
- l'avant-dernière et l'antépénultième lignes nous informent des commandes systèmes utilisées pour mettre en place la configuration de l'interface **tun0** ;
- la dernière ligne nous permet de dire : « On est connectés ».

Cependant, un gros warning attire notre attention : « No server certificate verification method has been enabled ». Ce message indique que nous avons reçu un certificat serveur qui semble valide et qu'il a bien été signé par la CA que nous avons, mais ce pourrait bien ne pas être le cas. Ce manque de vérification pourrait constituer une porte ouverte à une attaque de type « homme au milieu » plus connue sous son nom d'origine de « Man In The Middle ». Afin de corriger ceci, nous allons tout de suite rajouter dans la configuration client la ligne :

Fichier

```
remote-cert-tls server
```

Celle-ci demande au client d'aller vérifier deux choses sur le serveur :

- premièrement que le certificat serveur possède bien une « X509v3 Extended Key Usage » qui contient la phrase « TLS Web Server Authentication » ;
- deuxièmement que ce même serveur possède aussi une « X509v3 Key Usage » qui contient le *flag* 'Key Encipherment' (de valeur `A0` en hexadécimal).

Si tout ceci vous semble un peu obscur, retenez simplement que cette directive permet d'aller vérifier en profondeur que le certificat reçu émane bien du serveur et non d'une autre machine avec le même `ca.crt` qui voudrait se faire passer pour un serveur valide.

Enfin nous y sommes, le client relancé ne nous insulte plus, mais nous explique que tout va pour le mieux dans le meilleur des mondes :

Terminal

```
...
Wed May  3 15:53:29 2017 VERIFY OK: depth=1, C=FR, ST=Bretagne, L=Rennes,
O=GLMF, OU=BE, CN=GLMF CA, name=EasyRSA, emailAddress=cpellerin@bs-tech.fr
Wed May  3 15:53:29 2017 Validating certificate key usage
Wed May  3 15:53:29 2017 ++ Certificate has key usage  00a0, expects 00a0
Wed May  3 15:53:29 2017 VERIFY KU OK
Wed May  3 15:53:29 2017 Validating certificate extended key usage
Wed May  3 15:53:29 2017 ++ Certificate has EKU (str) TLS Web Server
Authentication, expects TLS Web Server Authentication
Wed May  3 15:53:29 2017 VERIFY EKU OK
Wed May  3 15:53:29 2017 VERIFY OK: depth=0, C=FR, ST=Bretagne, L=Rennes,
O=GLMF, OU=BE, CN=ovpngw, name=EasyRSA, emailAddress=cpellerin@bs-tech.fr
...
```

3. AMÉLIORATIONS

Comme nous venons de le voir, la configuration par défaut d'OpenVPN est totalement fonctionnelle, mais pas optimale. Il est possible d'améliorer cela en de nombreux points même sans aller chercher la petite bête pour le moment.

3.1 Les routes

Les plus observateurs d'entre vous auront pu se rendre compte que si le tunnel se monte et que l'on peut *pinger* chaque bout depuis l'autre, le serveur intranet installé sur le LAN en `172.31.0.10` reste inaccessible depuis notre station mobile. Comme nous voulons être le moins intrusif possible, il n'est pas recommandé de mettre en dur la route vers ce serveur dans notre configuration cliente, l'idéal serait que `ovpngw` puisse pousser vers les clients toutes les routes dont ils auront besoin. Ceci est possible et prévu, il suffit de rajouter une ligne par route dans le fichier de configuration du serveur VPN :

Fichier

```
push "route 172.31.0.0 255.255.255.0"
```

... et magiquement le client est mis au courant. Si on relance les deux côtés, on obtient ceci dans les logs clients :

Terminal

```
...
PUSH: Received control message: 'PUSH_REPLY,route 172.31.0.0
255.255.255.0,route-gateway 10.50.0.1,topology subnet,ping 10,ping-restart
120,ifconfig 10.50.0.2 255.255.255.0'
...
/sbin/ip route add 172.31.0.0/24 via 10.50.0.1
...
```

Si l'on compare avec les logs de la section 2, on remarque que la directive « route 172.31.0.0 255.255.255.0 » a été rajoutée. Nous voyons aussi qu'OpenVPN en tient compte en demandant à Linux de rajouter ladite route.

Une fois ceci fait, si les routes sont bien établies sur le LAN, nous pouvons avoir accès à notre intranet sans souci.

3.2 Des ciphers et des digest

3.2.1 Les ciphers

La liste des ciphers supportés par OpenVPN est assez longue et montre bien sa capacité d'adaptation aux divers besoins :

Terminal

```
# openvpn --show-ciphers

DES-CBC 64 bit default key (fixed)
RC2-CBC 128 bit default key (variable)
DES-EDE-CBC 128 bit default key (fixed)
DES-EDE3-CBC 192 bit default key (fixed)
DESX-CBC 192 bit default key (fixed)
BF-CBC 128 bit default key (variable)
RC2-40-CBC 40 bit default key (variable)
CAST5-CBC 128 bit default key (variable)
RC2-64-CBC 64 bit default key (variable)
AES-128-CBC 128 bit default key (fixed)
AES-192-CBC 192 bit default key (fixed)
AES-256-CBC 256 bit default key (fixed)
CAMELLIA-128-CBC 128 bit default key (fixed)
CAMELLIA-192-CBC 192 bit default key (fixed)
CAMELLIA-256-CBC 256 bit default key (fixed)
SEED-CBC 128 bit default key (fixed)
AES-128-CBC-HMAC-SHA1 128 bit default key (fixed)
AES-256-CBC-HMAC-SHA1 256 bit default key (fixed)
```

La mention **fixed** ou **variable** entre parenthèses permet de savoir si l'on peut changer la taille de la clé en utilisant la directive **keysize**.

Cependant, parmi ces ciphers, certains sont obsolètes voire carrément perméables aux attaques. La cryptographie évolue tous les jours tant du côté des « attaquants » que des « défenseurs », c'est l'éternel combat du canon contre la cuirasse. Afin de se prémunir contre un choix automatique qui pourrait être désastreux, il est possible de spécifier la liste des ciphers que nous autorisons tant pour le canal de contrôle que pour le canal de données. La même chose est bien entendu possible pour les algorithmes de hashage utilisés par les mécanismes d'authentification.

Pour spécifier le cipher à utiliser pour le canal de données, l'option est la suivante :

Fichier

```
cipher <nom_du_cipher>
```

Et pour le canal de contrôle :

Fichier
```
tls-cipher <cipher1:cipher2:...:ciphern>
```

Par exemple, on peut rajouter les trois lignes suivantes aux fichiers de configuration serveur et client :

Fichier
```
tls-version-min 1.2
cipher AES-256-CBC
tls-cipher TLS-ECDH-RSA-WITH-AES-256-GCM-SHA384
```

Là on se dit qu'on est tranquille. On relance le serveur, tout va bien, puis le client et là, ô surprise, la négociation ne passe pas le cap de :

Terminal
```
TLS: Initial packet from [AF_INET]192.168.1.99:1194, sid=d1682079 98f81b6f
```

Et le serveur nous insulte copieusement :

Terminal
```
TLS_ERROR: BIO read tls_read_plaintext error: error:1408A0C1:SSL routines:SSL3_
GET_CLIENT_HELLO:no shared cipher
```

Mais que se passe-t-il ? Nous avons bien spécifié les deux mêmes ciphers TLS des deux côtés et ils sont supportés par notre version d'OpenSSL :

Terminal
```
# openssl ciphers -v | grep -i ECDH-RSA
ECDH-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH/RSA Au=ECDH Enc=AESGCM(256) Mac=AEAD
...
```

Dans le cipher choisi, deux lettres font la différence : **EC**. Ces lettres signifient *Elliptic Curve*. Or nos certificats n'ont pas été créés avec cet algorithme :

Terminal
```
# openssl x509 -text -noout -in /root/roadwarrior.crt
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 2 (0x2)
...
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
```

Un certificat créé avec un algorithme de type EC afficherait :

Terminal
```
Public Key Algorithm: id-ecPublicKey
```

Nous ne pouvons donc pas faire usage des ciphers utilisant les algorithmes de type EC. Qu'avons-nous d'autre de disponible dans le même genre ?

Terminal
```
# openvpn --show-tls | grep RSA-WITH-AES-256-GCM-SHA384
TLS-ECDHE-RSA-WITH-AES-256-GCM-SHA384
TLS-DHE-RSA-WITH-AES-256-GCM-SHA384
TLS-ECDH-RSA-WITH-AES-256-GCM-SHA384
TLS-RSA-WITH-AES-256-GCM-SHA384
```

La deuxième ligne semble prometteuse, la même suite de cryptographie que voulue, mais sans *elliptic curve*. Essayons-la et, ô miracle, la connexion s'établit sans aucun souci :

Terminal
```
...
Data Channel Encrypt: Cipher 'AES-256-CBC' initialized with 256 bit key
...
Control Channel: TLSv1.2, cipher TLSv1/SSLv3 DHE-RSA-AES256-GCM-SHA384, 2048 bit RSA
```

On retrouve bien notre cipher.

Attention donc à l'adéquation entre le ou les ciphers demandés et ceux utilisés pour les certificats. On peut parfaitement générer des certificats utilisant des algorithmes en *elliptic curve*, mais pour cela il faut le faire à la main et se passer d'easy-rsa. Cette manipulation est détaillée dans le prochain article concernant les options avancées d'OpenVPN.

À noter qu'il existe un « cipher » spécial nommé **none** pour interdire tout chiffrement. Ça peut servir…

Pour information aussi, OpenVPN est capable d'utiliser quelques cartes accélératrices de chiffrement. Pour le moment, seules les cartes RSAX et Intel RDRAND sont nativement supportées. Cependant, il est possible de rajouter son propre module de support, mais cela dépasse très largement le cadre de cet article.

3.2.2 Les digests

De la même façon que nous pouvons choisir nos ciphers, nous pouvons aussi choisir le digest d'authentification pour les données avec la directive :

Fichier
```
auth <digest>
```

La liste des digests supportés par OpenVPN s'affiche avec la commande :

Terminal
```
# openvpn --show-digests
MD5 128 bit digest size
RSA-MD5 128 bit digest size
SHA 160 bit digest size
RSA-SHA 160 bit digest size
SHA1 160 bit digest size
RSA-SHA1 160 bit digest size
DSA-SHA 160 bit digest size
DSA-SHA1-old 160 bit digest size
DSA-SHA1 160 bit digest size
RSA-SHA1-2 160 bit digest size
DSA 160 bit digest size
RIPEMD160 160 bit digest size
RSA-RIPEMD160 160 bit digest size
MD4 128 bit digest size
RSA-MD4 128 bit digest size
ecdsa-with-SHA1 160 bit digest size
RSA-SHA256 256 bit digest size
RSA-SHA384 384 bit digest size
RSA-SHA512 512 bit digest size
RSA-SHA224 224 bit digest size
SHA256 256 bit digest size
SHA384 384 bit digest size
SHA512 512 bit digest size
SHA224 224 bit digest size
whirlpool 512 bit digest size
```

… ce qui en fait quelques-uns.

Si nous voulons éviter l'usage de SHA1 par défaut, rajoutons par exemple aux fichiers de configuration la ligne :

Fichier
```
auth SHA256
```

Une fois le serveur et le client redémarrés, nous pouvons vérifier que nos ordres sont bien exécutés :

Fichier
```
Data Channel Encrypt: Using 256 bit message hash 'SHA256' for HMAC authentication
```

3.3 Les logs

Si on laisse comme ça, les logs d'OpenVPN vont se retrouver dans syslog. C'est un comportement par défaut qui en vaut un autre. Pour changer cette option, il faut utiliser la directive :

Fichier
```
log <chemin_fichier_log>
```

Par exemple :

Fichier
```
log /var/log/openvpn.log
```

Comme nous l'avons déjà vu, la verbosité des logs se règle via la directive :

Fichier
```
verb <0-11>
```

Ceci a les conséquences suivantes :

- ⇨ **0** : Aucun log sauf pour les erreurs fatales ;
- ⇨ **1** à **4** : Utilisation normale, attention **4** commence à devenir bien verbeux ;
- ⇨ **5** : Imprime les indications **R** ou **W** pour chaque paquet respectivement reçu ou émis, en majuscule pour les paquets TCP ou UDP et en minuscule pour les paquets TUN ou TAP, ce qui donne ce genre de log :

Terminal
```
...
Thu May 11 14:28:33 2017 us=726927 roadwarrior/192.168.1.132:36229 MULTI: Learn:
86:79:3d:3d:66:15 -> roadwarrior/192.168.1.132:36229
wRwRwRwRwRwWRRwRwWRWRWRWRWRWRWRWRWRWRRwrWRwrWRwrWRwrWRwrWRwrWRwrWRwRwrWRwrWRwr
WRwrWWR
```

La partie en rouge correspond à une série de pings venant du client.

- ⇨ **6** à **11** : À n'utiliser que pour du debug, car le nombre de lignes générées par seconde devient énorme.

Attention si vous utilisez cette directive, quel que soit le niveau de log demandé, OpenVPN n'écrira plus rien sur la console, ce qui peut être gênant si vous le lancez à la main via la commande :

Terminal
```
# openvpn --config maconfig.conf
```

... tout en espérant visualiser les logs à l'écran. Cette méthode de démarrage force OpenVPN à rester en avant-plan et est fort utile pour le déverminage et les essais.

Vous remarquerez vite que dans ses logs OpenVPN se répète, voire à tendance parfois à bégayer. Pour éviter la répétition de lignes identiques, il faut utiliser la directive :

Fichier
```
mute <nombre de lignes>
```

Par exemple, la commande suivante supprimera des logs jusqu'à 20 lignes identiques à la précédente :

Fichier
```
mute 20
```

Il est aussi possible de savoir en temps quasi-réel quels clients sont connectés et avec quelle IP sans avoir besoin d'éplucher les logs. Pour ce faire, il existe l'option :

Fichier
```
status <chemin_fichier_status>
```

Son contenu ressemble à ça :

Terminal
```
OpenVPN CLIENT LIST
Updated,Thu May 11 14:53:43 2017
Common Name,Real Address,Bytes Received,Bytes Sent,Connected Since
roadwarrior,192.168.1.132:41387,7245,7651,Thu May 11 14:49:42 2017
ROUTING TABLE
Virtual Address,Common Name,Real Address,Last Ref
10.50.0.2,roadwarrior,192.168.1.132:41387,Thu May 11 14:49:59 2017
GLOBAL STATS
Max bcast/mcast queue length,0
END
```

On voit donc ici tout ce qui concerne notre client nommé `roadwarrior` : son adresse IP réelle (`192.168.1.132` ici) avec le port côté client, le nombre d'octets reçus et émis, l'heure et la date de connexion puis deux lignes plus bas, son adresse IP dans le tunnel.

Attention, il faut parfois attendre plusieurs dizaines de secondes avant que le fichier ne soit mis à jour, d'où la notion de « temps quasi-réel » signalée plus haut.

3.4 Les options pour les clients

Lorsque l'on connecte plusieurs clients à un serveur OpenVPN, il faut savoir que, pour des raisons de sécurité, les clients ne peuvent pas se voir entre eux. Si l'on désire qu'ils le puissent, il suffit de rajouter sur le serveur la directive :

Fichier
```
client-to-client
```

Il est aussi possible de limiter le nombre de clients connectés simultanément grâce à l'option :

Fichier
```
max-clients <nombre_de_clients>
```

Quand un poste distant se connecte au serveur, il se voit attribuer une adresse IP prise dans le *pool* spécifié dans la ligne `server` et rien ne nous dit que cette adresse ne changera pas d'une connexion à l'autre. D'autre part, il est possible de vouloir affiner les options pour certains clients, soit en fonction de leur système d'exploitation, soit en fonction d'autres critères. Il existe une option pour cela :

Fichier
```
client-config-dir <chemin_vers_repertoire>
```

Admettons que nous ayons mis en place cette option ainsi :

Fichier
```
client-config-dir /etc/openvpn/ccd
```

Nous aurons alors la possibilité de mettre dans ce répertoire des fichiers portant le même nom que celui indiqué par le *Common Name* du certificat du client. En cas de besoin, vous pouvez relire la section 1.1.1. Par exemple, notre poste client s'est vu attribuer un certificat dont le CN est `roadwarrior`, on va donc créer un fichier nommé `/etc/openvpn/ccd/roadwarrior` dans lequel on pourra mettre une série d'options spécifiques aux clients. Il s'agit des options `push`, `push-reset`, `iroute`, `ifconfig-push` et `config`.

Nous avons déjà rencontré l'option `push` qui permet de passer tout une série d'options au client comme les routes, le type de topologie, etc. Par défaut, la configuration globale en pousse un certain nombre et la directive `push-reset` est là pour indiquer au client de ne pas en tenir compte, mais de n'utiliser que les `push` suivants. Cette directive est bien entendu à usage exclusif des fichiers `ccd`.

La directive `ifconfig-push <adresse masque>` permet d'envoyer au client son adresse IP et son masque de sous-réseau dans le VPN. C'est comme cela que l'on peut figer l'adresse IP des postes distants. Attention quand même à ce qu'il n'y ait pas de recouvrement entre les adresses envoyées ainsi et celle du pool global, OpenVPN n'effectuant aucune vérification à ce niveau.

L'option `iroute` est assez spéciale, elle permet de créer une route du serveur vers un sous-réseau auquel appartient un client spécifique. Elle doit être utilisée en concordance avec la directive route qui, elle, s'occupe de router les paquets du kernel vers OpenVPN alors que `iroute` les envoie d'OpenVPN vers le poste distant. Si vous désirez que d'autres clients voient ce sous-réseau, il faut le leur indiquer via un `push "route..."` sans oublier le `client-to-client`. Dans ce cas, le `push "route..."` peut être global, car OpenVPN enverra cette route vers tous les clients sauf vers celui concerné par `iroute`.

Pour finir, la directive `config` permet de pousser vers le client le nom d'un fichier de configuration additionnel d'où des directives spécifiques pourront être extraites.

4. TUNNELS AU NIVEAU 2

Bien que le titre puisse le laisser penser, nous ne traiterons pas ici des passages secrets au niveau 2 de **Duke Nukem Forever**, mais bien de la possibilité offerte par OpenVPN de *tunneliser* des protocoles de niveau 2 comme Ethernet, IEEE802.11 (le wifi), FDDI ou ARCnet.

4.1 La configuration réseau du serveur

Afin de pouvoir travailler au niveau Ethernet, nous allons être obligés d'utiliser un *device* de type **tap** et de le « lier » à notre interface réseau physique grâce à l'utilisation d'un *bridge*. Pour ce faire, il faut installer, si ce n'est déjà fait, le paquet **bridge-utils** qui va nous apporter l'utilitaire **brctl** dont nous allons avoir besoin.

Commençons ensuite par créer un *bridge* que nous allons nommer **brvpn** :

Terminal
```
# brctl addbr brvpn
```

Puis nous y ajoutons notre interface extérieure, par exemple **eth1**, qui ne doit pas être déjà configurée. En effet c'est **brvpn** qui portera l'adresse IP externe et non plus **eth1** :

Terminal
```
# brctl addif brvpn eth1
```

Et demandons à OpenVPN de nous créer l'interface **tap0** qui n'existe pas encore afin de pouvoir l'insérer ensuite dans le *bridge* afin qu'elle partage ses paquets avec **eth1** :

Terminal
```
# openvpn --mktun --dev tap0
Thu May 11 11:43:57 2017 TUN/TAP device tap0 opened
Thu May 11 11:43:57 2017 Persist state set to: ON
# brctl addif brvpn tap0
```

Et pour finir, il faut passer l'interface **tap0** en mode **promiscuous**. Ce mode ordonne à l'interface d'accepter tous les paquets y compris ceux qui ne lui sont pas destinés :

Terminal
```
# ip link set tap0 up promisc on mtu 1500
```

Notre configuration est presque prête, nous n'avons plus qu'à configurer notre nouvelle interface **brvpn** au niveau IP :

Terminal
```
# ifconfig brvpn 172.31.0.1/24
```

Et nous sommes prêts.

4.2 Configurer OpenVPN

4.2.1 Configuration manuelle et tests

Dans ce cas de figure, la configuration est à peine plus compliquée. Tout d'abord, notre fichier de configuration serveur doit ressembler à ceci pour le moment :

Fichier
```
1 mode server
2 proto udp
3 dev tun
4 topology subnet
5 ca keys/ca_glmf.crt
6 cert keys/ovpngw.crt
```

```
 7 key keys/ovpngw.key
 8 dh keys/dh2048.pem
 9 server 10.50.0.0 255.255.255.0
10 keepalive 10 120
11 comp-lzo
12 persist-key
13 persist-tun
14 verb 3
15 push "route 172.31.0.0 255.255.255.0"
16
17 tls-version-min 1.2
18 cipher AES-256-CBC
19 tls-cipher TLS-DHE-RSA-WITH-AES-256-GCM-SHA384
20 auth sha256
21 mute 20
22 status /tmp/ovpn.status
23 log /var/log/ovpn.log
```

Il va falloir remplacer la ligne 1 par :

Fichier

```
server-bridge 172.31.0.1 255.255.255.0 172.31.0.100 172.31.0.200
```

Dans cette ligne on retrouve l'adresse IP externe de notre passerelle, le masque de sous-réseau associé et on ajoute la première et la dernière adresse IP du *pool* pour les clients.

Ensuite la ligne 3 sera remplacée par :

Fichier

```
dev tap0
```

Il faut préciser le *device* au complet, car il nous incombe de dire à OpenVPN sur quelle interface il doit travailler.

Pour finir, on peut supprimer les lignes 9 et 15 qui ne nous concernent plus. Une fois ceci fait, on peut relancer le serveur et passer à la configuration des clients.

Du côté des clients, il n'y a pas grand-chose à modifier mis à part la ligne :

Fichier

```
dev tun
```

Celle-ci devient :

Fichier

```
dev tap
```

Ici on ne précise pas le numéro… et le tour est joué. Le client relancé, il devrait se connecter sans souci et on verra apparaître une interface **tap0** avec une adresse IP du genre **172.31.0.100**. Le serveur en **172.31.0.1** devrait être *pingable* sans problème. On notera une légère différence dans le fichier de status en ce qui concerne l'adresse virtuelle qui n'est plus une adresse IP, mais une adresse MAC, preuve que l'on travaille bien maintenant au niveau de la couche 2 du modèle OSI :

Terminal

```
# cat /tmp/ovpn.status
OpenVPN CLIENT LIST
```

```
Updated,Thu May 11 14:44:53 2017
Common Name,Real Address,Bytes Received,Bytes Sent,Connected Since
roadwarrior,192.168.1.132:60203,6528,5925,Thu May 11 14:43:53 2017
ROUTING TABLE
Virtual Address,Common Name,Real Address,Last Ref
7e:77:fb:ca:f3:59,roadwarrior,192.168.1.132:60203,Thu May 11 14:43:55 2017
GLOBAL STATS
Max bcast/mcast queue length,0
END
```

4.2.2 Automatisation

Contrairement au tunnel niveau 3 qui ne nécessite que la configuration d'OpenVPN, au niveau 2 il est nécessaire de modifier les paramètres réseau. Pour automatiser la création du *bridge* sur une Debian, il faut aller modifier le fichier **`/etc/network/interfaces`** comme ceci :

Fichier

```
auto eth0
iface eth0 inet dhcp

auto brvpn
iface brvpn inet static
 address 172.31.0.1
 netmask 255.255.255.0
 bridge_ports eth1 tap0
 pre-up openvpn --mktun --dev tap0
 post-down openvpn --rmtun --dev tap0

auto eth1
iface eth1 inet manual
```

Ici l'interface **`eth0`** est le côté réseau local de la passerelle et l'interface **`eth1`** est le côté internet. C'est donc cette dernière que l'on va associer avec l'interface **`tap0`** dans le bridge nommé **`brvpn`**. Il faut faire très attention à mettre **`brvpn`** en **`auto`** et non en **`allow-hotplug`** et **`eth1`** en **`auto`** et en **`inet manual`** sinon ça ne fonctionnera pas. Les lignes de **`pre-up`** et de **`post-down`** sont là pour demander à OpenVPN de créer, respectivement supprimer, le device **`tap0`**. Une fois ces modifications effectuées et un reboot plus tard afin d'être vraiment certain que tout se déroule automatiquement, sans anicroche on devrait se retrouver avec ceci comme configuration réseau :

Terminal

```
# ifconfig
brvpn     Link encap:Ethernet  HWaddr 08:00:27:36:5c:d8
          inet addr:172.31.0.1  Bcast:172.31.0.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe36:5cd8/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:163 errors:0 dropped:0 overruns:0 frame:0
          TX packets:59 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:9712 (9.4 KiB)  TX bytes:3442 (3.3 KiB)

eth0      Link encap:Ethernet  HWaddr 08:00:27:7e:3d:d9
          inet addr:192.168.1.99  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe7e:3dd9/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1898 errors:0 dropped:0 overruns:0 frame:0
          TX packets:758 errors:0 dropped:0 overruns:0 carrier:0
```

```
          collisions:0 txqueuelen:1000
          RX bytes:153343 (149.7 KiB)  TX bytes:78453 (76.6 KiB)

eth1      Link encap:Ethernet  HWaddr 08:00:27:36:5c:d8
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:170 errors:0 dropped:0 overruns:0 frame:0
          TX packets:61 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:12200 (11.9 KiB)  TX bytes:3996 (3.9 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

tap0      Link encap:Ethernet  HWaddr 26:92:7f:f4:47:f1
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:15 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:1222 (1.1 KiB)  TX bytes:714 (714.0 B)
```

On retrouve principalement notre *bridge* créé automatiquement. Pour vérifier qu'il a bien les bonnes interfaces :

Terminal

```
# brctl show
bridge name bridge id  STP enabled interfaces
brvpn  8000.080027365cd8 no  eth1
        tap0
```

Maintenant, il est possible de faire passer absolument n'importe quel protocole de niveau 3 sur notre VPN. À nous les parties de **Doom** ou la configuration de serveurs **Novell Netware** :-)

CONCLUSION

Nous venons de passer en revue plusieurs aspects simples d'OpenVPN. Il existe une multitude d'options supplémentaires que nous aborderons en partie dans le prochain article, les passer toutes en revue nécessiterait un livre entier dont la première moitié serait consacrée à la cryptographie. Si vous ne trouvez pas ici ce dont vous avez besoin, n'hésitez pas à fouiller la documentation officielle **[1]** qui est fort bien faite, de même que le wiki **[2]**.

Pour ceux qui désireraient une interface graphique, il existe une version propriétaire d'OpenVPN qui est payante et apporte deux ou trois fonctionnalités supplémentaires. Ceci dit nous avons déjà de quoi faire largement avec la version communautaire. Bon courage et bonnes explorations. ■

RÉFÉRENCES

[1] Documentation officielle :
https://community.openvpn.net/openvpn/wiki/Openvpn23ManPage

[2] Wiki : **https://community.openvpn.net/openvpn/wiki**

ACTUELLEMENT DISPONIBLE

HACKABLE N°20 !

CRÉEZ UNE VEILLEUSE QUI MONTRE LES PHASES DE LA LUNE

NE LE MANQUEZ PAS

CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

http://www.ed-diamond.com

UTILISEZ LES OPTIONS AVANCÉES D'OPENVPN

Cédric PELLERIN – [Développeur embarqué senior chez Silicom]

Comme nous l'avons vu, il est tout à fait possible d'utiliser OpenVPN avec un fichier de configuration simple et légèrement adapté et en utilisant easy-rsa pour générer les clés et les certificats. Nous allons voir ici qu'il existe de nombreuses autres possibilités très intéressantes permettant d'augmenter significativement le niveau de sécurité et la souplesse d'utilisation.

Cet article fait suite à celui abordant les aspects élémentaires d'**OpenVPN** et a pour but de dévoiler certains aspects parfois peu connus de ce logiciel. Nous aborderons aussi la création des clés et des certificats de manière relativement détaillée.

Afin de savoir où nous allons, nous commencerons par poser les quelques exigences que voici :

1. Le serveur OpenVPN écoute sur le port TCP **21110**. Ce port est pris au hasard, le but étant d'éviter qu'il écoute sur son port habituel – le **1194** – c'est plus discret ;
2. Le certificat du serveur sera autosigné. C'est plus simple et ça suffit largement dans la plupart des cas ;
3. La clé privée sera de type ECDSA (*Elliptic Curve DSA*) et la courbe elliptique sera de type NIST/SECG 384 bits (secp384r1) afin d'utiliser des « ciphers » récents et bien robustes (pour le moment...) ;
4. Le fichier de paramétrage Diffie-Hellman aura une taille de clé de 2048 bits, 1024 devenant un peu insuffisant ;
5. L'authentification se fera par certificats X509 chiffrés en AES-256 et authentifiés par SHA-256, car SHA-1 est devenu trop léger ;
6. Les suites cryptographiques autorisées sont TLS_ECDH-ECDSA-AES256-SHA ou TLS_ECDH-RSA-AES256-SHA et aucune autre pour les mêmes raisons que pour l'exigence n°3 ;
7. L'authentification des clients sera basée sur un trio utilisateur/mot de passe/certificat dans le but de renforcer encore la sécurité ;
8. Le sujet (*Subject* ou *Distinguished Name*) des certificats clients sera de type **/CN = GLMF-xxxx** où **xxxx** est un nombre à 4 chiffres unique pour chaque client, une sorte de numérotation de ceux-ci. L'utilisation d'un sujet simple et unique par client nous permettra de simplifier le traitement lié à l'exigence précédente ;
9. Le serveur aura la charge de vérifier que le certificat client a bien été signé par son autorité de confiance, qu'il n'a pas été révoqué et qu'il n'a pas expiré ;
10. La topologie sera de type « subnet » afin d'offrir un vrai sous-réseau à nos clients ;
11. Une base de données contiendra la liste des clients autorisés à se connecter et un champ reflètera en temps réel l'état connecté/déconnecté des clients dans le but de pouvoir servir de base à une supervision et aussi de pouvoir interdire temporairement la connexion de certains clients ;
12. La configuration d'OpenVPN sera stockée dans **/etc/openvpn**, les clés et certificats dans le sous-répertoire **keys/**.

1. INSTALLATION DE BASE

1.1 Configuration du réseau

Notre réseau de test sera constitué par deux machines, un serveur dont l'adresse sur le LAN sera **172.25.10.1** et un client à l'adresse **172.25.10.10**.

Pour être clair, quand on devra rentrer des commandes sur le serveur, le prompt indiquera :

Terminal

```
server #
```

Et sur le client, nous aurons :

Terminal
```
client #
```

Sauf indication contraire, les commandes seront à effectuer par l'utilisateur root ou via `sudo`.

NOTE

Tout ce qui va suivre s'applique aux versions 2.3 et supérieures d'OpenVPN. La majeure partie de cet article devrait s'appliquer aussi aux versions 2.0+ (sauf la partie IPv6), mais ce n'est pas garanti. En cas de doute, le wiki OpenVPN [3] est une excellente source.

1.2 Installation et configuration initiale du serveur

Afin de démarrer proprement, nous allons commencer par installer OpenVPN de base sur les deux machines via le système de paquets de notre distribution, ici une **Debian 8.6** fournissant un OpenVPN 2.3.4 (une version 2.4.0 est disponible en Debian Stretch). Si vous utilisez une autre distribution, tout le reste de cet article est totalement agnostique et donc applicable pour vous sans modification.

Pour démarrer sur de bonnes bases, recopions les exemples de configuration que l'on trouve sur le site à l'adresse **[1]**.

Dans la mesure où nous pouvons le faire de façon simple, nous allons essayer de respecter déjà quelques-unes de nos exigences. Pour les autres, nous les mettrons en place petit à petit.

En ce qui concerne la configuration serveur, il suffit juste de modifier sept lignes :

- ⇨ modifier le n° de port sur la ligne `port 1194` qui devient `port 21110` ;
- ⇨ passer la ligne `proto udp` en `proto tcp` ;
- ⇨ changer la ligne `dh dh1024.pem` en `dh keys/dh2048.pem` ;
- ⇨ modifier le *pool* d'adresses VPN : `server 10.50.0.0 255.255.255.0` ;
- ⇨ indiquer que les clés et certificats sont dans le sous-répertoire `keys` en modifiant aussi les trois lignes :

Fichier
```
ca keys/ca.crt
cert keys/server.crt
key keys/server.key
```

Cela nous donne, une fois expurgée des commentaires, la configuration suivante :

Fichier
```
port 21110      # Exigence 1
proto tcp     # Protocole TCP (exigence 1)
dev tun
ca keys/ca.crt       # Exigence 12
cert keys/server.crt     # idem
key keys/server.key     # idem
dh keys/dh2048.pem     # DH en 2048 bits (exigences 4)
```

```
server 10.50.0.0 255.255.255.0# pool d'adresses IP pour les clients
ifconfig-pool-persist ipp.txt
keepalive 10 120
comp-lzo
persist-key
persist-tun
status openvpn-status.log
verb 3
```

Nous noterons ici l'absence de spécification pour l'adresse IP du serveur. Elle sera déduite de la ligne `server 10.50.0.0 255.255.255.0` et OpenVPN lui affectera la première du *pool*, soit 10.50.0.1. C'est très bien, mais parfois on aimerait pouvoir modifier ce comportement. Pour cela, il faut « éclater » la directive `server` en :

Fichier

```
ifconfig 10.50.0.254 255.255.255.0
ifconfig-pool 10.50.0.1 10.50.0.250
route 10.50.0.0 255.255.255.0
```

Ceci nous pose le serveur en `10.50.0.254`. Il est à noter que le serveur est obligatoirement à une « extrémité » ou à l'autre du *pool*. Dans notre cas, vu qu'il s'agit d'un `/24`, il ne pourra être que en `.1` ou en `.254`. Le jeu en vaut-il la chandelle ? À vous de juger en fonctions de vos besoins.

1.3 Création de la PKI

Pour commencer notre *Public Key Infrastructure*, nous devons tout d'abord générer une paire de clés répondant à nos exigences pour notre autorité de confiance. Comme indiqué dans la configuration d'OpenVPN, tous nos certificats et clés seront stockés, pour le moment, dans `/etc/openvpn/keys` afin de ne pas les mélanger avec le fichier de configuration. Bien entendu, toutes les commandes données ici sont valables pour **OpenSSL** comme pour **LibreSSL** :

Terminal

```
server # openssl ecparam \
                        -genkey \
                        -name secp384r1 \
                        -noout \
                        -out /etc/openvpn/keys/ca.key
```

Ceci va créer le fichier `ca.key` qui constitue la paire de clés de notre autorité de certification. Ce fichier servira à signer tous les autres certificats, clients comme serveur.

Les options utilisées sont :

- commande `ecparam` pour indiquer que nous allons générer (ou modifier) une clé de type EC (courbes elliptiques) ;
- option `-genkey` pour générer une paire de clés ;
- option `-name secp384r1` qui indique quel type d'EC nous voulons (liste disponible via la commande `openssl ecparam -list_curves`) ;
- option `-noout` pour indiquer à OpenSSL de ne pas nous afficher les clés ;
- option `-out` qui précise le fichier de sortie, ici donc `ca.key`.

Nous pouvons maintenant examiner notre paire de clés ainsi :

Terminal

```
server # openssl ec -in /etc/openvpn/keys/ca.key -text -noout
read EC key
Private-Key: (384 bit)
priv:
    00:f2:d9:52:63:a0:c7:00:eb:6d:8a:39:72:64:43:
    f2:dd:c0:e4:23:76:13:80:2a:82:3a:26:b9:81:96:
    ed:67:71:9b:eb:aa:8d:cd:ee:25:c5:b1:4a:c9:00:
    cc:d0:cb:4c
pub:
    04:ed:2a:92:36:88:04:d4:69:69:a1:93:1d:7e:f0:
    bf:05:72:3f:0a:5f:c9:d9:a4:37:db:e3:db:a5:56:
    8f:1a:f8:62:3c:43:19:55:cb:f6:ea:74:20:3c:c0:
    26:2c:c2:91:a6:12:b5:09:73:d6:06:7c:1e:e9:ac:
    c8:c7:ec:80:63:4c:61:c2:43:a1:d1:28:c8:33:05:
    f5:c7:ce:98:59:82:d2:ad:27:8d:0f:ff:02:d4:bc:
    41:a3:84:ff:95:90:6f
ASN1 OID: secp384r1
```

On retrouve bien la clé privée, la clé publique et elles sont bien de type secp384r1 (exigence 3).

La commande **openssl** se décompose ainsi :

- ⇨ commande **ec** pour consulter et manipuler les clés de type EC ;
- ⇨ option **-in** pour spécifier le fichier d'entrée ;
- ⇨ option **-text** pour demander un affichage lisible des clés.

Il nous faut maintenant générer le certificat correspondant, ce qui se fait en deux étapes ; génération d'un « certificate signing request » ou CSR puis signature de celui-ci par la même clé (auto-signature) afin de donner un certificat final :

Terminal

```
server # openssl req \
                        -new \
                        -sha256 \
                        -key /etc/openvpn/keys/ca.key \
                        -out /etc/openvpn/keys/ca.csr \
                        -subj "/CN=glmf.vpn"
```

Où :

- ⇨ commande **req** pour indiquer que nous allons demander un CSR ;
- ⇨ option **-new** pour un nouveau ;
- ⇨ option **-sha256** qui désigne le « message digest » qui servira à valider chaque message en le « hashant » (exigence 5) ;
- ⇨ option **-subj** qui permet de spécifier en ligne de commandes le sujet et éviter ainsi qu'OpenSSL nous pose plusieurs questions.

Et enfin :

Terminal

```
server # openssl x509 \
                        -req \
                        -sha256 \
                        -days 365 \
                        -in /etc/openvpn/keys/ca.csr \
                        -signkey /etc/openvpn/keys/ca.key \
                        -out /etc/openvpn/keys/ca.crt
```

```
Signature ok
subject=/CN=glmf.vpn
Getting Private key
```

La commande **x509** permet de gérer les certificats à ce format tandis que l'option **-days** précise le nombre de jours de validité et l'option **-signkey** précise quelle clé il faut utiliser pour signer le CSR.

Voilà, notre autorité de certification est prête, nous pouvons maintenant créer les clés et le certificat pour notre serveur :

Terminal

```
server # openssl ecparam \
                        -genkey \
                        -name secp384r1 \
                        -noout \
                        -out /etc/openvpn/keys/server.key
```

Exactement la même commande que pour notre CA, seul le fichier de destination change.

En ce qui concerne la génération du CSR puis la signature d'icelui, il va être plus simple de créer un petit fichier de configuration qui nous évitera de passer un grand nombre d'options en ligne de commandes. Appelons-le **/etc/openvpn/keys/server_ssl.cnf** :

Fichier

```
[ ca ]
default_ca = CA_default

[ CA_default ]
dir = /etc/openvpn/keys
new_certs_dir = $dir
unique_subject = no
certificate = $dir/ca.crt
database = $dir/index
private_key = $dir/ca.key
serial = $dir/serial
default_days = 365
default_md = sha256
policy = ca_policy
x509_extensions = ca_extensions
copy_extensions = copy
crlnumber = $dir/crlnumber
default_crl_days = 1825

[ ca_policy ]
countryName = optional
stateOrProvinceName = optional
localityName = optional
organizationName = optional
organizationalUnitName = optional
commonName = supplied
emailAddress = optional

[ ca_extensions ]
basicConstraints = CA:false
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer

[ req ]
prompt = no
```

```
encrypt_key = no
default_md = sha256
distinguished_name = dn
req_extensions = req_ext

[ dn ]
CN = glmf.vpn

[ req_ext ]
keyUsage = digitalSignature, keyAgreement
extendedKeyUsage = serverAuth
```

On y trouve l'entête `[ req ]` qui regroupe les options pour la commande `req` d'openssl :

- `prompt = no` ne demande pas le dn (il est indiqué plus bas) ;
- `encrypt_key = no` ne protège pas la clé privée par une *passphrase* qu'il faudrait rentrer à chaque démarrage d'OpenVPN ;
- `default_md = sha256` afin de correspondre à nos exigences ;
- `distinguished_name = dn` pointe sur la section `[ dn ]` où celui-ci est indiqué ;
- `req_extensions = req_ext` pointe sur la section `[ req_ext ]` où les extensions sont spécifiées. Ici, nous nous indiquons que ce certificat pourra servir à deux choses :
  - appliquer une signature numérique, ce qui est utile dans le cadre d'une authentification de l'émetteur d'un message et des données qui sont envoyées ;
  - utiliser un protocole d'échange de clé (*key agreement*) comme Diffie-Hellman dont nous allons nous servir (et nous ajoutons un `extendedKeyUsage` positionné à `serverAuth` afin que si le client utilise la directive `remote-cert-tls` il lui soit bien renvoyé « TLS Web Server Authentication »).

L'utilisation de variables est possible, voir ici la variable `$dir`.

Nous pouvons maintenant créer notre CSR grâce à la commande :

Terminal

```
server # openssl req \
                        -new \
                        -config /etc/openvpn/keys/server_ssl.cnf \
                        -key /etc/openvpn/keys/server.key \
                        -out /etc/openvpn/keys/server.csr
```

Il n'y a rien de bien différent si l'on compare avec la même action pour le CA, juste l'usage du fichier de configuration en plus qui évite les options `-sha256` et `-subj`. Ces deux paramètres sont, avec quelques autres, définis dans le fichier de configuration.

Passons maintenant à la signature de ce CSR. Cette signature fera usage de l'entrée `[ ca ]` et de ses sous-entrées dans `server_ssl.cnf`. Je ne vais pas ici détailler tous les paramètres, mais regardons tout de même les plus importants :

- Toutes les entrées sous `[ ca_policy ]` concernent le *Subject*. Tout est optionnel sauf le CN qui est fourni (*supplied*). On se simplifie la vie, mais rien ne vous empêche d'être plus précis...
- Une grande partie de la sous-entrée `[ ca_default ]` concerne l'emplacement des divers fichiers comme les clés, certificats, mais aussi le numéro de série et la « base de données » (en fait un fichier plat lisible) ainsi qu'un certain nombre de valeurs par défaut.

Les deux fichiers **index** (base de données) et **serial** (numéro de série) indiqués dans le fichier de configuration doivent exister au préalable. Pour **index**, un simple **touch** suffit :

Terminal
```
server # touch /etc/openvpn/keys/index
```

Pour le numéro de série, il est souvent recommandé de mettre **01** dedans, mais cela peut poser des problèmes si ce certificat doit aussi servir pour une connexion autre comme du https par exemple. Personnellement, je conseille d'y mettre une valeur pseudo aléatoire, ici sur quatre chiffres :

Terminal
```
server # echo `printf "%04x" $RANDOM` > /etc/openvpn/keys/serial
```

Nous pouvons enfin générer ce fameux certificat :

Terminal
```
server # yes | openssl ca \
                      -config /etc/openvpn/keys/server_ssl.cnf \
                      -out /etc/openvpn/keys/server.crt \
                      -infiles /etc/openvpn/keys/server.csr
Using configuration from /etc/openvpn/keys/server_ssl.cnf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
commonName            :ASN.1 12:'glmf.vpn'
Certificate is to be certified until Mar 23 11:03:54 2018 GMT (365 days)
Sign the certificate? [y/n]:

1 out of 1 certificate requests certified, commit? [y/n]Write out database with
1 new entries
Data Base Updated
```

L'usage de la commande **yes** nous évite simplement de devoir appuyer deux fois sur <y> pour confirmer les actions.

Et voilà, c'est fait ! Il ne reste plus qu'à créer le Diffie-Hellman, ce qui est simple :

Terminal
```
server # openssl dhparam 2048 -dsaparam -out /etc/openvpn/keys/dh2048.pem
```

L'option **-dsaparam** rend l'opération un petit peu moins sûre (cf. [2]), mais multiplie par **20** au minimum la vitesse de création, à vous de choisir. Pour donner un exemple, on passe d'une durée comprise entre une et cinq minutes sans cette option à une durée comprise entre cinq et quinze secondes pour la création d'un fichier **dhparam** en 2048 bits.

Nous sommes maintenant en mesure de lancer notre serveur OpenVPN :

Terminal
```
server # openvpn --config /etc/openvpn/server.conf
```

Et nous devons voir quelques lignes de log se terminant par « Initialization Sequence Completed ».

Afin d'être complets, nous allons dès maintenant commencer à gérer la CRL. Une CRL – ou *Certificate Revokation List* – sert à stocker la liste des certificats que nous avons révoqués. Nous verrons cette partie en détail à la fin de l'article, mais pour commencer il nous faut créer une liste vide.

Tout d'abord, nous aurons besoin du fichier `/etc/openvpn/keys/crlnumber` dans lequel nous mettrons la valeur de départ de `01` :

Terminal
```
echo 01 > /etc/openvpn/keys/crlnumber
```

Ce fichier sert à stocker le numéro de série de notre CRL, il est incrémenté à chaque nouvelle création. Pour générer notre CRL vide, la commande est la suivante :

Terminal
```
openssl ca \
            -config /etc/openvpn/keys/server_ssl.cnf \
            -gencrl \
            -keyfile /etc/openvpn/keys/ca.key \
            -cert /etc/openvpn/keys/ca.crt \
            -out /etc/openvpn/keys/crl.pem
Using configuration from /etc/openvpn/keys/server_ssl.cnf
```

Et voilà, la PKI serveur est enfin terminée.

1.4 Configuration du client

La configuration du client se fait comme pour le serveur, dans les grandes lignes. Nous allons voir les différences rapidement.

Tout d'abord, concernant le fichier de configuration OpenVPN, voici la version minimale :

Fichier
```
remote 172.25.10.1 1194
proto tcp
client
dev tun
resolv-retry 5
connect-retry-max 5
keepalive 10 120
nobind
comp-lzo
verb 3
ca /etc/openvpn/keys/ca.crt
cert /etc/openvpn/keys/client_1234.crt
key /etc/openvpn/keys/client_1234.key
```

La génération de la PKI client est à faire sur le serveur afin d'éviter d'une part de promener le fichier `ca.key` et d'autre part de garder le contrôle sur les certificats clients depuis le serveur.

Commençons par créer le répertoire `/etc/openvpn/clients` pour y stocker les clés et certificats de tous nos clients afin de pouvoir les gérer ultérieurement. Maintenant, nous pouvons créer la paire de clés pour le client :

Terminal
```
server # openssl ecparam -genkey -name secp384r1 -noout -out /etc/openvpn/
clients/client_1234.key
```

Pour la suite, nous allons devoir recopier le fichier `server_ssl.cnf` en `/etc/openvpn/clients/client_ssl.cnf` et le modifier un tout petit peu. Il suffit de changer la ligne `CN = glmf.vpn` par `CN = GLMF-1234` afin de répondre à notre exigence n° 8. Bien entendu, il faudra changer ce nombre pour créer un autre client.

Nous pouvons maintenant produire le certificat :

Terminal

```
server # openssl req -new -config /etc/openvpn/clients/client_ssl.cnf \
 -key /etc/openvpn/clients/client_1234.key \
 -out /etc/openvpn/clients/client_1234.csr
server # yes | openssl ca -config /tmp/client_ssl.cnf \
 -out /etc/openvpn/clients/client_1234.crt \
 -infiles /etc/openvpn/clients/client_1234.csr
Using configuration from /tmp/client_ssl.cnf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
commonName            :ASN.1 12:'GLMF-1234'
Certificate is to be certified until Mar 23 13:46:18 2018 GMT (365 days)
Sign the certificate? [y/n]:

1 out of 1 certificate requests certified, commit? [y/n]Write out database with
1 new entries
Data Base Updated
```

Recopions ensuite les fichiers `/etc/openvpn/clients/client_1234.key`, `/etc/openvpn/clients/client_1234.crt` et `/etc/openvpn/keys/ca.crt` vers le client dans le répertoire `/etc/openvpn/keys` avec un bon `scp` en renommant `client_1234.key` en `client.key` et `client_1234.crt` en `client.crt` afin d'uniformiser les fichiers de configuration client et de ne pas donner trop facilement d'indices à un pirate potentiel.

Une fois ceci fait, nous pouvons démarrer le client OpenVPN :

Terminal

```
client # openvpn --config /etc/openvpn/client.conf
```

Et là, la connexion est impossible, le serveur nous insulte avec un « No shared cipher » aussi péremptoire qu'intrigant. Que se passe-t-il ?

Il s'agit juste d'indiquer au serveur comme au client quels ciphers nous voulons pour le chiffrement TLS, car ceux que nous avons utilisés pour les certificats ne sont pas dans la liste par défaut. Cela se fait en rajoutant des deux côtés, à la fin du fichier de configuration, la ligne :

Fichier

```
tls-cipher ECDH-ECDSA-AES256-SHA:ECDH-RSA-AES256-SHA
```

Une fois cette ligne ajoutée, la connexion devrait s'établir et une nouvelle interface réseau nommée `tun0` apparaître sur le serveur et sur le client. Nous pouvons dès maintenant *pinger*, sur le réseau `10.50.0.0/24`, une machine depuis l'autre et réciproquement, le VPN est établi.

Pour éviter de devoir promener trois fichiers pour chaque client, il est possible de les exporter au format PCKS#12 qui va tout regrouper en un seul fichier :

Terminal

```
# cd /etc/openvpn/keys
# openssl pkcs12 -export -out client_1234.pfx -inkey client_1234.key -in
client_1234.crt -certfile ca.crt
```

OpenSSL nous demande la saisie d'une *passphrase* et là nous avons le choix, soit en mettre une, soit juste taper <**Return**> pour ne pas en mettre. Le choix vous appartient, mais il

faut savoir que si nous mettons une *passphrase* elle sera demandée à chaque lancement d'OpenVPN sur les clients, ce qui peut être perturbant voire très gênant dans le cas d'un client « enfoui » quelque part. D'autre part, la renseigner augmente encore plus la sécurité. C'est à vous de voir en fonction de vos besoins.

L'usage de certificats au format PKCS#12 nécessite une modification du fichier de configuration client. Il faut supprimer les trois directives suivantes :

Fichier
```
ca /etc/openvpn/keys/ca.crt
cert /etc/openvpn/keys/client.crt
key /etc/openvpn/keys/client.key
```

Pour les remplacer par une seule :

Fichier
```
pkcs12 /etc/openvpn/keys/client_1234.pfx
```

Il est à noter que cette modification fonctionne aussi côté serveur.

Une autre solution pour éviter la multiplicité des fichiers consiste à inclure directement les clés et certificats dans le fichier de configuration. Ceci se fait grâce à une syntaxe un peu exotique, il suffit de recopier le contenu des `.crt` et du `.key` directement entre deux balises `<ca>` et `</ca>` pour `ca.crt`, `<cert>` et `</cert>` pour `client.crt`, `<key>` et `</key>` pour `client.key`. À chaque fois, on ne prend que la partie entre `-----BEGIN...` et `-----END...` Attention au `.crt` qui comporte d'autres informations à ne pas recopier. Cela donne quelque chose comme ça :

Fichier
```
...début du fichier de configuration...
<ca>
-----BEGIN CERTIFICATE-----
MIIEmzCCA4OgAwIBAgIJAMQYXLennSJDMA0GCSqGSIb3DQEBCwUAMIGPMQswCQYD
...
dFwYQcHmnArX5gEi06wWfH+j2TqUHTDWZm340MBu5g==
-----END CERTIFICATE-----
</ca>

<cert>
-----BEGIN CERTIFICATE-----
MIIE9TCCA92gAwIBAgIBAjANBgkqhkiG9w0BAQsFADCBjzELMAkGA1UEBhMCRlIx
...
4/fjP6mPUu1apM/4Msu/XMtZz1nTqxqRQg==
-----END CERTIFICATE-----
</cert>

<key>
-----BEGIN PRIVATE KEY-----
MIIEvwIBADANBgkqhkiG9w0BAQEFAASCBKkwggSlAgEAAoIBAQDGKUXrOrvNANae
...
64s/iUWmlOomIXwzZwdtZKcfVQ==
-----END PRIVATE KEY-----
</key>
...suite du fichier de configuration...2. Améliorations
```

2.1 Augmenter la sécurité des messages

Notre exigence n°5 nous demande de chiffrer les messages en AES-256-CBC et d'utiliser SHA256 pour le hash de contrôle.

Si l'on regarde les logs de notre serveur, nous y trouvons des lignes comportant la mention suivante :

Fichier
```
Data Channel Decrypt: Using 160 bit message hash 'SHA1' for HMAC
authentication
```

Or nous voulons du SHA256, car le SHA1 n'est plus sûr (exigence n°5). Pour ce faire, il suffit de rajouter la ligne suivante des deux côtés :

Fichier
```
auth SHA256
```

Et nous avons la confirmation que nous attendions :

Fichier
```
Data Channel Decrypt: Using 256 bit message hash 'SHA256' for HMAC authentication
```

De la même façon, les logs d'OpenVPN nous indiquent :

Fichier
```
Data Channel Encrypt: Cipher 'BF-CBC' initialized with 128 bit key
```

... ce qui ne nous convient pas. Ajoutons donc une ligne dans les deux fichiers de configuration :

Fichier
```
cipher AES-256-CBC
```

Une fois OpenVPN relancé, nous obtenons :

Fichier
```
Data Channel Encrypt: Cipher 'AES-256-CBC' initialized with 256 bit key
```

Et hop, nous avons augmenté sérieusement notre niveau de chiffrement et de sécurité.

Il est aussi possible de rajouter la ligne suivante sur le serveur afin de ne pas autoriser des versions inférieures de ce protocole :

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

2.2 Les topologies réseau

Selon la documentation OpenVPN, il existe trois topologies possibles :

⇨ `net30` qui alloue un sous-réseau en `/30` (mais avec un *netmask* en `/32`...) pour chaque client. Il est recommandé si les postes clients fonctionnent sous **Windows**. Il s'agit du mode par défaut en OpenVPN 2.0+. Avec ce mode, on obtient une interface configurée comme sur cet exemple :

Fichier
```
tun0: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>  mtu 1500
        inet 10.50.0.6  netmask 255.255.255.255  destination 10.50.0.5
```

⇨ **p2p** qui est déprécié et qui fournit une configuration réseau en **/30** avec juste un lien entre le client et le serveur. Vu par **ifconfig**, cela donne :

Fichier
```
tun0: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>  mtu 1500
        inet 10.50.0.4  netmask 255.255.255.255  destination 10.50.0.1
```

⇨ **subnet** qui est en fait le successeur de **p2p**. Cette topologie fonctionne aussi avec Windows, mais exige un OpenVPN 2.1 minimum ainsi qu'une version 8.2 ou supérieure du driver **TAP-Win32** sous Windows. Elle fournit un sous-réseau en **/24** ce qui est beaucoup plus souple que le **p2p**. La commande **ifconfig** nous retourne ce genre de résultat :

Fichier
```
tun0: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>  mtu 1500
        inet 10.50.0.4  netmask 255.255.255.0  destination 10.50.0.4
```

Pour changer de topologie, il suffit d'ajouter la directive **topology** dans les fichiers de configuration. Pour nous mettre en topologie **subnet** (exigence n° 10) il suffit de mettre côté serveur et client(s) :

Fichier
```
topology subnet
```

2.3 L'interface de management

Il s'agit là d'une fonctionnalité assez peu connue d'OpenVPN, mais qui va nous permettre, si on lui demande gentiment, de disposer d'une interface de management autorisant la prise de contrôle à distance de notre serveur via **telnet** afin de lui faire exécuter plusieurs actions de contrôle.

Cette interface s'active en ajoutant la directive **management** suivie de l'adresse IP et du port sur lesquels elle doit écouter. Par exemple :

Fichier
```
management 172.25.10.1 21111
```

À la suite de quoi, un **telnet 172.25.10.1 21111** nous retournera la bannière de bienvenue :

Terminal
```
>INFO:OpenVPN Management Interface Version 1 -- type 'help' for more info
```

NOTE

Attention, l'usage de **telnet** implique que le trafic de management ne sera nullement chiffré. Aussi il est plus que conseillé de le faire écouter sur le **loopback** en **127.0.0.1** ou directement sur l'interface tunnel. Dans ce dernier cas, nous profiterons du chiffrement apporté par le VPN, mais certaines commandes ne seront pas disponibles, principalement celles servant à établir le tunnel. Il est aussi fort simple d'écrire un petit daemon qui écoutera en SSL sur un port et transmettra nos commandes à l'interface de management en local.

De nombreuses commandes sont à notre disposition, soit en direct, soit via un script et nous allons juste en examiner quelques-unes à titre d'exemple.

La première est la commande **status** qui permet d'avoir la liste des clients connectés avec leurs caractéristiques. Par exemple :

Terminal
```
status
OpenVPN CLIENT LIST
Updated,Thu Mar 23 17:20:54 2017
Common Name,Real Address,Bytes Received,Bytes Sent,Connected Since
GLMF-1234,172.25.10.10:52503,3047,2708,Thu Mar 23 17:18:52 2017
ROUTING TABLE
Virtual Address,Common Name,Real Address,Last Ref
10.50.0.4,GLMF-1234,172.25.10.10:52503,Thu Mar 23 17:18:53 2017
GLOBAL STATS
Max bcast/mcast queue length,0
END
```

Ceci nous montre que notre client **GLMF-1234** est bien connecté. Si nous voulons le déconnecter, nous pouvons le faire grâce à la commande **kill** :

Terminal
```
kill GLMF-1234
SUCCESS: common name 'GLMF-1234' found, 1 client(s) killed
```

Nous avons là un premier exemple de l'intérêt de mettre un *distinguished name* dans les certificats ; l'identifiant présenté ici (et ailleurs, nous le verrons plus bas) en est extrait directement.

Une autre option intéressante, liée à l'usage de cette console, est la directive **management-hold** qui démarrera le serveur (ou le client) en mode attente, donc sans autoriser la moindre connexion, jusqu'à ce que l'on se connecte sur la console et que l'on passe la commande :

Terminal
```
hold release
```

Cela donne la séquence suivante si l'on essaye sur le serveur :

Terminal
```
server # openvpn --config server.conf
OpenVPN 2.4.0 i586-pc-linux-gnu [SSL (OpenSSL)] [LZO] [EPOLL] [PKCS11] [MH]
[IPv6] built on Nov 19 2015
...
Need hold release from management interface, waiting...
```

Depuis une autre machine :

Terminal
```
$ telnet 172.25.10.1 21111
>INFO:OpenVPN Management Interface Version 1 -- type 'help' for more info
>HOLD:Waiting for hold release
hold release
SUCCESS: hold release succeeded
```

Sur le serveur :

Terminal
```
MANAGEMENT: Client connected from [AF_INET]172.25.10.12:21111
MANAGEMENT: CMD 'hold release'
```

Et le démarrage continue normalement. Cette directive fonctionne aussi bien sur un serveur OpenVPN que sur un client.

De nombreuses autres directives existent, je vous laisse les consulter sur la page officielle **[4]**.

2.4 Les hooks

2.4.1 L'authentification triple : certificat, login, mot de passe

Outre la partie échange de clés et vérification du certificat, il est possible de demander au client OpenVPN d'envoyer un compte et un mot de passe au serveur. Il existe un *hook* sur celui-ci permettant à un script de récupérer les données envoyées et de les comparer avec celles attendues.

Ajoutons à notre client **GLMF-1234** un compte nommé `user1234` et un mot de passe `4321Password`.

NOTE

Dans nos exemples simplifiés, nous mettrons le mot de passe en clair tant sur le client que sur le serveur. Cependant, comme nous maîtrisons toute la chaîne, il serait bon de chiffrer ce mot de passe dans une version définitive.

Attention cependant au fait que, même chiffré, si une personne mal intentionnée récupère le fichier de mots de passe, il pourra se faire passer pour le client s'il possède aussi le bon certificat. Pour plus de sécurité, il est possible d'exiger qu'OpenVPN demande le mot de passe et une éventuelle *passphrase* pour la clé via l'interface de management et les directives suivantes à insérer dans la configuration client :

Fichier

```
management-hold
management-query-passwords
auth-retry interact
```

Pour plus de détails, vous pouvez consulter la documentation complète de l'interface de management en [4].

Ceci se fait de la manière suivante :

⇨ Créer un fichier nommé, par exemple, `client.pwd` que l'on mettra dans `/etc/openvpn` :

Terminal

```
client # echo user1234 > /etc/openvpn/client.pwd
client # echo  4321Password >> /etc/openvpn/client.pwd
```

⇨ Rajouter dans le fichier de configuration du client la directive :

Fichier

```
auth-user-pass /etc/openvpn/client.pwd
```

Rajoutons ensuite dans le fichier de configuration côté serveur la directive :

Fichier

```
auth-user-pass-verify /etc/openvpn/passverify.pl via-env
```

Cette directive demande à OpenVPN d'appeler le script **/etc/openvpn/passverify.pl** en passant les compte et mot de passe via des variables d'environnement (**via-env**). Il est possible aussi de lui demander de les passer par fichier en utilisant **via-file** en lieu et place de **via-env**.

Une version assez minimaliste du script **passverify.pl** pourrait être la suivante :

Fichier

```
#!/usr/bin/perl -w
use strict;

my $login;
my $pass;
my $serial;

$login = '';
$pass = '';
$login = $ENV{'username'} if(exists($ENV{'username'}));
$pass = $ENV{'password'} if(exists($ENV{'password'}));

# On récupère le sujet dans le certificat
$serial = $ENV{'X509_0_CN'};
exit(1) if($serial ne "GLMF-1234");
exit(2) if($login ne "user1234");
exit(3) if($pass ne "4321Password");
exit(0);
```

Si le script retourne **0**, OpenVPN considère que tout va bien et continue la procédure, sinon il sort en erreur et rejette la connexion.

Afin d'autoriser OpenVPN à exécuter un script externe et à lui passer le mot de passe, il faut au serveur rajouter la directive :

Fichier

```
script-security <niveau>
```

Les quatre niveaux possibles sont :

⇨ **0** : interdiction d'appel de tout programme externe ;

⇨ **1** : (valeur par défaut) seuls les exécutables « built-in » comme **ifconfig**, **ip** ou **route** sont autorisés ;

⇨ **2** : autorise l'appel de scripts utilisateur en plus ;

⇨ **3** : comme le **2**, mais autorise en plus le passage des mots de passe via les variables d'environnement.

C'est donc le niveau **3** qui nous intéresse.

À l'heure actuelle, nos deux fichiers de configuration doivent ressembler à ça :

⇨ Serveur :

Fichier

```
topology subnet
port 21110
proto tcp
dev tun
ca keys/ca.crt
cert keys/server.crt
key keys/server.key  # This file should be kept secret
```

```
dh keys/dh2048.pem
server 10.50.0.0 255.255.255.0
ifconfig-pool-persist ipp.txt
keepalive 10 120
comp-lzo
persist-key
persist-tun
status openvpn-status.log
verb 3
tls-version-min 1.2
tls-server
auth sha256
cipher AES-256-CBC
tls-cipher ECDH-ECDSA-AES256-SHA:ECDH-RSA-AES256-SHA
management 192.168.1.99 21111
script-security 3
auth-user-pass-verify /etc/openvpn/passverify.pl via-env
```

⇨ Client :

Fichier

```
topology subnet
remote 172.25.10.1 21110
proto tcp
client
dev tun
resolv-retry 5
connect-retry-max 5
keepalive 10 120
nobind
comp-lzo
verb 3
ca /etc/openvpn/keys/ca.crt
cert /etc/openvpn/keys/client.crt
key /etc/openvpn/keys/client.key
auth SHA256
cipher AES-256-CBC
tls-cipher ECDH-ECDSA-AES256-SHA:ECDH-RSA-AES256-SHA
auth-user-pass /etc/openvpn/client.pwd
```

Donc si nous relançons le serveur et le client, nous devons avoir une connexion qui s'établit sans problème et trouver dans les logs du serveur une ligne qui ressemble à :

Fichier

```
TLS: Username/Password authentication succeeded for username 'user1234'
```

Et si on remplace le mot de passe par `toto` nous avons :

Fichier

```
WARNING: Failed running command (--auth-user-pass-verify): external
program exited with error status: 3
TLS Auth Error: Auth Username/Password verification failed for peer
```

Le code de sortie `3` indique bien que le problème vient du mot de passe (`exit(3) if($pass ne "4321Password");`).

Si l'on désire ne pas passer au niveau 3 de sécurité pour les scripts, il est possible d'envoyer les données via un fichier dont le nom est passé en paramètre au script lors de son appel. Par la suite, OpenVPN supprimera ledit fichier juste après l'exécution du *hook*.

Cependant, il est quand même conseillé d'indiquer un point de montage en mémoire, comme `/run` ou `/var/run` en fonction de votre distribution par exemple, via la directive `tmp-dir`. Pour ce faire, il faut rajouter à la configuration serveur la ligne :

Fichier
```
tmp-dir /run
```

Et remplacer la ligne :

Fichier
```
auth-user-pass-verify /etc/openvpn/passverify.pl via-env
```

par

Fichier
```
auth-user-pass-verify /etc/openvpn/passverify.pl via-file
```

On peut maintenant passer au niveau 2 :

Fichier
```
script-security 2
```

Et le script `passverify.pl` devient :

Fichier
```
#!/usr/bin/perl -w
use strict;

my $login;
my $pass;
my $serial;
my $fh;

$login = '';
$pass = '';
my $file = $ARGV[0];
open($fh, "<", $file) or exit(4);
$login = <$fh>;
chomp($login);
$pass = <$fh>;
chomp($pass);
close($fh);

# On récupère le sujet dans le certificat
$serial = $ENV{'X509_0_CN'};
exit(1) if($serial ne "GLMF-1234");
exit(2) if($login ne "user1234");
exit(3) if($pass ne "4321Password");
exit(0);
```

Bien évidemment, un vrai script ira chercher les informations dans une base de données plutôt que de les avoir codées en dur, mais c'est pour la démonstration.

Pour ceux qui le désireraient, il est possible de demander à OpenVPN de passer le *login* en lieu et place du *common name* grâce à la directive : `username-as-common-name` mise côté serveur.

En cas d'échec d'authentification le comportement du client OpenVPN peut être modifié par la directive `auth-retry <type>` où `type` peut prendre les valeurs suivantes :

- **none** : le client va s'arrêter, c'est le comportement par défaut ;
- **nointeract** : le client va réessayer en boucle de se connecter sans rien demander. C'est l'option à choisir pour un client enfoui auquel on n'a pas vraiment accès ;
- **interact** : le client va demander de saisir le *login* et le mot de passe avant de tenter une reconnexion.

2.4.2 Vérification du certificat client

Il est possible de renforcer encore la sécurisation du système en faisant passer toute une batterie de tests aux certificats clients via un script externe.

Celui-ci se déclenche en ajoutant à la configuration du serveur les lignes :

Fichier

```
tls-export-cert /run/exportcert
tls-verify /etc/openvpn/certcheck.pl
```

La première directive décrit un répertoire dans lequel OpenVPN pourra extraire toutes les données des certificats. Il est donc important de le créer sur un point de montage en mémoire pour que les données ne soient jamais accessibles même dans le cas d'une coupure de courant juste au mauvais moment.

Comme d'habitude, le script **certcheck.pl** doit retourner **0** si tout va bien, autre chose en cas d'erreur et le code de retour est visible dans les logs serveur.

Le script que je vais vous proposer ici est une version adaptée de l'exemple que l'on peut trouver dans les exemples fournis avec le paquet Debian : **/usr/share/doc/openvpn/examples/sample-scripts/verify-cn**. OpenVPN passe deux arguments à ce script :

1. La profondeur actuelle dans la chaîne de certificats. Dans le cas typique d'une chaîne à deux niveaux le certificat racine est au niveau **1** et celui du client au niveau **0**. Le script sera appelé pour chaque niveau, ce qui permet d'effectuer des vérifications sur toute la chaîne de certificats.
2. Le sujet (pour nous **/CN=...**) extrait du certificat fourni.

Un script **certcheck.pl** assez minimal, mais suffisant pourrait être :

Fichier

```
#!/usr/bin/perl -w
use strict;

die "usage: checkcert.pl certificate_depth subject" if (@ARGV != 2);

my ($depth, $subject) = @ARGV;

my $command;
my $ret;
my $login;

if ($depth == 0)
{
 # On peut récupérer le login depuis l'environnement. On ne s'en sert
 # pas dans cet exemple.
 $login = $ENV{'X509_0_CN'};
 $login =~ s/^product\.//;
```

```
  # Vérifier si le certificat a bien été signé par le CA dont on
  # dispose.
  $command = "/usr/bin/openssl verify -CAfile /etc/openvpn/keys/
ca.crt ".$ENV{'peer_cert'};
  $ret = '$command';
  exit(1) if($? != 0);

  # Vérifier que le certificat client n'a pas encore expiré ou ne va
  # pas le faire dans l'heure prochaine (1h = 3600 sec).
  $command = "openssl x509 -checkend 3600 -noout -in ".$ENV{'peer_
cert'};
  $ret = '$command';
  exit(2) if($? != 0);

  # Même chose pour le ca.crt, on ne sait jamais...
  $command = "openssl x509 -checkend 3600 -noout -in /etc/openvpn/
keys/ca.crt";
  $ret = '$command';
  exit(3) if($? != 0);

  # Vérifier la syntaxe du sujet (GLMF-<digits>).
  exit(4) if ($subject !~ /^CN=GLMF-\d+$/);
}

exit 0;
```

Il est plus que conseillé de jouer avec ce script et de modifier les certificats clients afin de bien voir l'impact des changements apportés. L'usage du module **Data::Dumper** et d'un **print Dumper(%ENV)** dans un fichier est très utile pour voir ce qui se passe.

2.4.3 Récupération en temps réel du statut des clients

Nous avons vu qu'il est possible de consulter le statut des clients via la console de management et nous savons que l'on peut aussi demander au serveur de tenir à jour un fichier reflétant ce statut avec la directive **status <fichier>** dans laquelle **fichier** sera du genre **/var/log/openvpn.status** par exemple. Cependant, ce fichier est mis à jour avec des délais variables et parfois trop longs pour nos besoins. Il existe deux *hooks* pour remédier à cet état de fait :

Fichier

```
client-connect <script>
client-disconnect <script>
```

Ici **script** pourra être par exemple **/etc/openvpn/connection.pl**.

Les données sont passées par des variables d'environnement et nous retrouvons celles qui nous intéressent dans les variables d'environnement **username** pour le *login*, **ifconfig_pool_remote_ip** pour l'adresse IP et **script_type** pour savoir si le client est en train de se connecter (**client-connect**) ou de se déconnecter (**client-disconnect**).

Dans le cas présent, il est fortement conseillé de travailler avec une base de données dans laquelle nous avons déclaré une table qui regroupe les caractéristiques des clients. Par exemple :

Fichier

```
CREATE TABLE client(id integer primary key, valid integer default 0,
login text, ipaddress text, password text, connected integer default 0);
```

Nous avons donc `id` la clé primaire, `valid` qui permettrait de rajouter un test dessus dans `passverify.pl` et de rejeter des connexions qu'on a décidé de bloquer juste en mettant ce *flag* à `0`, `login` qui est le *login* des clients, `ipaddress` l'adresse IP associée, `password` qui se passe de commentaire et `connected` que nous allons garder à jour dans le script ci-dessous. Nous avons là la base d'une petite supervision de nos VPNs avec la possibilité d'agir dessus comme bon nous semble.

Le script `connection.pl` minimal est plutôt simple :

Fichier

```
#!/usr/bin/perl -w

use strict;
use DBI;

my $login;
my $state;
my $dbh;
my $sth;
my $ip;
my $DATABASE = "/usr/share/mydb";

$login = $ENV{'username'};
$state = $ENV{'script_type'}; # client-connect or client-disconnect
$ip = $ENV{'ifconfig_pool_remote_ip'};

$dbh = DBI->connect("dbi:SQLite:dbname=$DATABASE", "", "", { RaiseError=>0
}) or exit(1);
if($state eq "client-connect")
{
 $sth = $dbh->prepare("update client set connected=1, ipaddress=$ip where
plnum=$login");
}
else
{
 $sth = $dbh->prepare("update client set connected=0 where plnum=$login");
}

$sth->execute();
$sth->finish();
$dbh->disconnect();

exit(0);
```

Bien entendu, il faut rajouter les tests d'erreur et plein d'autres choses que je n'ai pas mis pour rester concentré sur l'essentiel.

2.4.4 Ajout, modification ou suppression d'une adresse IP dans le réseau VPN

Parfois, il peut être très intéressant de savoir quand une adresse IP apparaît, disparaît ou est modifiée. Par exemple, si certains de nos clients servent de routeur pour atteindre des réseaux distants, s'il faut modifier des règles de firewall, etc.

Pour autoriser ce genre d'action, OpenVPN propose la directive **`learn-address <script>`** à mettre côté serveur. Le script se verra fournir trois paramètres, dans l'ordre :

- l'opération qui pourra être **`add`**, **`update`**, ou **`delete`** ;
- l'adresse concernée qui peut être une adresse IP simple, un sous-réseau ou même une mac-address quand on utilise les *devices* tap ;
- le *common name* du client concerné, extrait de son certificat. Ce paramètre n'est pas fourni pour une opération de type **`delete`**.

Ce cas de figure est tellement spécifique que je ne vais pas vous proposer de script en exemple, mais il est fort simple à écrire. Ce script n'a pas besoin de retourner de valeur particulière, car les codes de retour éventuels ne sont pas pris en compte par OpenVPN.

2.5 Configuration spécifique client par client

Lorsqu'un client se connecte, la partie serveur d'OpenVPN se comporte comme un serveur DHCP vis-à-vis de ce premier. Il lui fournit donc une adresse IP prise dans une plage spécifiée dans sa configuration, mais rien ne nous garantit qu'un client retrouvera la même adresse d'une connexion sur l'autre. Nous avons vu dans les deux derniers *hooks* étudiés qu'il existe plus d'une manière d'associer l'adresse IP d'un client à son *common name*, mais on peut désirer fournir à tout ou partie de notre *pool* de clients une adresse IP fixe. On peut aussi désirer préciser certains autres paramètres en les adaptant finement client par client.

Ceci se fait en ajoutant la directive **`client-config-dir <répertoire>`** à la configuration de notre serveur. Dans ce répertoire, habituellement nommé **`/etc/openvpn/ccd`**, nous devons mettre des fichiers nommés d'après le *common name* de nos clients. Par exemple, pour notre client **`GLMF-1234`**, il faudra créer un fichier nommé **`/etc/openvpn/ccd/GLMF-1234`** dans lequel on écrira la partie spécifique de la configuration de ce client. Il s'agit le plus souvent de fixer l'adresse IP, ce qui se fait via la directive **`ifconfig-push`** suivie de l'adresse IP voulue.

Lorsque nous avons plusieurs clients, il peut être utile d'autoriser la communication d'un client à l'autre. Cette fonctionnalité n'est pas activée par défaut, mais il suffit de rajouter l'option **`client-to-client`** dans la configuration du serveur pour la mettre en place.

2.6 Révocation de certificats

Maintenant que nous savons comment faire se connecter des clients, que devons-nous faire quand nous désirons en bannir un définitivement ? Nous avons sous la main les clés et certificats de tous nos clients sur le serveur, nous pouvons donc révoquer les certificats quand nous le désirons grâce à la commande :

Terminal

```
server # openssl ca -revoke /etc/openvpn/keys/client_1234.crt -config server_
ssl.cnf
```

Une fois ceci fait, si nous regardons le fichier **`/etc/openvpn/keys/index`**, nous obtenons ceci :

Terminal

```
server # cat index
V180323110345Z   4437 unknown /CN=glmf.vpn
V180323110354Z   4438 unknown /CN=glmf.vpn
R180323134618Z 170324155801Z 4439 unknown /CN=GLMF-1234
```

Nous remarquons que la première colonne pour le client `GLMF-1234` est passée de `V` à `R` qui signifie *Révoqué*. Les autres colonnes représentent respectivement les dates d'expiration et de révocation, l'ID du certificat, son nom (si renseigné, `unknown` sinon), le *distinguished name*.

Une fois le certificat révoqué, il faut créer une nouvelle CRL afin de mettre OpenVPN au courant :

Terminal

```
server # openssl ca -gencrl -out /etc/openvpn/keys/crl.pem -config /etc/openvpn/
keys/server_ssl.cnf
Using configuration from /etc/openvpn/keys/server_ssl.cnf
```

Qu'a donc été modifié dans ce fichier que nous venons de remplacer ? Avant la révocation, le fichier `crl.pem` nous présentait les informations suivantes :

Terminal

```
server # openssl crl -in crl.pem -text -noout
Certificate Revocation List (CRL):
        Version 2 (0x1)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: /CN=glmf.vpn
        Last Update: Mar 24 15:50:51 2017 GMT
        Next Update: Mar 23 15:50:51 2022 GMT
        CRL extensions:
            X509v3 CRL Number:
                1
No Revoked Certificates.
    Signature Algorithm: ecdsa-with-SHA256
         30:65:02:31:00:a3:fd:49:76:81:4b:8f:95:4e:d2:1d:40:4e:
         11:19:45:b5:96:e1:48:e8:f9:cf:bb:14:05:e6:81:15:72:a3:
         db:0d:7a:ec:f3:85:e3:88:b6:70:8d:83:91:d7:6e:31:68:02:
         30:12:cb:25:e5:11:d0:c5:91:f9:23:6e:92:d2:60:13:61:fe:
         68:e5:05:d2:b7:55:a1:e9:11:15:4b:76:6a:b6:01:73:ad:b2:
         f9:0f:21:8c:75:1d:5a:cc:8f:2f:5c:6b:7c
```

Et après la révocation :

Terminal

```
server # openssl crl -in crl.pem -text -noout
Certificate Revocation List (CRL):
        Version 2 (0x1)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: /CN=glmf.vpn
        Last Update: Mar 24 15:58:48 2017 GMT
        Next Update: Mar 23 15:58:48 2022 GMT
        CRL extensions:
            X509v3 CRL Number:
                2
Revoked Certificates:
    Serial Number: 4439
        Revocation Date: Mar 24 15:58:01 2017 GMT
```

```
    Signature Algorithm: ecdsa-with-SHA256
         30:65:02:31:00:b7:47:2c:ea:43:ad:20:74:c9:53:45:32:f5:
         7d:be:e2:7c:b3:be:c8:ad:c2:a3:71:57:7c:75:d4:ad:5d:30:
         df:d1:3d:f0:97:61:f5:1f:6b:a1:fe:b3:71:b2:27:d3:34:02:
         30:47:d0:75:1a:1d:c7:dd:b1:f2:c5:85:e2:e9:0e:e2:ac:16:
         70:62:05:76:6d:52:1b:f9:9b:10:49:c9:54:32:2b:78:aa:fc:
         28:7b:8e:98:74:f3:c9:c0:8e:04:88:c7:e9
```

Faisons un `diff` entre ces deux fichiers pour y voir plus clair et, si l'on omet la partie signature, cela nous donne :

Fichier

```
5,6c5,6
<         Last Update: Mar 24 15:50:51 2017 GMT
<         Next Update: Mar 23 15:50:51 2022 GMT
---
>         Last Update: Mar 24 15:58:48 2017 GMT
>         Next Update: Mar 23 15:58:48 2022 GMT
9,10c9,12
<                 1
< No Revoked Certificates.
---
>                 2
> Revoked Certificates:
>     Serial Number: 4439
>         Revocation Date: Mar 24 15:58:01 2017 GMT
```

On remarque plusieurs choses :

- le CRL Number a été incrémenté, on est passé de `1` à `2` ;
- la signature SHA256 a été modifiée (encore heureux vu que le fichier a changé) ;
- nous passons de la ligne `No Revoked Certificates` à `Revoked Certificates:` comprenant l'ID du certificat révoqué (`4439`) ainsi que la date de révocation. Nous pouvons vérifier que l'ID est bien le bon :

Terminal

```
server # openssl x509 -text -in /etc/openvpn/clients/client_1234.crt -noout |
grep "Serial Number"
        Serial Number: 17465 (0x4439)
```

- la date après `Next Update:` a légèrement augmenté pour que la différence entre `Next Update` et `Last Update` reste la même. Mais d'où vient ce delta de cinq ans ? Tout simplement du fichier de configuration `ssl_server.cnf` que nous avons écrit, plus précisément de la directive `default_crl_days = 1825` et 1825 jours = 5 ans, CQFD. Plus sérieusement, cette option indique à OpenSSL que si l'on ne fournit pas une nouvelle CRL – qui peut parfaitement ne pas être modifiée, seule la date de création compte – avant le 23/03/2022, tous les certificats seront considérés comme révoqués et plus aucune connexion ne pourra s'effectuer ! Voilà pourquoi il est de bon ton de créer une CRL vide au démarrage.

Faisons maintenant en sorte qu'OpenVPN prenne en charge cette liste de révocation. Il suffit de rajouter dans la configuration serveur la ligne suivante et de relancer OpenVPN :

Fichier

```
crl-verify keys/crl.pem
```

Une fois ceci fait, toute tentative de connexion du client `GLMF_1234` se soldera par un échec sans explication particulière côté client, mais avec une ligne très explicite dans les logs serveur :

Fichier

```
CRL CHECK FAILED: CN=GLMF-1234 is REVOKED
```

En commentant la ligne que nous venons de rajouter sur le serveur, le client pourra se reconnecter comme si de rien n'était.

Il est à noter qu'un redémarrage d'OpenVPN n'est pas nécessaire après une mise à jour de la CRL, la prise en compte est automatique.

Maintenant, pouvons-nous faire quelque chose si nous voulons réactiver un client sans lui recréer de certificat – dans le cas d'une machine inaccessible par exemple ? La réponse est positive, mais pas très académique. Nous avons vu que le fichier de « base de données » d'OpenSSL est un simple fichier plat parfaitement lisible et donc éditable. Ouvrons-le donc, remplaçons le `R` de *Revoked* par le `V` de *Valid*, supprimons la date de révocation en prenant garde à bien conserver les tabulations (`:set list` sous Vim est notre ami) puis enregistrons-le et créons une nouvelle CRL comme expliqué plus haut. Relançons le client et miracle, il se connecte sans souci. Inutile de dire qu'il s'agit là d'un pis aller, car si nous avons dû révoquer un certificat avant son expiration c'est qu'il était ou risquait d'être compromis. Cette solution n'est donc à utiliser qu'avec une extrême prudence, mais elle fonctionne.

2.7 Passage en IPv6

Tout le monde sait maintenant qu'IPv4 étant plus que moribond il va bien falloir passer à IPv6 et cela OpenVPN le fait très bien. Dans les faits, ce dernier exige pour le moment d'être en *dual stack* dans le tunnel et donc de garder nos adresses IPv4 en plus de la nouvelle plage en IPv6, mais ce n'est pas un problème, car nous utilisons en règle générale des plages d'adresses privées.

Partons du principe qu'IPv6 est activé et fonctionnel en version de base sur nos machines. Pour ce test, nous allons simplement utiliser les adresses de lien local en `fe80` qui sont configurées automatiquement. Notre serveur a pour adresse `fe80::a00:27ff:fe7e:3dd9/64`.

Nous avons juste à effectuer deux modifications côté serveur :

- remplacer `proto tcp` par `proto udp6` (`tcp6` existe, mais ne semble pas fonctionnel pour le moment) ;
- ajouter la ligne `server-ipv6 2001:db8:0:123::/64` qui fournira les adresses IPv6 dans le tunnel.

Et c'est tout !

Pour la partie client, il suffit de remplacer deux lignes aussi :

- `remote 192.168.1.99 21110` par `remote fe80::a00:27ff:fe7e:3dd9 21110` (adresse et port du serveur) ;
- `proto tcp` par `proto udp6`.

Une fois le serveur et le client redémarrés, nous pouvons regarder l'interface `tun0` de notre serveur :

Terminal

```
server # ifconfig tun0
tun0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-
00-00
          inet addr:10.50.0.1  P-t-P:10.50.0.1  Mask:255.255.255.0
          inet6 addr: 2001:db8:0:123::1/64 Scope:Global
```

Et nous observons une belle adresse IPv6 (la première du *pool*) se rajouter à l'adresse IPv4 habituelle.

Notre client pour sa part a bien récupéré une adresse du *pool* v6 :

Terminal

```
client # ifconfig tun0
tun0: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>  mtu 1500
        inet 10.50.0.4  netmask 255.255.255.0  destination 10.50.0.4
        inet6 2001:db8:0:123::1002  prefixlen 64  scopeid 0x0<global>
```

Et nous pouvons *pinger* notre serveur sans problème à travers le tunnel :

Terminal

```
client # ping6 2001:db8:0:123::1
PING 2001:db8:0:123::1(2001:db8:0:123::1) 56 data bytes
64 bytes from 2001:db8:0:123::1: icmp_seq=1 ttl=64 time=1.13 ms
```

Et voilà, nous sommes passés en IPv6 non seulement dans le tunnel, mais aussi en dehors. La *stack* IPv6 d'OpenVPN est encore en cours de développement et il reste des choses à faire, mais ça fonctionne et c'est franchement très simple à mettre en œuvre.

2.8 Cohabitation avec un serveur HTTPS

Parfois pour de sombres raisons *firewallesques* on peut vouloir faire écouter le serveur sur le port `443`. OpenVPN est l'un des rares logiciels de *tunneling* à supporter cette option et c'est fort bien. Dans ce cas, que faire si notre serveur de tunnel sert aussi de serveur web ? Je sais, mauvaise infrastructure réseau, changer d'infrastructure. Cependant, ce n'est pas toujours possible et il faut trouver une solution. Manifestement, nous ne sommes pas les seuls à y penser, car une directive existe pour ce cas de figure, elle s'appelle `port-share <adresse> <port>`. Lorsqu'elle est utilisée, OpenVPN va router tout le trafic chiffré qu'il ne comprend pas vers l'adresse et le port spécifiés. Souvent l'adresse est `127.0.0.1`, mais il est complètement possible de renvoyer le flux vers une autre machine. Attention juste au routage dans ce cas-là. Pour rester simple, nous allons installer un **lighttpd** sur notre serveur VPN, le configurer pour écouter uniquement en https sur le port `4443` en `localhost` :

Terminal

```
# apt install lighttpd
```

On crée vite fait un certificat pour https :

Terminal

```
# openssl req -newkey rsa:2048 -new -nodes -x509 -days 3650 -keyout key.pem
-out cert.pem
# cat key.pem >> cert.pem
```

On modifie `/etc/lighttpd/lighttpd.conf` :

Fichier

```
server.bind                = "127.0.0.1"
server.port                = 4443
ssl.engine                 = "enable"
ssl.pemfile                = "/root/cert.pem"
```

Et on rajoute la directive qui va bien dans `/etc/openvpn/server.conf` :

Fichier

```
port-share 127.0.0.1 4443
```

Attention à bien repasser le serveur et les clients en full IPv4 si lighttpd n'est pas en IPv6 sinon gare aux conflits...

On relance lighttpd et OpenVPN, on vérifie que le client continue de se connecter et on valide avec notre navigateur favori que le site web est accessible.

CONCLUSION

Comme nous avons pu le voir, OpenVPN est un logiciel plein de ressources peu connues permettant une maîtrise complète de nos VPNs pour peu que l'on s'astreigne à lire la documentation et à se plonger aussi dans les arcanes d'OpenSSL. Une fois cet apprentissage effectué, nous nous trouvons en possession d'un outil formidable, simple à installer, totalement multiplateforme et ne demandant pas d'infrastructure dédiée. Je ne vous cacherai pas que nous sommes loin d'avoir fait le tour des possibilités, mais nous avons vu les plus intéressantes. Pour aller encore plus loin, deux ou trois VMs dans un coin, un peu de temps et d'huile de coude sont les seuls outils nécessaires pour faire correspondre ce produit à nos besoins à 100 %. Bon courage !

RÉFÉRENCES

[1] **https://openvpn.net/index.php/open-source/documentation/howto.html#examples**

[2] **https://security.stackexchange.com/questions/54359/what-is-the-difference-between-diffie-hellman-generator-2-and-5)**

[3] **https://openvpn.net/index.php/open-source/documentation/manuals/**

[4] **https://openvpn.net/index.php/open-source/documentation/miscellaneous/79-management-interface.html**

ANNEXE

Afin de générer plus facilement une PKI complète pour vos tests, vous trouverez sur GitHub (**https://github.com/GLMF/GLMFHS92**) un petit script sans aucune prétention ni optimisation qui permet de tout recréer. Il part du principe qu'il est lancé dans `/etc/openvpn`, que la partie CA et serveur se retrouvera dans `keys/` et les certificats et clés pour les postes seront mis dans `clients/`. S'il est lancé sans option, il ne créera que les certificats clients à condition que la CA soit opérationnelle, s'il est lancé avec l'option `-s` il créera toute la PKI. ■

DISPONIBLE DÈS LE 29 SEPTEMBRE

MISC HORS-SÉRIE N°16 !

MAÎTRISEZ LES OUTILS MATÉRIELS & LOGICIELS POUR LES AUDITER

NE LE MANQUEZ PAS

CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

http://www.ed-diamond.com

CONFIGUREZ LES POSTES CLIENTS Y COMPRIS SUR DES SYSTÈMES NON LIBRES

Cédric PELLERIN & Sidoine PIERREL

L'utilisation d'OpenVPN n'est nullement limitée aux systèmes open source comme on a pu le voir précédemment. Chaque système a son mode de configuration bien à lui et nous allons en passer en revue quelques-uns.

1. WINDOWS

Inutile ici de préciser la version du système, car mis à part l'encadrement des fenêtres rien ne change de **Windows XP** à **Windows 10**. Le logiciel se télécharge sur le site d'**OpenVPN** et s'installe comme presque tous les programmes Windows en cliquant sur ***Suivant***, ***Suivant***...***Terminer*** (voir figure 1).

La seule différence concerne le fait qu'il faut aussi installer le *driver* tap pour Windows, ce qui se fait automatiquement après avoir accepté.

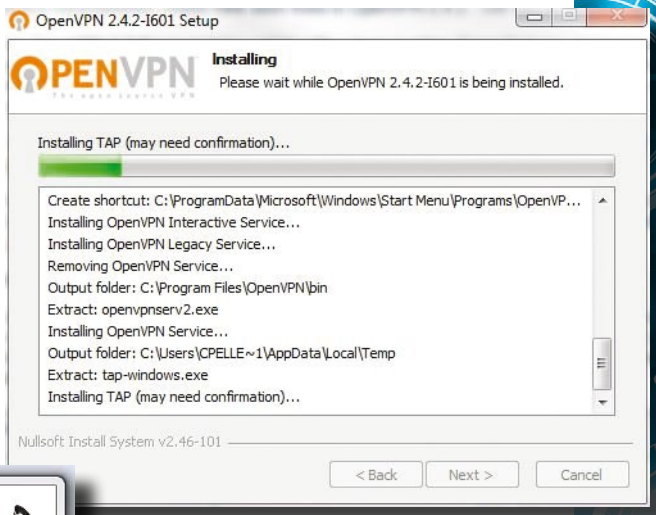


Fig. 1 : Installation du client OpenVPN sous Windows.

L'installation peut se révéler assez longue pour des raisons obscures et une fois achevée rien ne se passe. Il faut aller chercher dans le menu ***Démarrer*** pour lancer **OpenVPN GUI** qui va se contenter de nous rajouter une icône dans la barre d'état (voir figure 2).

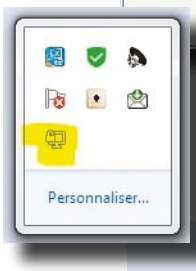


Fig. 2 : Icône OpenVPN.

C'est en effectuant un clic droit dessus (voir figure 3) que l'on peut importer un fichier de configuration préalablement préparé manuellement. Aucune aide à la création n'est proposée.

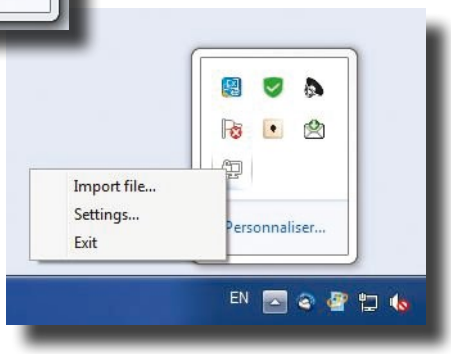


Fig. 3 : Menu contextuel.

En cliquant sur ***Settings***, on a juste accès à un mini-panneau de configuration plutôt simpliste (voir figure 4).

Une fois la configuration importée, les options deviennent un peu plus nombreuses comme on peut le voir sur la figure 5, page suivante.

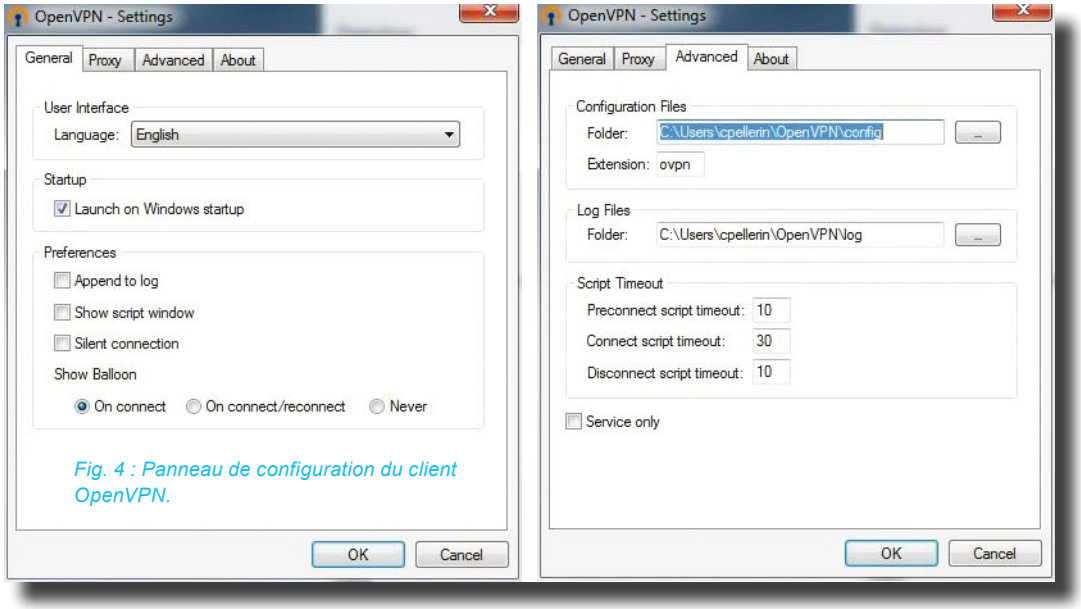


Fig. 4 : Panneau de configuration du client OpenVPN.

L'option ***Edit config*** se contente de lancer **Notepad** et d'ouvrir le fichier importé. Si notre configuration a été préparée sous Linux, on se retrouve avec les problèmes de retours chariot bien connus. Le fichier est situé par défaut dans `C:\Users\<votre_utilisateur_windows>\OpenVPN\Config\<config_sans_extension>\<config>.ovpn`. Pour être plus clair, si j'importe un fichier de configuration nommé `win.ovpn` (le fichier `roadwarrior.conf` renommé tout simplement), il se retrouvera dans `C:\Users\cpellerin\OpenVPN\Config\win\win.ovpn`.

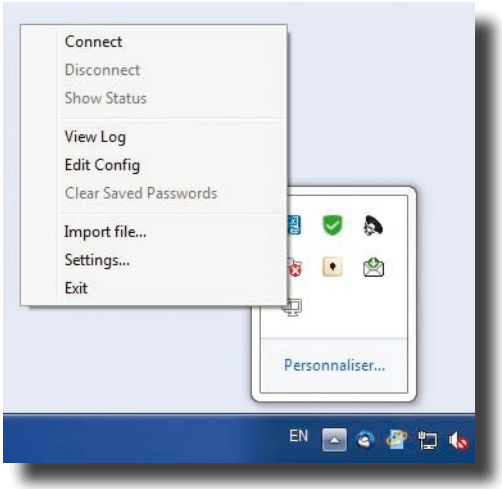


Fig. 5 : Options après import d'une configuration.

Pour modifier ce fichier, il est fortement conseillé d'oublier l'option d'édition du menu, du moins au début, et d'aller ouvrir le `.ovpn` avec un éditeur qui tient la route genre **Vim** ou **Notepad++**.

Comme la configuration est déjà isolée dans un sous-répertoire, j'ai décidé de poser les clés et certificats au même endroit que le fichier ovpn, donc si vous faites de même n'oubliez pas d'enlever les `keys/` devant le ca, le crt et la key.

Aucune option n'existe pour importer les clés et certificats, il faut donc les mettre au même endroit manuellement.

Une fois tout ceci réalisé, nous pouvons tenter une première connexion en cliquant à droite sur l'icône et en choisissant ***Connect***, ce qui va nous ouvrir une fenêtre de log qui disparaîtra toute seule si tout va bien et une petite bulle nous donnera notre adresse IP (voir figure 6).

Fig. 6 : Infobulle de connexion.

Nous sommes connectés, le reste est une affaire de routage plus ou moins simple en fonction de vos besoins et envies...

2. MAC OS

Le client pour **MacOS** s'appelle **Tunnelblick** et il fonctionne à partir de la version 10.4 dite « Tiger ». Cependant, le code n'est plus maintenu pour les versions de macOS inférieures à 10.7.5, notamment pour les machines PPC et x86 32 bits. De plus, il faut absolument une version d'**OpenSSL** supérieure ou égale à 0.9.8 sachant que Tiger propose une version 0.9.7... Je l'utilisais dans les années 2005-2006 et il fonctionnait très bien. Pour ceux qui veulent le tester, le `.dmg` est téléchargeable sur **https://tunnelblick.net**.

3. ANDROID

Le client OpenVPN utilisé ici s'appelle **OpenVPN Connect**. Il est assez simple à utiliser si l'on respecte certaines règles :

- l'extension du fichier de configuration doit obligatoirement être `.ovpn` ;
- si vous n'incluez pas vos certificats directement dans le fichier de configuration, il faut utiliser la KeyChain Android en important les certificats via un fichier au format PKCS#12 et ne rien mettre les concernant dans le fichier de configuration (pas de directive `ca`, `cert` ni `key`). C'est l'option que nous avons choisi, il faut donc créer avant tout ledit fichier via la commande habituelle sur le serveur :

Terminal

```
# cd /etc/openvpn
# openssl pkcs12 -in clients/bilbon.crt -inkey clients/bilbon.key -certfile
keys/ca.crt -export -out bilbon.p12
```

On va commencer par importer le fichier de « credentials » au format PKCS#12 qui aura été préalablement posé quelque part sur le système de fichier local ou sur la carte SD via le menu ***Import*** puis ***Import PKCS#12 from SD card***.

On choisit le bon fichier `.p12` et Android nous demande le mot de passe, comme d'habitude pour ce type de fichier.

Ceci étant fait nous pouvons aller importer le fichier de configuration, expurgé de ses directives `ca`, `cert` et `key` et renommé avec l'extension `.ovpn`. Si tout se passe bien, le logiciel nous demande de sélectionner le certificat de la KeyChain qu'il faut utiliser et nous pouvons nous connecter (voir figure 7).

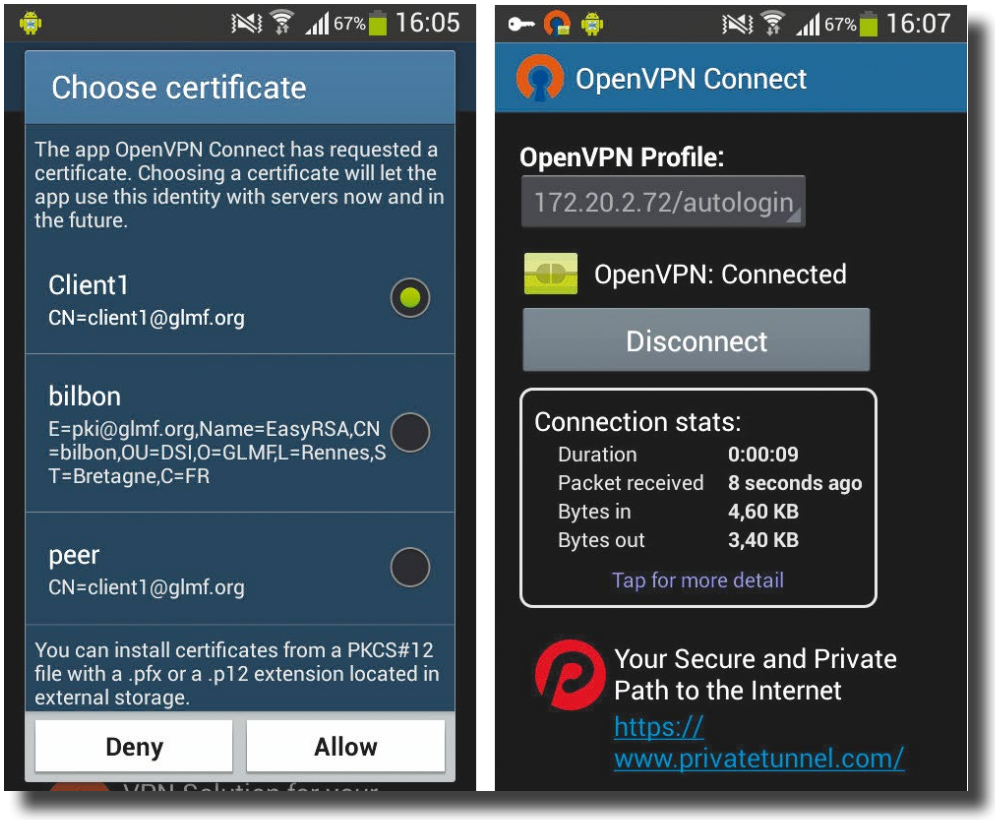


Fig. 7 : Choix du certificat et connexion.

CONCLUSION

Voici donc comment utiliser OpenVPN avec la plupart des systèmes aujourd'hui en usage. La procédure est presque toujours la même, mais certains détails ne sont pas faciles à trouver si on ne lit pas la documentation et que l'on ne maîtrise pas bien l'OS sous-jacent. Monter un VPN reste une affaire de gens qui s'y connaissent et c'est pour cela que ce hors-série a été écrit :) ▮

IPSEC

3

IPSEC

À découvrir dans cette partie...

Mettez en place un serveur IPSec

Dans cet article, nous détaillerons la configuration de StrongSwan, un serveur IPSec. Nous verrons que, bien qu'un peu plus complexe qu'OpenVPN, la mise en place d'un tel serveur reste tout de même assez simple. p. 90

Configurez les postes IPSec clients

Les postes clients sont forcément hétérogènes et il faut donc se pencher sur la configuration d'IPSec sous de multiples systèmes d'exploitation, supportant IKEv2 ou seulement L2TP/IPSec en IKEv1. p. 104

METTEZ EN PLACE UN SERVEUR IPSEC

Cédric PELLERIN

IPSec est la version « officielle » de tunneling pour TCP/IP. Il a été développé pour IPv6 puis backporté vers IPv4. Cet ensemble de protocoles est disponible sur toutes les plateformes, mais sa mise en œuvre est un peu plus complexe que celle d'OpenVPN.

1. INSTALLATION ET PKI

Cette première partie sera commune aux deux modes de fonctionnement d'**IPSec**. Le premier traité, car le plus récent et appelé à devenir le standard sera le mode tunnel utilisant IKEv2 – connu aussi sous le nom de IPSec only ou bare IPSec – et ensuite nous aborderons la partie « historique, mais se rencontrant encore sur le terrain » c'est-à-dire le mode transport en L2TP/IPSec et IKEv1.

Comme pour le reste de ce hors-série, nous partons sur une **Debian 8.7.1** que nous installons sans fioriture ni interface graphique. Seul le serveur SSH est rajouté pour nous simplifier la vie.

Une fois la base installée et configurée, nous pouvons mettre en place les paquets spécifiques à IPSec avec la commande :

Terminal
```
# apt install strongswan libcharon-extra-plugins
```

Afin de vérifier que tout va bien, nous pouvons lancer la commande :

Terminal
```
# ipsec version
Linux strongSwan U5.2.1/K3.16.0-4-amd64
Institute for Internet Technologies and Applications University of Applied
Sciences Rapperswil, Switzerland See 'ipsec --copyright' for copyright
information.
```

1.1 L'autorité de certification

Afin de nous éviter l'achat d'un certificat, nous allons autosigner les nôtres et pour ce faire, nous devons créer une autorité de certification ou CA. Nous avons vu comment faire avec **OpenSSL** dans l'article sur OpenVPN avancé, mais **StrongSwan** nous offre tout ce qu'il faut pour nous simplifier la vie. Utilisons donc la commande **`ipsec`**.

Tous les certificats et clés pour StrongSwan sont stockés dans des sous-répertoires de **`/etc/ipsec.d`**. Commençons donc par nous y mettre puis tapons la commande suivante qui créera la paire de clés en RSA 4096 :

Terminal
```
# ipsec pki --gen --type rsa --size 4096 --outform pem > private/strongswanKey.pem
```

Cette commande génère (**`--gen`**) une paire de clés de type RSA (**`--type rsa`**) d'une taille de 4096 bits (**`--size 4096`**) au format PEM (**`--outform pem`**) et stocke tout ça dans le fichier **`strongswanKey.pem`**. Il ne faut pas oublier de le rendre lisible uniquement par l'utilisateur **`root`** avec un **`chmod 600`** puis nous pouvons créer le certificat autosigné :

Terminal
```
# ipsec pki --self --ca --lifetime 3650 --in private/strongswanKey.pem --type
rsa --dn "CN=strongSwan Root CA" --outform pem > cacerts/strongswanCert.pem
```

L'option **`--lifetime 3650`** lui donne une durée de validité de 10 ans et les autres options parlent d'elles-mêmes.

1.2 Le certificat serveur

La CA étant en place, il nous faut maintenant un certificat pour le serveur. La paire de clé se génère de la même manière que pour le CA. Pour éviter de faire ramer les machines un peu lentes, nous allons nous limiter à 2048 bits pour celui-là :

Terminal

```
# ipsec pki --gen --type rsa --size 2048 --outform pem > private/vpnHostKey.pem
# chmod 600 private/vpnHostKey.pem
```

Il reste à le signer avec notre autorité de certification :

Terminal

```
# ipsec pki --pub --in private/vpnHostKey.pem --type rsa | \
ipsec pki --issue --lifetime 730 \
--cacert cacerts/strongswanCert.pem \
--cakey private/strongswanKey.pem \
--dn "CN=ipsec.glmf.org" \
--san ipsec.glmf.org \
--flag serverAuth --flag ikeIntermediate \
--outform pem > certs/vpnHostCert.pem
```

La commande étant fort longue, elle est ici présentée sur plusieurs lignes. Le premier appel à **ipsec pki** permet d'extraire la clé publique du certificat serveur. Cette clé est passée via un *pipe* au deuxième appel à **ipsec pki** qui va générer (**--issue**) un certificat en utilisant pour le signer le certificat et la clé de la CA. En plus de cela, les options **--dn** et **--san** nous permettent d'inclure un **distinguished name** et un **subject alternative name**. L'un des deux au moins (je préfère mettre les deux, c'est plus sûr) doit correspondre exactement avec le fqdn du serveur qui doit être résolvable. Dans le cas contraire, la connexion échouera systématiquement. Pour les essais, une entrée dans **/etc/hosts** suffit largement. Les deux *flags* rajoutés sont là pour faire plaisir respectivement à Windows (**serverAuth**) et à MacOS (**ikeIntermediate**). Comme leur ajout ne pose pas de problème pour quiconque, je les mets systématiquement.

1.3 Le certificat client

Les commandes sont presque exactement les mêmes que pour le serveur :

Terminal

```
# ipsec pki --gen --type rsa --size 4096 --outform pem > private/Client1Key.pem
# chmod 600 private/Client1Key.pem
# ipsec pki --pub --in private/Client1Key.pem --type rsa | \
ipsec pki --issue --lifetime 730 \
--cacert cacerts/strongswanCert.pem \
--cakey private/strongswanKey.pem \
--dn "CN=client1@glmf.org" \
--san client1@glmf.org \
--outform pem > certs/Client1Cert.pem
```

Pour un client, le **dn** et le **san** peuvent être leur adresse e-mail.

Afin de pouvoir importer plus facilement tout ce dont nous avons besoin dans notre PKI cliente, nous pouvons exporter tout cela au format PKCS12 via la commande OpenSSL :

Terminal

```
# openssl pkcs12 -export -inkey private/Client1Key.pem -in certs/Client1Cert.
pem -name "Client1 VPN Certificate" -certfile cacerts/strongswanCert.pem -caname
"strongSwan Root CA" -out Client1.p12
```

Pour protéger ce fichier, OpenSSL nous demande un mot de passe qu'il faudra fournir lors de l'import sur le client.

1.4 Révoquer un certificat

Un certificat peut être compromis soit par un vol, soit tout simplement, car la personne le possédant quitte la société. Pour faire savoir à StrongSwan que le certificat est compromis, on utilise une CRL pour *Certificates Revokation List*. À la première utilisation la CRL étant inexistante, nul besoin de tenir compte de révocations antérieures. La commande est assez simple :

Terminal

```
# cd /etc/ipsec.d/
# ipsec pki --signcrl --reason key-compromise \
--cacert cacerts/strongswanCert.pem \
--cakey private/strongswanKey.pem \
--cert certs/<premier_certificat_a_revoquer.pem> \ --outform pem > crls/crl.pem
```

Pour révoquer d'autres certificats, il faut tenir compte de la CRL existante et les commandes deviennent :

Terminal

```
# cd /etc/ipsec.d/
# cp crls/crl.pem crl.pem.tmp
# ipsec pki --signcrl --reason key-compromise \
--cacert cacerts/strongswanCert.pem \
--cakey private/strongswanKey.pem \
--cert certs/<certificat_a_revoquer>.pem \
--lastcrl crl.pem.tmp \
--outform pem > crls/crl.pem
# rm crl.pem.tmp
```

En résumé, on commence par faire une copie de sauvegarde de la liste de révocation existante, on précise à **`ipsec pki --signcrl`** d'en tenir compte (option --**`lastcrl`**) puis on efface la sauvegarde.

2. MODE TUNNEL EN IKEV2

2.1 Configuration du serveur

Un serveur StrongSwan utilise trois fichiers de configuration principaux :

- **`/etc/strongswan.conf`** qui est constitué par une série d'*includes* pour lui-même et pour le daemon IKE (*Internet Keys Exchange*) nommé fort à propos du doux nom de **Charon** ;
- **`/etc/ipsec.conf`** qui permet de configurer les connexions des divers types de clients possibles ;

⇨ **/etc/ipsec.secrets** qui regroupe les authentifications possibles (clés RSA, *login/password* en fonction de la méthode demandée par le client, etc.).

2.1.1 strongswan.conf

Normalement, ce fichier n'a pas à être retouché pour que StrongSwan fonctionne. Il permet d'inclure les divers modules possibles, principalement pour Charon et d'inclure toutes les configurations situées dans **/etc/strongswan.d**. Si l'on jette un coup d'œil aux fichiers contenus dans ce répertoire, on remarque que toutes les lignes sont commentées et que donc on utilisera les valeurs par défaut, ce qui nous convient très bien pour une première approche. Les fichiers présents sont :

⇨ **starter.conf** qui permet à StrongSwan de savoir où trouver la configuration ipsec (**ipsec.conf**) et de charger certains modules au démarrage ;

⇨ **charon.conf** qui regroupe toutes les options utiles lors de l'établissement d'un tunnel avec un client comme les adresses des serveurs DNS à fournir, la gestion des routes, tout un tas de paramètres permettant d'affiner le comportement d'IKE, etc. ;

⇨ **charon-logging.conf** qui s'occupe de la partie log de Charon ;

⇨ **pki.conf** qui permet d'inclure d'éventuels modules concernant la PKI ;

⇨ **scepclient.conf** qui concerne le protocole SCEP (*Simple Certificate Enrollment Protocol*) permettant à un serveur StrongSwan d'accepter la connexion de clients Cisco respectant cette norme ;

⇨ **tnc.conf** permettant la configuration de clients utilisant le protocole TNC (*Trusted Network Connect*).

Pour finir, on trouvera sous **strongswan.d/charon/** les fichiers de configuration pour chaque module de Charon.

2.1.2 ipsec.conf

Commençons avec une version minimaliste, mais fonctionnelle :

Fichier

```
 1 # ipsec.conf - strongSwan IPsec configuration file
 2
 3 # basic configuration
 4
 5 config setup
 6         # strictcrlpolicy=yes
 7         # uniqueids = no
 8         charondebug="cfg 2, dmn 2, ike 2, net 2"
 9
10 conn %default
11         keyexchange=ikev2
12         ike=aes128-sha256-ecp256,aes256-sha384-ecp384,aes128-
sha256-modp2048,aes128-sha1-modp2048,aes256-sha384-modp4096,aes256-
sha256-modp4096,aes256-sha1-modp4096,aes128-sha256-modp1536,aes128-
sha1-modp1536,aes256-sha384-modp2048,aes256-sha256-modp2048,aes256-
sha1-modp2048,aes128-sha256-modp1024,aes128-sha1-modp1024,aes256-
sha384-modp1536,aes256-sha256-modp1536,aes256-sha1-modp1536,aes256-
sha384-modp1024,aes256-sha256-modp1024,aes256-sha1-modp1024!
13         esp=aes128gcm16-ecp256,aes256gcm16-ecp384,aes128-sha256-
ecp256,aes256-sha384-ecp384,aes128-sha256-modp2048,aes128-sha1-
modp2048,aes256-sha384-modp4096,aes256-sha256-modp4096,aes256-sha1-
modp4096,aes128-sha256-modp1536,aes128-sha1-modp1536,aes256-sha384-
modp2048,aes256-sha256-modp2048,aes256-sha1-modp2048,aes128-sha256-
```

```
modp1024,aes128-sha1-modp1024,aes256-sha384-modp1536,aes256-sha256-
modp1536,aes256-sha1-modp1536,aes256-sha384-modp1024,aes256-sha256-
modp1024,aes256-sha1-modp1024,aes128gcm16,aes256gcm16,aes128-
sha256,aes128-sha1,aes256-sha384,aes256-sha256,aes256-sha1!
 14         dpdaction=clear
 15         dpddelay=300s
 16         rekey=no
 17         left=%any
 18         leftsubnet=0.0.0.0/0
 19         leftcert=vpnHostCert.pem
 20         right=%any
 21         rightdns=8.8.8.8,8.8.4.4
 22         rightsourceip=172.16.16.0/24
 23
 24 conn IPSec-IKEv2
 25         auto=add
 26
 27 conn IPSec-IKEv2-EAP
 28         also="IPSec-IKEv2"
 29         rightauth=eap-mschapv2
 30         rightsendcert=never
 31         eap_identity=%any
 32
```

Le fichier est assez parlant par lui-même, nous allons juste éclaircir certaines options :

- ligne 6 : si cette option est à **yes**, les connexions RSA vont exiger la présence d'une CRL, même si celle-ci est vide. Valeur par défaut à **no** ;
- ligne 7 : à **yes** (par défaut) empêchera toute connexion multiple avec la même combinaison certificat et/ou mot de passe. Normalement, cette situation ne devrait pas se produire sauf à donner le même certificat à plusieurs clients. Auquel cas le dernier arrivé déconnecte celui en place ;
- ligne 8 : on ajoute un peu de verbosité aux logs ;
- lignes 10 à 22 : on positionne les valeurs par défaut pour tous les types de connexion ;
- ligne 11 : on bloque IKE sur la v2 ;
- ligne 12 et 13 : on donne la liste des ciphers utilisables respectivement pour IKE (établissement de la connexion) et ESP (chiffrement des données dans le tunnel) ;
- lignes 14 et 15 : on détermine l'action à mener lorsqu'un client ne répond plus aux paquets de maintien de connexion (ici on supprime la connexion) et on positionne le *timeout* à **300** secondes (**30** par défaut) ;
- ligne 16 : ceci concerne la renégociation quand une connexion est sur le point d'expirer. Même mise à **no**, cette option n'empêchera pas une renégociation de s'effectuer si l'autre côté le demande expressément ;
- les lignes 17 à 19 concernent le côté serveur du tunnel. Dans IPSec, la gauche est toujours le côté local à la machine et son réseau tandis que la droite c'est le côté distant et son réseau. Pour s'en souvenir, *right* commence par 'r' comme *remote* et *left* commence par 'l' comme *local*. Ici on définit que le serveur peut être n'importe qui (**%any**) et que le certificat à utiliser sera **vpnHostCert.pem**. Normalement, on met le réseau local à la place de **%any**, mais ce raccourci peut être utile en cas de réseaux multiples, ou comme ici pour les tests ;

- ⇨ lignes 20 à 22 : on définit que la droite peut être aussi n'importe qui, en clair que l'on acceptera tout le monde comme client, on ne filtrera pas sur l'adresse IP. On positionne aussi les serveurs DNS et le *pool* d'adresses IP pour les tunnels ;
- ⇨ ligne 24 : on définit les paramètres propres à une connexion nommée **IPSec-IKEv2** ;
- ⇨ ligne 27 : on définit ceux pour une connexion de type EAP-MSCHAPv2.

StrongSwan se débrouille tout seul en fonction des paramètres envoyés par le client pour trouver la ou les configurations qui correspondent. Par exemple, si le client se connecte avec un certificat, le serveur va nous expliquer cela :

Terminal

```
13[CFG] looking for peer configs matching 172.20.2.72[%any]...172.20.2.74[CN=
cedric@bidouillesoft.net]
13[CFG] candidate "IPSec-IKEv2", match: 1/1/28 (me/other/ike)
13[CFG] candidate "IPSec-IKEv2-EAP", match: 1/1/28 (me/other/ike)
13[CFG] selected peer config 'IPSec-IKEv2'
13[CFG] using certificate "CN=cedric@bidouillesoft.net"
```

Et si on demande une authentification par *login*/*password* en EAP-MSCHAPv2 :

Terminal

```
06[CFG] looking for peer configs matching 172.20.2.72[%any]...172.20.2.74[172.
20.2.74]
06[CFG]   candidate "IPSec-IKEv2", match: 1/1/28 (me/other/ike)
06[CFG]   candidate "IPSec-IKEv2-EAP", match: 1/1/28 (me/other/ike)
06[CFG] selected peer config 'IPSec-IKEv2'
06[IKE] peer requested EAP, config inacceptable
06[CFG] switching to peer config 'IPSec-IKEv2-EAP'
```

Le tout sans rien toucher nulle part sur le serveur.

2.1.3 ipsec.secrets

Ce fichier regroupe toutes les informations nécessaires à l'authentification des connexions :

Fichier

```
: RSA vpnHostKey.pem
user1 : EAP "toto"
user2 : XAUTH "titi"
```

Ceci signifie simplement que si on s'authentifie par certificat RSA, il faut utiliser les clés contenues dans **vpnHostKey.pem**, si on a une demande en EAP, l'utilisateur doit s'appeler **user1** et le mot de passe est **toto**, en IKEv1+ XAUTH RSA l'accréditation se fera avec **user2** et le mot de passe **titi**.

Une fois tout ceci fait, il ne faut pas oublier de relancer StrongSwan avec la commande :

Terminal

```
# /etc/init.d/ipsec restart
```

Pour finir, il faut savoir que StrongSwan n'affecte pas d'adresse IP au côté serveur, il considère qu'il est sur un routeur et que ce n'est pas son travail. Comme les clients se voient affecter des adresses commençant à **1** nous allons, pour les tests, configurer une interface virtuelle en **254** :

Terminal

```
# ifconfig eth0:0 172.16.16.254/24
```

2.2 Les tests avec Windows 10

Dans le but de faire des tests rapidement, nous allons utiliser le client IPSec intégré à Windows 10. D'autres systèmes sont passés en revue dans l'article qui suit.

IPSec étant intégré à Windows 10 on pourrait s'attendre à ce que ce soit simple. En fait oui et non... La configuration du tunnel reste assez facile, mais l'importation du certificat est un cauchemar d'ergonome.

2.2.1 Importation du certificat

Pour importer un certificat, il faut lancer la **Microsoft Management Console** (`mmc.exe`) puis aller dans ***Fichier > Ajouter/Supprimer un composant logiciel enfichable*** et ajouter un Certificat en précisant bien dans la fenêtre suivante qu'il s'agit d'un compte d'ordinateur, c'est très important.

Une fois ceci fait, on revient sur la console de *management* et on ouvre ***Racine de la console > Certificats > Personnel > Certificats*** ensuite dans le cadre à droite on clique sur ***Autres actions > Toutes les tâches > Importer...*** et on suit le magicien jusqu'au bout en pensant à préciser une extension `*.pfx;*.p12` et non `*.cer;*.crt` comme suggéré par défaut.

Lorsque le mot de passe est demandé, on renseigne celui saisi lors de l'export de la section 1.3 et on coche la case pour dire que la clé est exportable. On peut demander à ce que le contenu soit placé automatiquement, ce qui devrait poser le certificat lui-même dans ***Personnel > Certificats*** et le CA dans ***Autorité de certification racine de confiance > Certificats***.

Une fois ceci fait, pour créer le VPN il faut commencer par un clic droit sur le menu ***Démarrer*** et choisir ***Connexions réseau***. L'écran de la figure 1 apparaît alors.

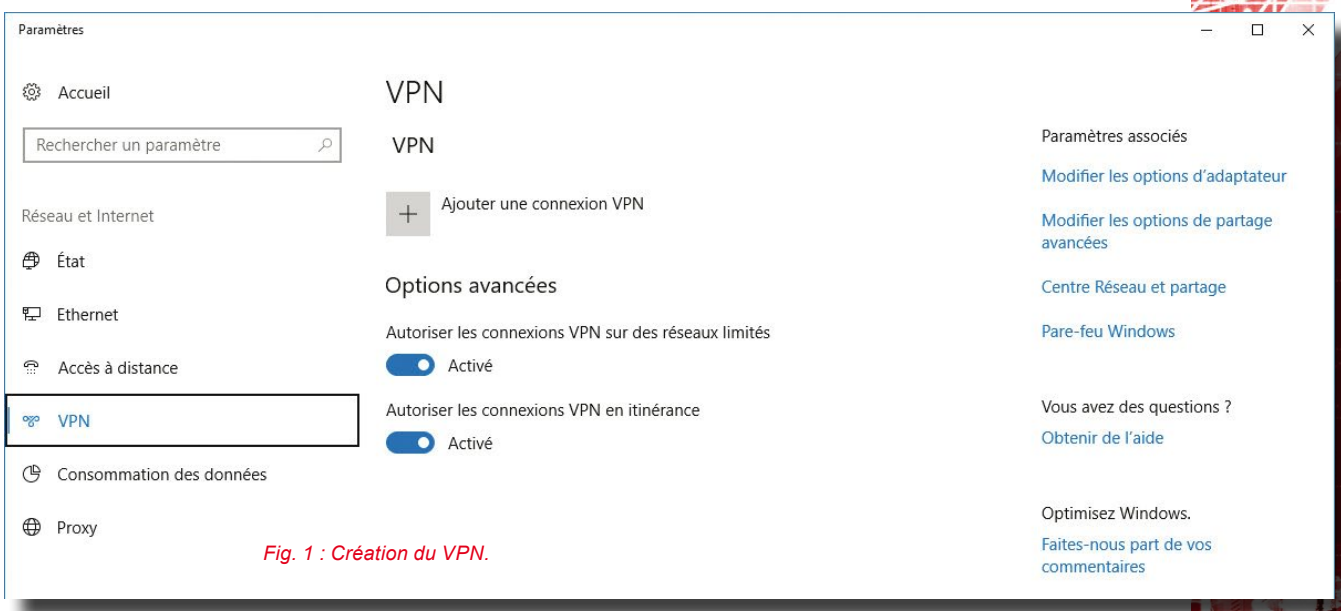


Fig. 1 : Création du VPN.

Choisir ***VPN*** à gauche puis cliquer sur ***Ajouter une connexion VPN***. On obtient un beau panneau bleu à remplir comme indiqué figure 2, page suivante.

Une fois ceci fait, il faut aller modifier la connexion « à la main », car l'option ***Certificat*** n'a pas été prise en compte correctement, du moins sur ma version de Windows 10. Pour ce faire, il faut trouver l'option ***Modifier les options d'adaptateur***, ce qui nous amène à une fenêtre nettement plus connue (voir figure 3, page suivante).

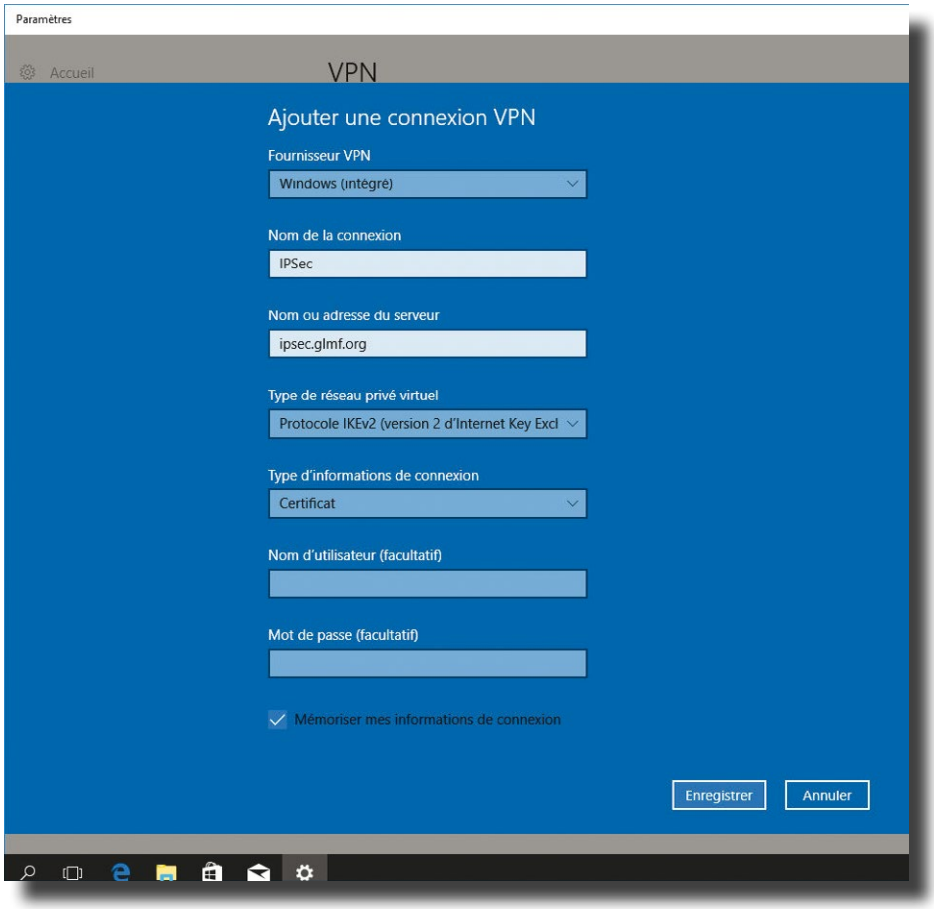


Fig. 2 : Configuration du VPN.

Fig. 3 : Modifier l'interface.

On clique à droite sur la connexion IPSec, on choisit l'onglet ***Sécurité*** puis on coche ***Utiliser des certificats de l'ordinateur***. Une fois tout ceci fait, on peut revenir à la configuration réseau, cliquer sur ***IPSec*** puis sur le bouton ***Se connecter*** et voilà, c'est fait.

Pour tester un autre mode d'authentification que le certificat, c'est simple, il faut revenir sur les propriétés de la connexion et au lieu de cocher ***Utiliser des certificats***, on coche ***Utiliser le protocole EAP*** et on choisit en dessous l'option ***Mot de passe sécurisé*** (EAP-MSCHAP version 2). Lors de la tentative de connexion, une *pop-up* apparaît demandant le *login* et le mot de passe. Il suffit de renseigner ceux que l'on a mis dans `/etc/ipsec.secrets` à la ligne ***EAP*** et c'est bon.

Après avoir cliqué sur ***Ok*** le tunnel devrait s'établir et la fenêtre nous afficher un « Connecté » de bel augure. Nous n'avons plus qu'à ouvrir une console et *pinger* le serveur en **172.16.16.254** pour vérifier que tout va bien.

3. MODE TRANSPORT EN L2TP/IPSEC ET IKEV1

Ce mode de fonctionnement repose en fait sur L2TP (*Layer 2 Tunneling Protocol*) pour créer le tunnel et sur IPSec pour le chiffrement, car L2TP n'en propose pas. Pour pouvoir disposer de L2TP sur notre serveur, il nous faut installer le paquet `xl2tpd` qui va rapatrier le paquet `ppp`. Ceux qui ont bricolé sur Internet à l'époque des modems doivent commencer à sentir leurs cheveux se hérisser...

3.1 La configuration serveur

Afin de pouvoir insérer notre connexion sans souci nous allons devoir bouger quelques options de `conn %default` vers les connexions IKEv2 existantes. C'est ainsi que la partie commune devient :

Fichier

```
conn %default
        keyexchange=ikev2
        ike=aes128-sha256-ecp256,aes256-sha384-ecp384,aes128-sha256-
modp2048,aes128-sha1-modp2048,aes256-sha384-modp4096,aes256-sha256-
modp4096,aes256-sha1-modp4096,aes128-sha256-modp1536,aes128-sha1-
modp1536,aes256-sha384-modp2048,aes256-sha256-modp2048,aes256-sha1-
modp2048,aes128-sha256-modp1024,aes128-sha1-modp1024,aes256-sha384-
modp1536,aes256-sha256-modp1536,aes256-sha1-modp1536,aes256-sha384-
modp1024,aes256-sha256-modp1024,aes256-sha1-modp1024!
        esp=aes128gcm16-ecp256,aes256gcm16-ecp384,aes128-sha256-
ecp256,aes256-sha384-ecp384,aes128-sha256-modp2048,aes128-sha1-
modp2048,aes256-sha384-modp4096,aes256-sha256-modp4096,aes256-sha1-
modp4096,aes128-sha256-modp1536,aes128-sha1-modp1536,aes256-sha384-
modp2048,aes256-sha256-modp2048,aes256-sha1-modp2048,aes128-sha256-
modp1024,aes128-sha1-modp1024,aes256-sha384-modp1536,aes256-sha256-
modp1536,aes256-sha1-modp1536,aes256-sha384-modp1024,aes256-sha256-
modp1024,aes256-sha1-modp1024,aes128gcm16,aes256gcm16,aes128-
sha256,aes128-sha1,aes256-sha384,aes256-sha256,aes256-sha1!
        dpdaction=clear
        dpddelay=300s
        rekey=no
```

Et que l'on rajoute les lignes suivantes à conn IPSec-IKEv2 et à conn IPSec-IKEv2-EAP :

Fichier

```
        keyexchange=ikev2
        left=%any
        leftsubnet=0.0.0.0/0
        leftcert=vpnHostCert.pem
        lefthostaccess=yes
        right=%any
        rightdns=8.8.8.8,8.8.4.4
        rightsourceip=172.16.16.0/24
```

Ceci afin qu'elles ne viennent pas perturber la connexion L2TP que nous rajoutons maintenant. Afin de changer un peu, nous allons travailler avec une clé partagée (Pre-Shared Key ou PSK) :

Fichier

```
18 conn L2TP-PSK
19         keyexchange=ikev1
20         authby=secret
21         auto=add
22         type=transport
23         left=172.20.2.72
24         leftsubnet=172.20.0.0/22
25         right=%any
26         rightprotoport=17/%any
```

Ce n'est franchement pas gros, mais tout y est :

- le nom de la nouvelle connexion est **L2TP-PSK** ;
- ligne 19 : on fixe la version de IKE à **1** ;
- ligne 20 : l'authentification se fait par clé partagée. On aurait pu mettre aussi **authby=psk**, c'est un synonyme ;
- ligne 22 : on spécifie que l'on travaille en mode **transport** et non en mode **tunnel** ;
- lignes 23 et 24 : on fixe l'adresse du serveur et les paramètres du réseau local ;
- ligne 25 : on accepte n'importe quelle tentative de connexion ;
- ligne 26 : on fixe le protocole à **UDP** (n° 17 cf. **/etc/protocols**) et on laisse libre le numéro de port.

Une fois ceci fait, on peut aller rajouter la clé dans **/etc/ipsec.secrets** :

Fichier

```
: PSK "mysecretkey"
...
```

On peut maintenant redémarrer Strongswan et aller regarder ce qui se passe dans **/etc/xl2tpd/xl2tpd.conf** pour la partie L2TP. Par défaut, il s'agit d'un assez long fichier dans lequel tout est commenté, mais assez peu d'options sont utiles pour notre cas. Commençons par fixer deux valeurs globales qui sont l'adresse du serveur et le numéro de port :

Fichier

```
1 [global]                              ; Global parameters:
2 port = 1701                         ; * Bind to port 1701
3 listen-addr = 172.20.2.72
```

Puis créons notre configuration serveur :

Fichier

```
 5 [lns default]
 6 ip range = 10.254.253.128-10.254.253.250
 7 local ip = 10.254.253.1
 8
 9 require chap = yes
10 refuse pap = yes
11 require authentication = yes
12 name = StrongswanVPN
13 ppp debug = yes
14 pppoptfile = /etc/ppp/options.xl2tpd
```

Pour **xl2tpd** une connexion serveur est décrite dans des sections préfixées par le trigramme **lns** – pour *L2TP Network Server* – alors qu'une connexion de type client sera dans une section préfixée par **lac** (*L2TP Access Concentrator*). Nous utilisons ici le nom **default** pour indiquer que c'est celle-là qui sera démarrée au lancement de **xl2tpd** sauf avis contraire.

Lignes 6 et 7, nous fixons les adresses à utiliser dans le tunnel. Ici le serveur sera en **10.254.253.1** et les stations auront une adresse prise dans le *pool* **10.254.253.128** à **250**.

Ligne 9 à 11, nous indiquons utiliser **CHAP**, interdire **PAP** (mot de passe transitant en clair non merci) et exiger une authentification.

Ligne 14, nous indiquons où trouver les options pour le protocole PPP sous-jacent.

Il nous reste à écrire la configuration de PPP dans **/etc/ppp/options.xl2tpd** :

Fichier

```
 1 ipcp-accept-local
 2 ipcp-accept-remote
 3 noccp
 4 auth
 5 crtscts
 6 idle 1800
 7 mtu 1200
 8 mru 1200
 9 nodefaultroute
10 proxyarp
11 connect-delay 5000
```

Ici les anciens vont se retrouver fortement rajeunis. En effet, nous allons utiliser des options que les moins de vingt ans ne peuvent pas connaître et qui ne servent sans doute plus à grand-chose, il faut bien l'avouer.

Lignes 1 et 2, nous expliquons à PPP d'utiliser les adresses IP fournies par L2TP.

Ligne 3, l'option **noccp** désactive la compression. Dans notre cas, ce n'est de toute façon pas à PPP de gérer cela.

Ligne 4, nous forçons une authentification qui se fera ici par **CHAP**.

La ligne 5 est un reliquat d'utilisation par modem qui indique d'utiliser les lignes de contrôle RTS et CTS pour piloter la connexion RS232C. Autant dire qu'ici elle ne sert rigoureusement à rien sauf si vous voulez essayer de vous connecter à 56kbps (max) en RTC...

Ligne 6, nous indiquons à PPP de se déconnecter au bout de 1800 secondes – soit 30 minutes – d'inactivité de la connexion.

En lignes 7 et 8, nous fixons le *Maximum Transmit Unit* et le *Maximum Receive Unit*. Si un jour vous utilisez des connexions par GSM, la modification de ces deux lignes pourrait vous sauver la mise dans certains cas.

En ligne 9, l'option **nodefaultroute** interdit à l'utilisateur d'utiliser le tunnel comme route par défaut.

L'option **proxyarp** de la ligne 10 permet de faire voir l'adresse MAC du tunnel au réseau local et ainsi d'éviter que des ARP *requests* ne trouvent pas de réponse.

Enfin en ligne 11 nous attendons maximum 5 secondes entre la fin du script de connexion et la réception du premier paquet. S'il n'arrive pas dans ce délai, la communication est coupée.

Et enfin nous pouvons nous occuper de configurer **CHAP** afin que l'authentification PPP fonctionne. Cela se fait dans le fichier **/etc/ppp/chap-secrets** :

Fichier

```
# client    server  secret          IP addresses
test            *       "test"      10.254.253.128/25
*               test    "test"      10.254.253.128/25
test2           *       "test"      10.254.253.128/25
*               test2   "test"      10.254.253.128/25
boss            *       "test"      10.254.253.2
*               boss    "test"      10.254.253.2
```

Nous avons ici trois *logins* possibles, tous avec le même mot de passe **test**. Pour **test** et **test2**, l'adresse IP sera dans la plage donnée et pour **boss** elle est fixe. Cette adresse sera celle affectée in fine à l'interface ppp donc celle du client dans le tunnel.

Nous pouvons maintenant relancer **xl2tpd** et passer aux clients.

3.2 Tests sous Windows 10

En ce qui concerne la configuration Windows, elle ne diffère que peu de celle vue en partie 2, aussi nous allons simplement créer un nouveau tunnel, cette fois en L2TP/IPSec.

Il faut tout d'abord ouvrir les connexions réseau via un clic droit sur le menu ***Démarrer*** puis choisir ***VPN*** puis ***Ajouter une connexion VPN*** comme pour la version IKEv2 et ensuite remplir le formulaire.

On enregistre puis il faut aller modifier des options bien cachées de la carte réseau virtuelle en cliquant sur ***Modifier les options d'adaptateur***, ce qui nous ouvre la fenêtre habituelle ; on clique à droite sur la carte nommée ***L2TP*** et on choisit ***Propriétés*** et l'onglet ***Sécurité*** (voir figure 4).

On vérifie que ***Protocole CHAP*** est bien coché puis on clique sur ***Paramètres avancés***.

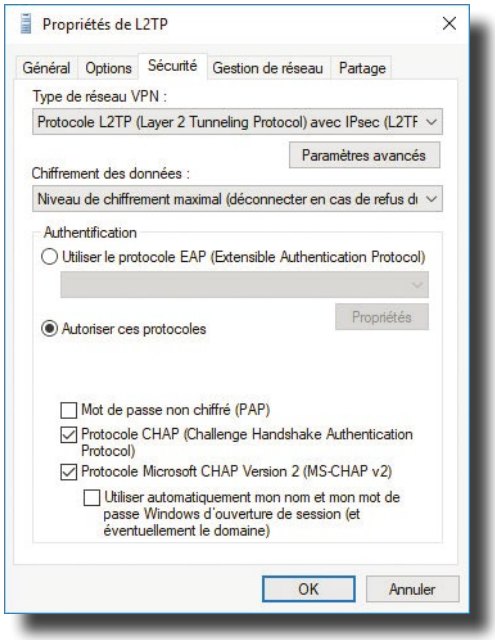


Fig. 4 : Paramètres de sécurité L2TP/IPSEC.

On saisit la clé renseignée dans `/etc/ipsec.secrets` côté serveur, on valide, on ferme la fenêtre et on revient sur la liste des connexions VPN où on clique sur ***L2TP*** (voir figure 5).

Fig. 5 : Connexion du tunnel L2TP.

On clique sur ***Se connecter***, on saisit l'un des noms et mots de passe CHAP renseignés dans `/etc/ppp/chap-secrets` et si on ne s'est pas trompé dans un coin, on doit obtenir la même chose qu'en figure 6.

Ceci permet ensuite d'obtenir la sortie de la figure 7 pour valider (Figure 7, ci-contre).

Fig. 6 : Fenêtre « Connecté ».

4. IPV6

Comme mentionné plus en avant, IPSec a été conçu comme couche de sécurité pour IPv6, ce dernier est donc naturellement supporté. Les seules modifications à effectuer par rapport aux configurations créées plus haut consistent à remplacer les adresses IPv4 par des adresses IPv6. Si, par exemple

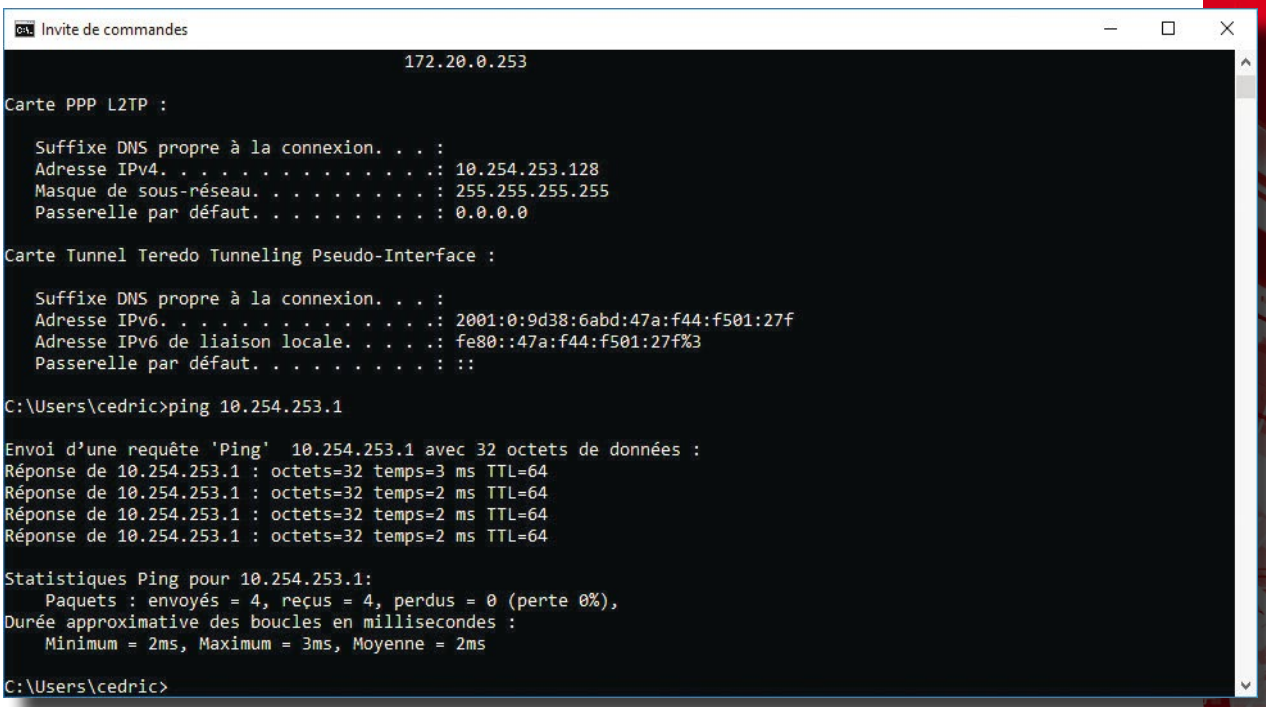


Fig. 7 : Validation de la connexion.

nous reprenons la configuration de la section 2.1.2 et que le réseau autorisé est en **2001:feed::/48**, les modifications à apporter sont les suivantes :

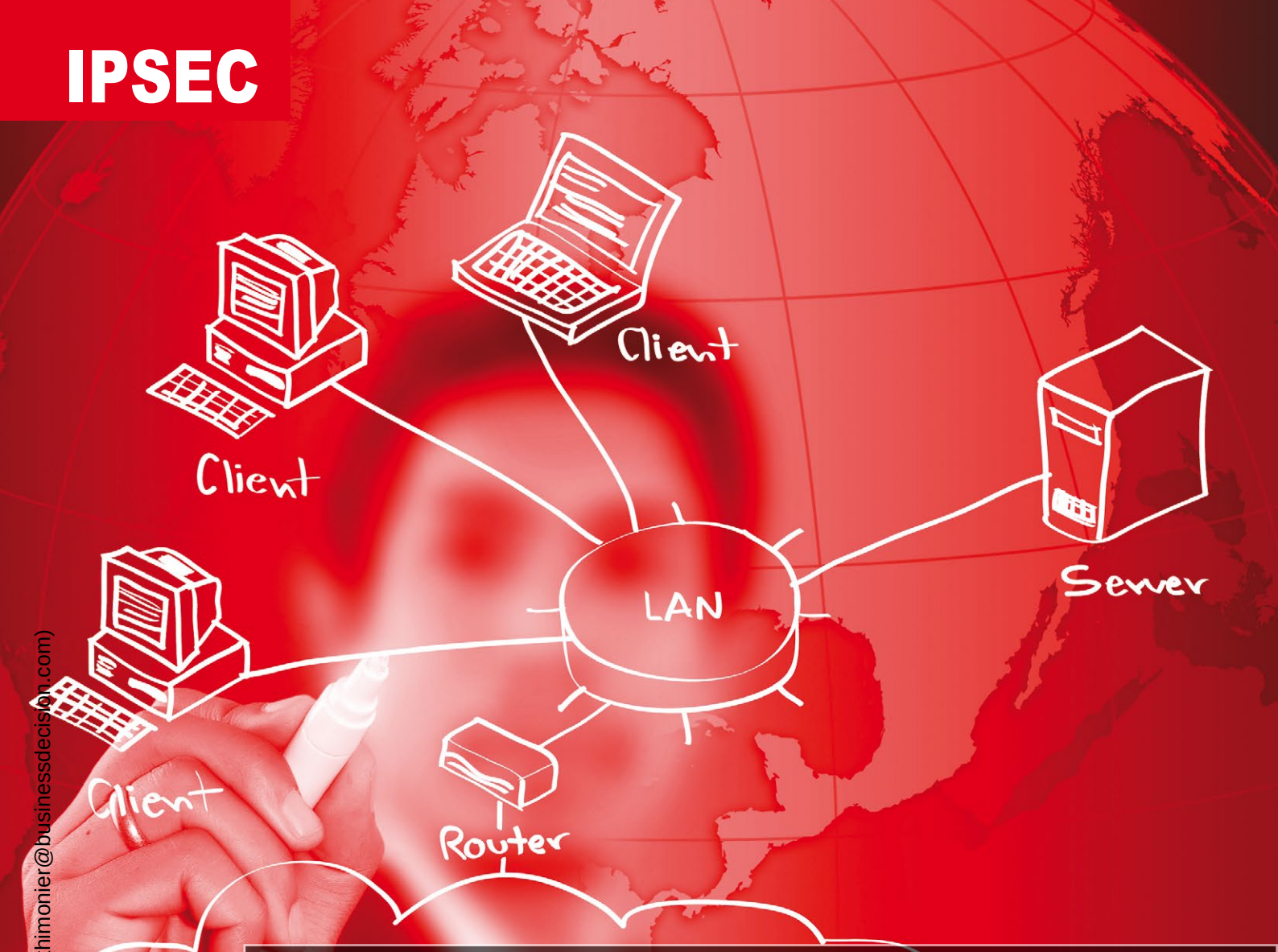
Fichier

```
17          left=%any
18          leftsubnet=::/0
19          leftcert=vpnHostCert.pem
20          right=%any
21          rightdns=2001:4860:4860::6464,2001:4860:4860::64
22          rightsourceip=2001:feed::/48
```

Il n'y a donc rien de bien sorcier si l'on connaît IPv6.

CONCLUSION

Nous avons pu voir dans cet article que, malgré les nombreuses options disponibles qui peuvent s'avérer déroutantes, monter un tunnel IPSec peut se faire simplement. Il est évident que nous n'avons fait qu'effleurer les possibilités de Strongswan. Ce logiciel dispose de nombreuses capacités supplémentaires soit en natif, soit via ses *plugins*, mais nous n'avons pas la place ici d'aller plus loin. De très bons tutoriels existent sur Internet, mais faites attention à ce qu'ils concernent bien la version de Strongswan que vous utilisez et si vous étudiez des informations concernant d'autres logiciels d'IPSec comme **FreeSwan**, **LibreSwan** ou autres, l'adaptation des fichiers de configuration n'est pas immédiate. Dans l'immédiat, vous êtes armés pour explorer et pour tenter, pour ceux qui le peuvent, la connexion avec des *appliances* de type **Cisco** ou **Juniper**. Bon courage et bonne chance... ■



CONFIGUREZ LES POSTES IPSEC CLIENTS

Cédric PELLERIN & Sidoine PIERREL

Dans cet article, nous allons aborder d'autres systèmes d'exploitation côté client, ce qui nous permettra de mettre en lumière certaines limitations. Nous allons tout d'abord passer en revue ceux qui fonctionnent en IKEv2 puis en examiner qui ne supportent que L2TP/IPSec en IKEv1.

1. CLIENTS EN IKEV2

1.1 Windows 7

Pourquoi parler de **Windows 7** alors que celui-ci est obsolète ? Tout simplement parce qu'un parc installé énorme existe encore et que de nombreux DSI sont réticents à passer sous **Windows 10**. Cette version va donc encore perdurer plusieurs années.

1.1.1 Configurer le client

Pour insérer les certificats, la procédure est exactement la même que celle pour Windows 10 indiquée dans l'article précédent.

Pour créer le VPN, il faut aller dans le panneau de configuration, choisir ***Centre Réseau et partage***, puis ***Configurer une nouvelle connexion ou un nouveau réseau***.

Dans la fenêtre qui apparaît, choisir ***Connexion à votre espace de travail*** puis préciser que l'on veut créer une nouvelle connexion et fournir le nom complet de votre serveur.

Pensez à cocher en bas ***Ne pas me connecter maintenant***. Le système vous demande un utilisateur et un mot de passe qui seront inutiles avec le certificat que l'on vient d'ajouter, vous devez ne rien saisir et c'est prêt.

NOTE

Le choix du nom à fournir ici doit impérativement correspondre avec le CN et le san indiqués lors de la création du certificat serveur sinon la connexion ne se fera jamais. Pour les tests, `/etc/hosts` et `%systemroot%\system32\drivers\etc\hosts` sont nos amis.

1.1.2 Établir la connexion

Toujours dans le panneau de configuration, choisir ***Centre Réseau et partage***, puis ***Modifier les paramètres de la carte***. Choisir la Connexion VPN que l'on vient de créer, clic droit dessus puis ***Propriétés***. Vérifiez dans l'onglet ***Général*** que l'adresse du serveur est bonne puis aller dans l'onglet ***Sécurité***, et dans ***Type de réseau VPN***, choisir ***IKEv2*** et cocher ***Utiliser des certificats d'ordinateur***.

Une fois ceci fait, on peut double-cliquer sur la connexion et le tunnel s'établit.

1.2 Linux

Avant de commencer, validons rapidement que notre interface virtuelle est bien montée côté serveur afin de pouvoir le *pinger* ensuite.

Nous avons déjà créé le certificat client, il nous reste à écrire les fichiers de configuration `/etc/ipsec.conf` et `/etc/ipsec.secrets`. Commençons par le premier qui va ressembler assez fortement à celui du serveur :

Fichier

```
1 config setup
2     # strictcrlpolicy=yes
3     # uniqueids = no
4
5 # Add connections here.
```

```
 6
 7 conn client1
 8     keyexchange=ikev2
 9     ike=aes128-sha256-ecp256,aes256-sha384-ecp384,aes128-sha256-
modp2048,aes128-sha1-modp2048,aes256-sha384-modp4096,aes256-sha256-
modp4096,aes256-sha1-modp4096,aes128-sha256-modp1536,aes128-sha1-
modp1536,aes256-sha384-modp2048,aes256-sha256-modp2048,aes256-sha1-
modp2048,aes128-sha256-modp1024,aes128-sha1-modp1024,aes256-sha384-
modp1536,aes256-sha256-modp1536,aes256-sha1-modp1536,aes256-sha384-
modp1024,aes256-sha256-modp1024,aes256-sha1-modp1024!
10     esp=aes128gcm16-ecp256,aes256gcm16-ecp384,aes128-sha256-
ecp256,aes256-sha384-ecp384,aes128-sha256-modp2048,aes128-sha1-
modp2048,aes256-sha384-modp4096,aes256-sha256-modp4096,aes256-sha1-
modp4096,aes128-sha256-modp1536,aes128-sha1-modp1536,aes256-sha384-
modp2048,aes256-sha256-modp2048,aes256-sha1-modp2048,aes128-sha256-
modp1024,aes128-sha1-modp1024,aes256-sha384-modp1536,aes256-sha256-
modp1536,aes256-sha1-modp1536,aes256-sha384-modp1024,aes256-sha256-
modp1024,aes256-sha1-modp1024,aes128gcm16,aes256gcm16,aes128-
sha256,aes128-sha1,aes256-sha384,aes256-sha256,aes256-sha1!
11     right=ipsec.glmf.org
12     rightid=%ipsec.glmf.org
13     rightsubnet=0.0.0.0/0
14     rightauth=pubkey
15     leftsourceip=%config
16     leftauth=pubkey
17     leftcert=Client1Cert.pem
18     auto=add
```

Détaillons les lignes les plus importantes :

- ligne 7 : le nom de la connexion qui nous servira plus tard. Nous avons ici nommé notre connexion **client1**, mais tout autre nom aurait pu faire l'affaire. Il est possible de définir plusieurs connexions dans le fichier **ipsec.conf** afin de répondre aux différents besoins ;
- ligne 8 : on précise la version de IKE à utiliser ;
- lignes 9 et 10 : les ciphers autorisés pour IKE et ESP ;
- ligne 11 : **right** indique le fqdn du serveur. À ce sujet, il faut bien vérifier que le client est capable de résoudre le nom utilisé ici soit via un DNS, soit en le mettant dans le fichier **/etc/hosts** ;
- ligne 12 : **rightid** doit absolument correspondre au CN du certificat serveur sinon la connexion est refusée ;
- ligne 14 : **rightauth** indique le mode d'authentification, échange de certificats en ce qui nous concerne ;
- ligne 15 : **leftsourceip** concerne la fourniture de l'adresse IP du tunnel. Avec **%config**, cela revient à la demander au serveur (une sorte de DHCP), mais on pourrait tout aussi bien la figer ;
- ligne 17 : **leftcert** indique quel certificat utiliser pour cette connexion ;
- ligne 18 : la directive **auto=add** nous permet ici de charger automatiquement la connexion au démarrage d'IPSec, mais sans la démarrer. Les options autres que **add** sont :
  - **route** qui installe des *sniffers* au niveau du kernel permettant de lancer automatiquement la connexion si un trafic est détecté entre **leftsubnet** et **rightsubnet** ;
  - **start** qui démarre immédiatement la connexion au lancement du *backend* Strongswan ;

→ **ignore** qui invalide la connexion, c'est comme si on l'avait effacé du fichier de configuration et c'est aussi la valeur par défaut donc attention...

Avant de lancer la connexion, il faut démarrer la partie *backend* de StrongSwan :

Terminal
```
# /etc/init.d/ipsec restart
```

Puis nous pouvons lancer notre connexion avec :

Terminal
```
# ipsec up client1
```

Si vous étiez en **ssh** sur votre poste client, normalement vous devez avoir perdu la main. Cependant, vous pouvez vous connecter depuis le serveur en utilisant l'adresse IP du tunnel donnée par le dernier message que vous avez vu avant d'être bloqué : **installing new virtual IP 172.16.16.1**.

Le fonctionnement de Strongswan fait qu'il change la route par défaut au fin fond des tables de routage afin d'utiliser l'interface physique à son propre usage. Pour le vérifier :

Terminal
```
# ip route show table 220
default via 172.20.2.72 dev eth0 proto static src 172.16.16.1
```

Au début ça surprend, mais après on s'y fait... Une fois tout ceci fait, il est possible de *pinger* chaque bout du tunnel depuis l'autre.

Pour ceux qui préfèrent les « clickodrômes », il existe un *plugin* pour **NetworkManager**, dont le *package* est nommé **network-manager-strongswan** sous **Debian** et qui se présente comme sur la figure 1 et ensuite il est possible de configurer les principaux paramètres de connexion (voir figure 2).

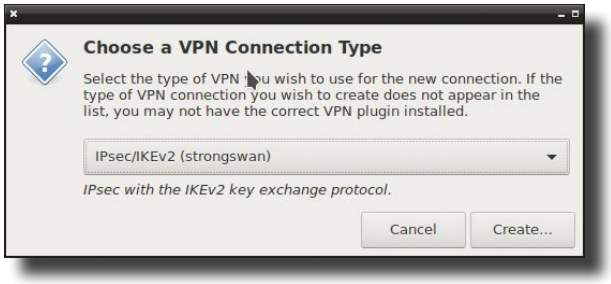


Fig. 1 : Connexion avec Network Manager.

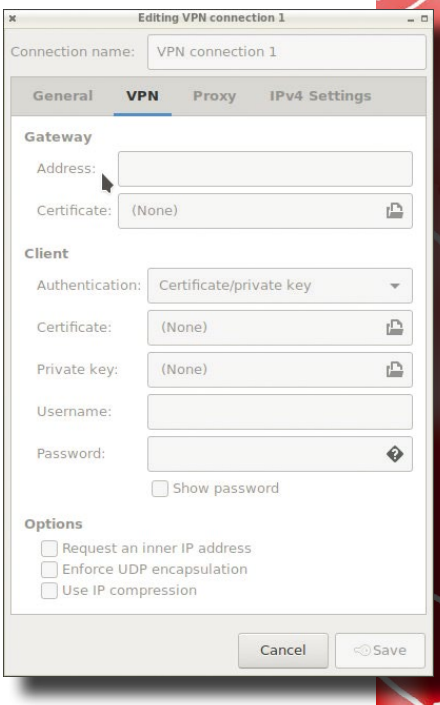


Fig. 2 : Paramètres VPN dans le Network Manager.

1.3 Android

1.3.1 Installation des certificats

Pour pouvoir poser sans problème l'ensemble des certificats sur un appareil **Android**, il faut les *packager* au format PKCS#12 en utilisant la commande **openssl**. Si notre client s'appelle **Client1**, voici comment il faut faire :

Terminal
```
# cd /etc/ipsec.d
# openssl pkcs12 -in certs/Client1Cert.pem -inkey private/Client1Key.pem
-certfile cacerts/strongswanCert.pem -export -out peer.p12
```

Cette commande va d'abord vous demander un mot de passe, puis le confirmer et ensuite elle créera un fichier nommé **`peer.p12`** qui contiendra l'ensemble des certificats et clés nécessaires. Il faut ensuite aller le poser sur le téléphone en utilisant un navigateur de fichiers quelconque ou **`adb`** dans le répertoire **`Download`** de la mémoire de l'appareil.

Une fois ceci fait, il faut aller dans les paramètres, onglet ***Plus***, option ***Sécurité*** et presque tout en bas on trouve l'option ***Installer depuis le stock. Périp..*** (voir figure 3).

On clique dessus et Android vous demande le mot de passe pour ouvrir le **`.p12`**, il s'agit de celui renseigné lors de la création. Après il vous est offert la possibilité de changer le nom sous lequel cet ensemble de clés et certificats sera présenté.

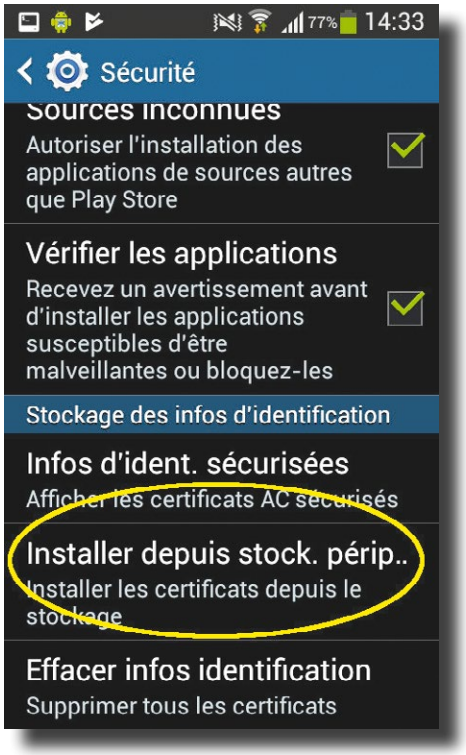


Fig. 3 : Installation de certificats dans Android.

1.3.2 Configuration du tunnel

Nous allons ici utiliser le client Strongswan officiel disponible sur **Google Play**. Sa configuration est assez simple, au lancement il faut créer un profil de tunnel (***add VPN profile***) et remplir les champs nécessaires (voir figure 4).

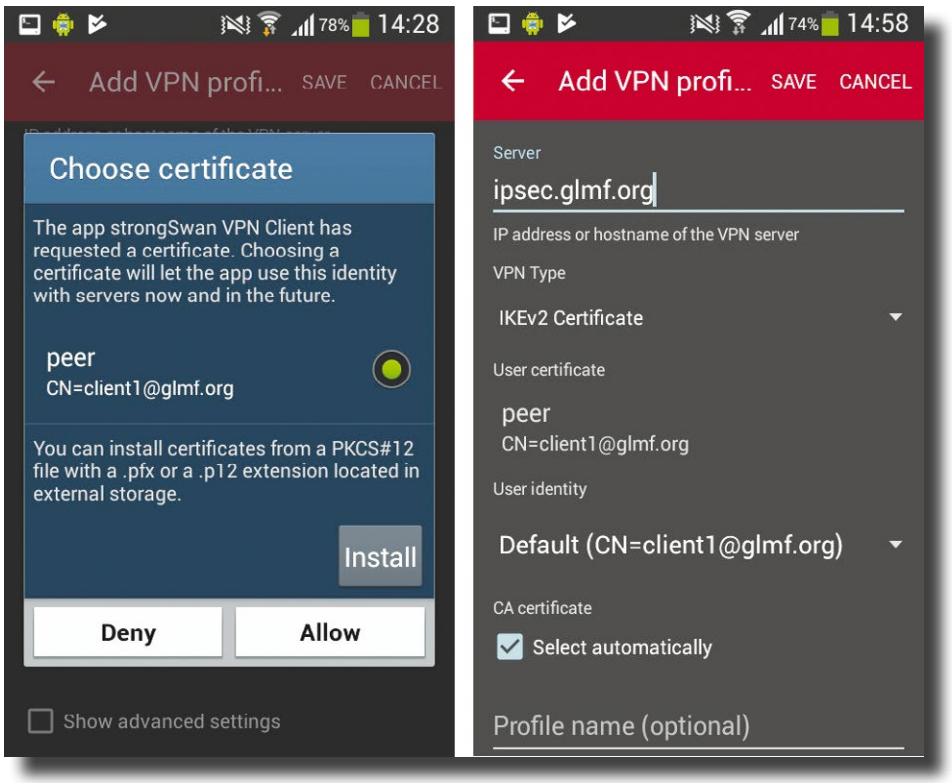


Fig. 4 : Paramétrer le client Strongswan.

Une fois tout ceci effectué, on sauvegarde et il ne reste plus qu'à nous connecter en cliquant sur le nom du profil créé. En cas de soucis, le logiciel nous donne accès au fichier de log intégral qui est le même que sous Linux, nous sommes donc en terrain connu.

1.4 Les autres clients

Le client Strongswan de même que le client **Apple** intégré pour IKEv2 ne semblent disponibles que pour **MacOS 10.6** et supérieurs. Ne possédant qu'un Mac avec un MacOS 10.4, les tests ne seront faits que sur la version L2TP/IPSec qui va être vue ci-après.

2. CLIENTS EN L2TP/IPSEC IKEV1

2.1 Windows 7

Windows 7 est parfaitement capable de se connecter aussi en L2TP/IPSec. La configuration est la même que pour IKEv2 à quelques exceptions près :

- ⇨ choisir une connexion L2TP-IPSec et non une connexion VPN ;
- ⇨ se reporter à la configuration pour Windows 10 vue dans le précédent article.

2.2 Mac OS/X

Pour fonctionner sous Mac OS/X 10.4 (désolé je n'ai pas plus récent en stock), il va falloir commencer par laisser accessible la totalité des ciphers disponibles pour IKE. Il va juste suffire de commenter la ligne commençant par `ike=` dans `/etc/ipsec.conf` puis de relancer `ipsec`. En effet, le Tigre semble ne pas supporter les ciphers récents (en même temps il date quand même de 2005, on peut difficilement lui en vouloir).

Sous **MacOS**, il faut aller dans les ***Applications*** et choisir ***Connexion à internet***. Dans la fenêtre qui apparaît, on clique sur ***VPN (L2TP)***, on renseigne l'adresse IP du serveur et les identifiants CHAP (voir figure 5).

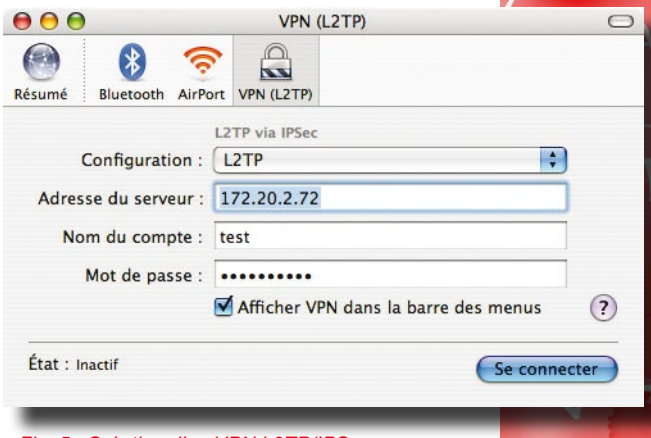


Fig. 5 : Création d'un VPN L2TP/IPSec sous Mac OS/X

Allez ensuite dans la liste ***Configuration*** et choisissez ***Modifier les configurations***. Si ce n'est pas encore fait, le système vous propose avant tout de sauvegarder votre configuration en lui donnant un nom. Ensuite, il faut remplir les champs de manière traditionnelle. Le champ ***Mot de passe*** est à remplir avec le mot de passe CHAP et le champ ***Secret partagé*** doit être renseigné avec la clé partagée (PSK). On peut alors enfin cliquer sur ***Se connecter*** !

2.3 Les autres clients

Les tunnels L2TP/IPSec sont supportés par 90 % des systèmes un tant soit peu modernes. Android possède un client natif de même qu'**iOS** au moins depuis la version 9. Tous les routeurs propriétaires que vous pourrez trouver le supportent aussi quasi certainement.

CONCLUSION

Il existe certainement un client IPSec pour à peu près tous les systèmes d'exploitation raisonnablement récents. Leur configuration est plus ou moins simple, mais surtout ils peuvent ne pas supporter encore IKEv2. Bien vérifier cela surtout en production afin de ne pas monter un tunnel qui ne sera pas supporté par l'ensemble du parc et se retrouver avec plusieurs serveurs à maintenir et à faire papoter entre eux. ▮

ALLEZ PLUS LOIN AVEC OPENVPN

4 ALLEZ PLUS LOIN AVEC OPENVPN

À découvrir dans cette partie...

Intégrez un firewall à OpenVPN

En mode bridge les paquets ne passent pas par le firewall Linux. Pour intégrer un filtrage, il faut donc écrire un petit plugin. p. 112

Développez vos plugins OpenVPN

OpenVPN offre la possibilité de développer des plugins en C de manière à répondre de manière encore plus précise à vos besoins. Vous apprendrez dans cet article comment en réaliser un. p. 118

INTÉGREZ UN FIREWALL À OPENVPN

Cédric PELLERIN

Lorsque l'on utilise le mode bridge d'OpenVPN et donc les interfaces de type tap, les paquets ne passent pas par le firewall Linux, mais sont traités directement par le serveur qui fait office de switch. Une fonction méconnue et peu documentée permet quand même de filtrer un minimum les communications de client à client, et ce dynamiquement.

Dans l'article précédent, nous avons vu que l'un des événements côté serveur pouvant être attrapé par une extension s'appelle OPENVPN_PLUGIN_ENABLE_PF. Lorsqu'un *plugin* capture cet événement et ne sort pas en erreur, une variable d'environnement nommée `pf_file` est créée et contient un nom de fichier dans lequel OpenVPN s'attend à trouver des règles de filtrage. Cette variable est dès lors disponible avec les autres lors des appels de « hook » comme `client-connect`.

Ce *plugin* ne fait rien d'autre que de demander à OpenVPN de se préparer à recevoir des règles de filtrage et à positionner une variable d'environnement.

Bien entendu, les informations à mettre dans le fichier dont le nom est dans `pf_file` doivent respecter une syntaxe bien précise.

1. LA SYNTAXE DES FICHIERS PF

La grammaire à utiliser reprend celle bien connue des fichiers `.ini` datant de Windows 3, à savoir des sections commençant par une ligne comprenant un nom entre crochets suivie par une série de lignes du type `directive=valeur`. Dans notre cas, les sections possibles sont :

- `[clients accept]` ou `[clients drop]` qui définit la politique par défaut vis-à-vis des autres clients ;
- `[subnets accept]` ou `[subnets drop]` qui fait pareil vis-à-vis des sous-réseaux rajoutés dynamiquement via des `push "route..."` ;
- `[kill]` qui ordonne au serveur de déconnecter le client dès qu'il tente de se connecter ;
- `[end]` qui marque la fin du fichier.

Chaque section définit la politique globale du filtrage. Si la politique est de type `accept`, il est possible de bloquer spécifiquement certains clients en mettant leur nom précédé du signe `-` dans la section.

Si la politique est de type `drop`, il est possible d'autoriser spécifiquement certains clients en mettant leur nom précédé du signe `+` dans la section.

Les sections `[clients accept]` et `[clients drop]` sont mutuellement exclusives, c'est-à-dire que l'on aura l'une ou l'autre, mais pas les deux dans un fichier pf. C'est la même chose pour les sections `[subnets accept]` et `[subnets drop]`.

La section `[end]` est obligatoire, mais des sections `[clients]` et `[subnets]`, une seule peut être utilisée.

1.1 Exemple

Afin de clarifier tout cela, rien ne vaut un exemple simple. Supposons que nous voulons interdire à « Bob » de voir qui que ce soit sauf la machine de « Jean », autoriser « Fred » à voir tout le monde sauf « Bob » tandis que « Jean » n'aura aucune restriction. Les communications étant bidirectionnelles si un premier poste doit avoir accès à un second, il faut que le second ait accès au premier, autrement dit la matrice décrivant les autorisations doit être symétrique. Voici ce que cela nous donne :

	Bob	Fred	Jean
Bob	-	✗	✓
Fred	✗	-	✓
Jean	✓	✓	-

Le nom utilisé pour identifier chaque poste doit être celui renseigné dans le *Common Name* du certificat client.

Ceci traduit en fichiers pf nous donne :

⇨ `Jean.pf` :

Fichier

```
[clients accept]
[subnets accept]
[end]
```

Les sections sont vides, car la politique est `accept` et il n'y a aucune exception ;

⇨ `Fred.pf` :

Fichier

```
[clients accept]
-Bob
[subnets accept]
[end]
```

Fred accepte tout le monde (politique `accept`) sauf Bob (`-Bob`) ;

⇨ `Bob.pf` :

Fichier

```
[clients drop]
+Jean
[subnets accept]
[end]
```

Bob n'accepte personne (politique `drop`) sauf Jean (`+Jean`).

2. LE DÉVELOPPEMENT DU PLUGIN

Afin d'activer la prise en charge du *firewall*, il suffit que le *plugin* qui attrape `OPENVPN_PLUGIN_ENABLE_PF` retourne `OPENVPN_PLUGIN_FUNC_SUCCESS`. Le code source de `pf.c` qui donnera le plugin pf.so est des plus simples :

Fichier

```
1  // pf.c to compile as a dynamic library named pf.so
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "openvpn-plugin.h"
5
6  // We don't need any static information but we need to return
   // something in openvpn_plugin_open_v2
7  struct context
8  {
9      int dummy;
10 };
11
12 OPENVPN_EXPORT openvpn_plugin_handle_t
```

```
 13 openvpn_plugin_open_v2(unsigned int *type_mask, const char *argv[],
const char *envp[], struct openvpn_plugin_string_list **return_list)
 14 {
 15     struct context *ctx;
 16     ctx = (struct context *)calloc(0, sizeof(struct context));
 17
 18     *type_mask = OPENVPN_PLUGIN_MASK (OPENVPN_PLUGIN_ENABLE_PF);
 19
 20     return (openvpn_plugin_handle_t) ctx;
 21 }
 22
 23 OPENVPN_EXPORT int
 24 openvpn_plugin_func_v2 (openvpn_plugin_handle_t handle, const int
type, const char *argv[], const char *envp[], void *per_client_context,
struct openvpn_plugin_string_list **return_list)
 25 {
 26
 27     if (type == OPENVPN_PLUGIN_ENABLE_PF)
 28     {
 29         return OPENVPN_PLUGIN_FUNC_SUCCESS;
 30     }
 31     else
 32     {
 33         /* should not happen! */
 34         return OPENVPN_PLUGIN_FUNC_ERROR;
 35     }
 36 }
 37
 38 OPENVPN_EXPORT void
 39 openvpn_plugin_close_v1 (openvpn_plugin_handle_t handle)
 40 {
 41     struct context *ctx = (struct context *)handle;
 42     free (ctx);
 43 }
```

Si vous avez lu avec attention l'article consacré au développement des *plugins*, vous pouvez constater que celui-ci ne fait juste… rien. Il retourne avec succès, ce qui suffit.

La partie active du code sera effectuée par un simple script bash nommé **`applypf.sh`** qui sera appelé via la directive **`client-connect`** à rajouter à la configuration du serveur. Il ne faut pas oublier la directive **`script-security`** à descendre au niveau 3 pour permettre le passage des arguments via les variables d'environnement :

Fichier

```
script-security 3
plugin /etc/openvpn/pf.so
client-connect /etc/openvpn/applypf.sh
...
```

Commençons par un script simple pour valider notre architecture :

Fichier

```
# applypf.sh
#!/bin/bash
```

```
echo "CN = ${common_name}" > /tmp/toto
echo "PF = ${pf_file}" >> /tmp/toto
exit 0
```

Ce script va se contenter de *dumper* les variables qui nous intéressent. Rendons-le exécutable puis relançons le serveur :

Terminal

```
NOTE: the current --script-security setting may allow this configuration to call
user-defined scripts
PLUGIN_INIT: POST /etc/openvpn/pf.so '[/etc/openvpn/pf.so] [/tmp/toto]'
intercepted=PLUGIN_ENABLE_PF
```

Le serveur OpenVPN nous avertit au sujet du `script-security` et nous informe qu'il a bien chargé notre *plugin*.

Maintenant, démarrons le client, attendons que la connexion soit établie puis allons voir notre fichier `/tmp/toto` :

Terminal

```
# cat /tmp/toto
CN = Jean
PF = /tmp/openvpn_pf_62fc6a33c4e63fdd94bf30a13636c021.tmp
```

OpenVPN nous a même préparé le fichier qui va bien dans `/tmp` :

Terminal

```
# ls -l /tmp
total 8
-rw------- 1 root root   0 May 22 16:21 openvpn_pf_62fc6a33c4e63fdd94bf30a13636
c021.tmp
-rw------- 1 root root 232 May 22 16:20 ovpn.status
-rw-r--r-- 1 root root  75 May 22 16:21 toto
```

Il ne nous reste plus qu'à remplir ce fichier avec les informations qui vont bien et voir ce qui se passe.

Commençons avec les règles pour le client « Jean » dans le fichier `/etc/openvpn/pf_rules/Jean.pf` :

Fichier

```
[clients accept]
[subnets accept]
[end]
```

Puis modifions le script pour en tenir compte :

Fichier

```
#!/bin/bash

rules_file="/etc/openvpn/pf_rules/${common_name}.pf"
if [ -f "${rules_file}" ] && [ ! -z "${pf_file}" ]; then
 cat "${rules_file}" > "${pf_file}"
else
 # if anything is not as expected, fail
```

```
    exit 1
fi
exit 0
```

En résumé, on construit la variable `rules_files` en rajoutant le chemin d'accès et on recopie dans le fichier pointé par elle le contenu du fichier de règles du client concerné.

On relance tout et quand notre client se connecte, le serveur nous informe d'une modification importante :

Terminal

```
OPTIONS IMPORT: reading client specific options from: /tmp/openvpn_cc_
d8b63c25decede0d197743a7f67423a0.tmp
```

C'est bon, nous voyons qu'OpenVPN sait lire les informations du fichier de règles.

Pour tester tout cela, nous allons simplement monter nos deux autres VMs clientes Bob et Fred. Pour ce faire, il faut juste recopier la configuration OpenVPN de Jean sur Bob et sur Fred, créer de nouveaux certificats et clés client sur le serveur puis les recopier sur les machines respectives et nous sommes prêts. Si vous ne savez pas encore comment faire, tout est expliqué dans l'article « Créez une configuration OpenVPN simple » dans cette même section.

Si l'on veut que Bob et Fred puissent se connecter, il ne faut pas oublier de créer leurs fichiers de règles comme indiqué plus haut sinon le *plugin* sortira en erreur directement.

Une fois les trois machines connectées, tentons de *pinger* Bob depuis Jean qui devrait être le seul autorisé à le faire :

Terminal

```
Jean# ping bob
PING compta1 (172.31.0.101) 56(84) bytes of data.
64 bytes from compta1 (172.31.0.101): icmp_seq=1 ttl=64 time=1.71 ms
64 bytes from compta1 (172.31.0.101): icmp_seq=2 ttl=64 time=0.724 ms
64 bytes from compta1 (172.31.0.101): icmp_seq=3 ttl=64 time=0.666 ms
```

Puis essayons de *pinger* Bob depuis Fred :

Terminal

```
Fred# ping bob
PING compta1 (172.31.0.101) 56(84) bytes of data.
From 172.31.0.100 icmp_seq=1 Destination Host Unreachable
From 172.31.0.100 icmp_seq=2 Destination Host Unreachable
From 172.31.0.100 icmp_seq=3 Destination Host Unreachable
```

Nous obtenons bien le résultat attendu. Je vous laisse le soin de valider les autres configurations et de faire des tests plus approfondis...

CONCLUSION

Cette petite extension, très peu documentée, peut paraître anecdotique, mais elle s'avère bien utile dans certains cas de figure, principalement lorsque l'on est en mode bridge et que l'on désire maîtriser le trafic client à client, car nous sommes là dans une configuration qui rend `iptables` totalement inopérant. ▮

DÉVELOPPEZ VOS PLUGINS OPENVPN

Cédric PELLERIN

Comme de nombreux logiciels dignes de ce nom, OpenVPN est capable d'accepter des plugins écrits en C et disponibles sous la forme de bibliothèques partagées. Regardons comment écrire les nôtres.

OpenVPN est livré avec tellement de possibilités de configuration et d'extensions via des scripts qu'on se demande parfois ce que l'on pourrait vouloir rajouter. Cependant selon le bon vieux principe du « plus on en rajoute plus le client en veut », il faut parfois se retrousser les manches et mettre les mains dans le cambouis. Nous allons ici écrire un petit *plugin* en C qui permettra simplement d'aller vérifier la présence d'une chaîne de caractères arbitraire dans un fichier. C'est très bête, mais si ce fichier est sur une clé USB, cela rajoute une couche de sécurité non négligeable : pas de clé, pas de connexion.

NOTE

Cet article est rédigé pour la version 2.4.0 d'OpenVPN. Les numéros de version et les prototypes sont sujets à changement en fonction de la version d'OpenVPN utilisée.

1. LES PRINCIPES

Chercher de la documentation sur l'écriture de *plugins* sous OpenVPN semble peine perdue jusqu'à ce que l'on pense à aller regarder les sources et notamment le fichier `openvpn-plugin.h`. Ce dernier est parfaitement documenté et montre que la gestion des *plugins* sous OpenVPN est presque un jeu d'enfant.

Il va nous falloir écrire trois fonctions de base pour que notre bibliothèque soit reconnue comme un *plugin* valable par le serveur OpenVPN :

- ⇨ `openvpn_plugin_open_v2` qui s'exécute au chargement du *plugin*, donc au démarrage d'OpenVPN ;
- ⇨ `openvpn_plugin_func_v2` qui est le cœur du *plugin* ;
- ⇨ `openvpn_plugin_close_v1` s'exécute à la fermeture d'OpenVPN.

D'autres fonction optionnelles sont disponibles :

- ⇨ `openvpn_plugin_client_constructor_v1` sert à allouer de la mémoire client par client ;
- ⇨ `openvpn_plugin_client_destructor_v1` est appelé à chaque destruction d'une instance client ;
- ⇨ `openvpn_plugin_select_initialization_point_v1` permet de sélectionner le moment où OpenVPN va appeler `openvpn_plugin_open` ;
- ⇨ `openvpn_plugin_min_version_required_v1` permet de figer la version minimale de l'interface des *plugins*.

Les `_v*` à la fin des noms de fonction reflètent la version de l'interface. Les fonctions `openvpn_plugin_open_v1` et `openvpn_plugin_func_v1` existent, mais sont dépréciées.

1.1 Description des fonctions

1.1.1 openvpn_plugin_open_v2

Fonction obligatoire.

```
openvpn_plugin_handle_t openvpn_plugin_open_v2(unsigned int *type_mask, const char *argv[], const char *envp[], struct openvpn_plugin_string_list **return_list);
```

- ⇨ `type_mask` est à positionner par la routine pour indiquer à OpenVPN sur quel(s) événement(s) elle sera appelée. Il s'agit de valeurs à transformer en *flags* via la macro `OPENVPN_PLUGIN_MASK` et ensuite un OR est possible pour cumuler les événements. Les valeurs possibles sont :
  - → `OPENVPN_PLUGIN_UP` : OpenVPN démarre ;
  - → `OPENVPN_PLUGIN_DOWN` : OpenVPN s'arrête ;
  - → `OPENVPN_PLUGIN_ROUTE_UP` : des routes sont ajoutées ;

- → **OPENVPN_PLUGIN_IPCHANGE** : le client VPN change d'adresse IP sur son interface physique (client DHCP par exemple) ;
- → **OPENVPN_PLUGIN_TLS_VERIFY** : permet de vérifier finement le contenu des certificats quand une connexion TLS est en attente de validation après avoir passé avec succès tous les autres tests. Ce *hook* est appelé pour chacun des certificats présents dans la chaîne ;
- → **OPENVPN_PLUGIN_AUTH_USER_PASS_VERIFY** : lancé sur le serveur quand un nouveau client se connecte, en phase de vérification optionnelle de *login*/*password* ;
- → **OPENVPN_PLUGIN_CLIENT_CONNECT** : sur le serveur quand un client se connecte ;
- → **OPENVPN_PLUGIN_CLIENT_DISCONNECT** : sur le serveur quand un client se déconnecte ;
- → **OPENVPN_PLUGIN_LEARN_ADDRESS** : sur le serveur quand un client se voit affecter, modifier ou supprimer son adresse IP dans le tunnel ;
- → **OPENVPN_PLUGIN_TLS_FINAL** : appelé en phase finale de négociation TLS, après **TLS_VERIFY** et **AUTH_USER_PASS_VERIFY** ;
- → **OPENVPN_PLUGIN_ENABLE_PF** : permet d'utiliser le petit firewall intégré dans OpenVPN.

Pour faire appeler un *plugin* à chaque connexion et déconnexion d'un client, il suffit de positionner **type_mask** ainsi :

Fichier

```
type_mask = OPENVPN_PLUGIN_MASK(OPENVPN_PLUGIN_CLIENT_CONNECT) | OPENVPN_
PLUGIN_MASK(OPENVPN_PLUGIN_CLIENT_DISCONNECT)
```

- ⇨ **argv[]** regroupe la liste des arguments passés à la bibliothèque dans le fichier de configuration d'OpenVPN. et **argv[0]** contient le chemin complet d'accès au **.so** comme nous en avons l'habitude en C ;
- ⇨ **envp[]** est un tableau de variables d'environnement maintenues par OpenVPN sous la forme **nom=valeur**. Pour des raisons de sécurité, ces variables ne sont pas accessibles en temps normal dans l'environnement de fonctionnement d'OpenVPN ;
- ⇨ **return_list** permet de renvoyer des données à OpenVPN en cas de besoin. Le type **openvpn_plugin_string_list** est une structure définie ainsi :

Fichier

```
struct openvpn_plugin_string_list
{
  struct openvpn_plugin_string_list *next;
  char *name;
  char *value;
};
```

Il s'agit d'une liste simplement chaînée se comportant comme un dictionnaire basique.

Cette fonction retourne un **openvpn_plugin_handle_t** qui est en fait un **void *** et qui permet de faire passer des informations d'une fonction à l'autre comme on le verra. Si la valeur renvoyée est **NULL**, ce sera considéré comme un échec et stoppera le démarrage d'OpenVPN.

Cette fonction est appelée avant tout autre et son rôle est d'initialiser le *plugin*. L'état des *plugins* sera préservé si OpenVPN reçoit le signal de restart « soft » via **SIGUSR1**, mais pas s'il est redémarré via **SIGHUP**. Dans ce dernier cas, le plugin est fermé et réouvert.

1.1.2 openvpn_plugin_func_v2

Fonction également obligatoire.

```
int openvpn_plugin_func_v2(openvpn_plugin_handle_t handle, const int type,
const char *argv[], const char *envp[], void *per_client_context, struct
openvpn_plugin_string_list **return_list);
```

- ⇨ **handle** est la valeur (en général un pointeur) renvoyée par la fonction **openvpn_plugin_open_v2**. C'est le moyen de faire « circuler » des données d'une fonction à une autre ;
- ⇨ **type** : comme précédemment il s'agit d'un des types **PLUGIN_x** ;
- ⇨ **argv[]** : comme précédemment ;
- ⇨ **envp[]** : comme précédemment ;
- ⇨ **per_client_context** : le pointeur renvoyé par **openvpn_plugin_client_constructor_v1** si cette dernière a été utilisée ;
- ⇨ **return_list** : comme précédemment.

Dans 90 % des cas, cette fonction renvoie soit **OPENVPN_PLUGIN_FUNC_SUCCESS** si tout va bien, soit **OPENVPN_PLUGIN_FUNC_ERROR** en cas d'erreur.

Si la fonction est appelée sur un événement de type **OPENVPN_PLUGIN_AUTH_USER_PASS_VERIFY**, elle peut aussi renvoyer la valeur **OPENVPN_PLUGIN_FUNC_DEFERRED**. Ceci active l'authentification asynchrone qui permet au *plugin* de signaler le succès ou l'échec de l'opération d'authentification quelques secondes après le retour d'**openvpn_plugin_func_v2** en mettant dans le fichier défini par la directive **auth_control_file** – à renseigner dans la configuration d'OpenVPN – les valeurs **0** pour un échec et **1** pour un succès. OpenVPN supprimera ce fichier automatiquement dès qu'il n'en aura plus besoin.

Si l'événement déclencheur est de type **OPENVPN_PLUGIN_ENABLE_PF** et que la fonction ne retourne pas en erreur, le filtrage de paquet sera activé pour le client concerné. Cette notion de filtrage dépasse le cadre de cet article et est traitée plus en détail ailleurs dans ce hors-série.

1.1.3 openvpn_plugin_close_v1

Fonction obligatoire elle aussi.

```
void  openvpn_plugin_close_v1(openvpn_plugin_handle_t handle);
```

- ⇨ **handle** : comme précédemment.

Cette fonction est appelée immédiatement avant le déchargement du *plugin* et est destinée à faire le ménage, principalement concernant la gestion de la mémoire et des éventuels fichiers encore ouverts.

1.1.4 openvpn_plugin_abort_v1

Fonction optionnelle.

```
void openvpn_plugin_abort_v1(openvpn_plugin_handle_t handle);
```

- ⇨ **handle** : comme précédemment.

Cette fonction est appelée en cas d'erreur fatale, mais uniquement si **openvpn_plugin_open** s'est terminée avec succès.

1.1.5 openvpn_plugin_client_constructor_v1

Fonction optionnelle.

```
void *openvpn_plugin_client_constructor_v1(openvpn_plugin_handle_t handle);
```

- ⇨ **handle** : comme précédemment.

Cette fonction est utilisée pour créer des contextes de mémoire client par client qui sont passés à la fonction `openvpn_plugin_func_v2`. Elle est censée allouer la mémoire désirée et retourner le pointeur ainsi défini ou `NULL` si aucune réservation mémoire n'est nécessaire. Elle est appelée à chaque création d'un client avant même qu'il ne soit authentifié.

1.1.6 openvpn_plugin_client_destructor_v1

Fonction optionnelle.

```
void *openvpn_plugin_client_destructor_v1(openvpn_plugin_handle_t handle,
void *per_client_context);
```

⇨ `handle` : comme précédemment ;

⇨ `per_client_context` : contexte renvoyé par `openvpn_plugin_client_constructor_v1` s'il existe.

Cette fonction est appelée lors de la destruction d'un client.

1.1.7 openvpn_plugin_select_initialization_point_v1

Fonction optionnelle.

```
int openvpn_plugin_select_initialization_point_v1(void);
```

Cette fonction va permettre de décaler la phase d'initialisation du *plugin* à l'un des quatre instants possibles en renvoyant l'une des valeurs suivantes :

⇨ `OPENVPN_PLUGIN_INIT_PRE_CONFIG_PARSE` : pour que le *plugin* soit initialisé avant la lecture de la configuration afin de pouvoir renvoyer des paramètres de configuration ;

⇨ `OPENVPN_PLUGIN_INIT_PRE_DAEMON` : valeur par défaut, avant de passer en arrière-plan ;

⇨ `OPENVPN_PLUGIN_INIT_POST_DAEMON` : après être passé en arrière-plan ;

⇨ `OPENVPN_PLUGIN_INIT_POST_UID_CHANGE` : après avoir changé l'utilisateur qui exécute OpenVPN.

1.1.8 openvpn_plugin_min_version_required_v1

Fonction optionnelle.

```
int openvpn_plugin_min_version_required_v1(void);
```

Retourne la version minimale de l'interface pour que le *plugin* fonctionne.

1.2 Séquence de fonctionnement

La séquence générale d'appel possible en fonctionnement normal est la suivante :

Démarrage initial du serveur :

`openvpn_plugin_open_v1`

`openvpn_plugin_client_constructor_v1` (création du « template » client générique)

`openvpn_plugin_func_v1 OPENVPN_PLUGIN_UP`

`openvpn_plugin_func_v1 OPENVPN_PLUGIN_ROUTE_UP`

Connexion d'un nouveau client :

`openvpn_plugin_client_constructor_v1`

`openvpn_plugin_func_v1 OPENVPN_PLUGIN_ENABLE_PF`

`openvpn_plugin_func_v1 OPENVPN_PLUGIN_TLS_VERIFY` (appelé une fois par certificat présent dans la chaîne)

`openvpn_plugin_func_v1 OPENVPN_PLUGIN_AUTH_USER_PASS_VERIFY`

`openvpn_plugin_func_v1 OPENVPN_PLUGIN_TLS_FINAL`

`openvpn_plugin_func_v1 OPENVPN_PLUGIN_IPCHANGE`

[Si `OPENVPN_PLUGIN_AUTH_USER_PASS_VERIFY` a retourné `OPENVPN_PLUGIN_FUNC_DEFERRED`, OpenVPN attend ici que l'authentification soit validée (ou non) par le processus lié à `auth_control_file`]

`openvpn_plugin_func_v1 OPENVPN_PLUGIN_CLIENT_CONNECT_V2`

`openvpn_plugin_func_v1 OPENVPN_PLUGIN_LEARN_ADDRESS`

La session client est créée...

Pour chaque « TLS soft reset » :

`openvpn_plugin_func_v1 OPENVPN_PLUGIN_ENABLE_PF`

`openvpn_plugin_func_v1 OPENVPN_PLUGIN_TLS_VERIFY` (appelé une fois par certificat présent dans la chaîne)

`openvpn_plugin_func_v1 OPENVPN_PLUGIN_AUTH_USER_PASS_VERIFY`

`openvpn_plugin_func_v1 OPENVPN_PLUGIN_TLS_FINAL`

[Si `OPENVPN_PLUGIN_AUTH_USER_PASS_VERIFY` retourne `OPENVPN_PLUGIN_FUNC_DEFERRED`, OpenVPN s'attend à ce que l'authentification soit vérifiée via `auth_control_file` dans l'intervalle de temps défini par l'option `hand-window`. Durant cette période, le trafic ne sera pas interrompu sur le canal de données.]

La session client se poursuit...

...

La session client se ferme.

`openvpn_plugin_func_v1 OPENVPN_PLUGIN_CLIENT_DISCONNECT`

`openvpn_plugin_client_destructor_v1`

Arrêt du serveur :

`openvpn_plugin_func_v1 OPENVPN_PLUGIN_DOWN`

`openvpn_plugin_client_destructor_v1` (pour le client « générique »)

`openvpn_plugin_close_v1`

Nous avons donc ici toute la chronologie de fonctionnement d'un serveur OpenVPN avec les divers événements qui peuvent y être associés. Cela nous permet de choisir assez facilement le ou les événements que nous voulons traiter dans notre *plugin*.

2. UN EXEMPLE

Après cette litanie théorique assez peu digeste, il est temps de passer à la pratique en réalisant un *plugin* qui va tout simplement aller, lors de la phase de démarrage, vérifier le contenu d'un fichier quelque part sur le système de fichiers. Cela pourrait sembler sans intérêt sauf si ledit fichier est sur une clé USB, dans ce cas la possession de cette clé est obligatoire pour se connecter.

Nous allons supposer que la clé USB se monte systématiquement sur `/media/cedric/VPNUSB`, ce qui s'obtient en affectant le label `VPNUSB` à la clé utilisée une fois démontée :

Terminal

```
# fatlabel /dev/sdd1 VPNUSB
```

Commençons par regarder comment notre *plugin* sera déclaré dans la configuration d'OpenVPN sur les postes clients :

Fichier

```
...
plugin /usr/local/bin/testplugin.so /media/cedric/VPNUSB/verify.ovpn
...
```

Cette directive `plugin` prend en argument le chemin complet ou relatif vers la bibliothèque représentant le *plugin* et ensuite tous les paramètres qui seront passés au dit plugin via le tableau `argv`.

Avec ceci nous pouvons définir simplement la fonction `openvpn_plugin_open_v2` :

Fichier

```
 53 OPENVPN_EXPORT openvpn_plugin_handle_t
 54 openvpn_plugin_open_v2(unsigned int *type_mask, const char *argv[],
const char *envp[], struct openvpn_plugin_string_list **return_list)
 55 {
 56     char *filename = NULL;
 57
 58     if(argv[1])
 59         filename = strdup(argv[1]);
 60
 61     // Check if file exists
 62     if(access(filename, F_OK) == -1)
 63         filename = NULL;
 64
 65     *type_mask = OPENVPN_PLUGIN_MASK(OPENVPN_PLUGIN_UP);
 66
 67     return (openvpn_plugin_handle_t) filename;
 68 }
```

Nous voyons tout de suite qu'il faut préfixer nos fonctions à exporter vers OpenVPN par `OPENVPN_EXPORT`, mais ensuite rien de très sorcier. Nous récupérons ensuite en ligne 58 le nom du fichier à tester et s'il n'existe pas, nous renvoyons `NULL` dans le handle, ce qui aura pour effet d'empêcher directement OpenVPN de démarrer. Il faudra donc brancher la clé avant de lancer la connexion, mais cela semble logique, car tout le *plugin* sera exécuté au démarrage.

En ligne 65, nous interceptons uniquement l'événement « démarrage » et enfin nous renvoyons le *handle* qui pointe sur le nom de fichier.

La fonction `openvpn_plugin_func_v2` reste elle aussi d'une simplicité exemplaire :

Fichier

```
 36 OPENVPN_EXPORT int
 37 openvpn_plugin_func_v2(openvpn_plugin_handle_t handle, const int
type, const char *argv[], const char *envp[], void *per_client_context,
struct openvpn_plugin_string_list **return_list)
 38 {
 39     char *filename;
 40
 41     filename = (char *)handle;
 42
 43     if (type == OPENVPN_PLUGIN_UP)
 44     {
 45         return file_verify(filename, argv, envp);
 46     }
 47     else
 48     {
 49         return OPENVPN_PLUGIN_FUNC_ERROR;
 50     }
 51 }
```

Nous récupérons via le handle le nom du fichier dont nous devons vérifier le contenu (ligne 41) et si l'événement est le bon, nous appelons la fonction `file_verify` qui va traiter l'affaire, sinon nous sortons en erreur.

La fonction qui fait presque tout le travail pourrait constituer un exemple à mettre dans le manuel du grand débutant en C :

Fichier

```
 8 static int file_verify(char *filename, const char *args[], const char
*envp[])
 9 {
10     FILE *fd;
11     char login[80];
12     char password[80];
13     const char *user, *pass;
14
15     if((fd = fopen(filename, "r")) == NULL)
16         return OPENVPN_PLUGIN_FUNC_ERROR;
17
18     if((fgets(login, 80, fd)) == NULL)
19         return OPENVPN_PLUGIN_FUNC_ERROR;
20     if((fgets(password, 80, fd)) == NULL)
21          return OPENVPN_PLUGIN_FUNC_ERROR;
22
23      fclose(fd);
24
25     if((strcmp(login, "Utilisateur\n")) == 0 && (strcmp(password, "Mot_
de_Passe\n") == 0))
26     {
27         return OPENVPN_PLUGIN_FUNC_SUCCESS;
28     }
29     else
30     {
31         return OPENVPN_PLUGIN_FUNC_ERROR;
32     }
33 }
```

Il s'agit là d'un simple traitement de fichier et de chaînes de caractères comme on en connaît des milliers. On part du principe que le fichier valide est constitué de deux lignes, une avec « Utilisateur » la seconde avec « Mot_de_Passe » et si ce n'est pas ça, on renvoie une erreur.

Maintenant, jetons un coup d'œil à la fonction de fermeture du *plugin* :

Fichier

```
70 OPENVPN_EXPORT void
71 openvpn_plugin_close_v1(openvpn_plugin_handle_t handle)
72 {
73     char *filename;
74
75     filename = (char *)handle;
76     free(filename);
```

On se contente de libérer l'espace mémoire utilisé par le *handle*, espace alloué par `strdup` dans la fonction de démarrage.

Bien entendu, il ne faut pas oublier les *includes* utiles :

Fichier

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 #include "openvpn-plugin.h"
```

Le fichier `openvpn-plugin.h` est à récupérer dans `/usr/include/openvpn`, du moins sur une Debian.

Le binaire à générer étant une bibliothèque partagée, voici le `Makefile` qui va bien :

Fichier

```
 1 PREFIX=/etc/openvpn
 2 CFLAGS += -DPREFIX='"$(PREFIX)"'
 3
 4 all: test_plugin.so
 5
 6 test_plugin.o: test_plugin.c
 7     $(CC) $(CFLAGS) -fPIC -c test_plugin.c
 8
 9 test_plugin.so: test_plugin.o
10     $(CC) -fPIC -shared -Wl,-soname,test_plugin.so -o test_plugin.so
test_plugin.o -lc
11
12 install: test_plugin.so
13     mkdir -p $(DESTDIR)$(PREFIX)
14     install -c test_plugin.so -m 755 $(DESTDIR)$(PREFIX)
15
16 uninstall:
17     rm -f $(DESTDIR)$(PREFIX)/test_plugin.so
18
19 clean:
20     rm -f *.so *.o
```

Ce `Makefile` nous générera un `.so` en fonction de l'architecture de votre station. Si vous êtes en 64 bits et que vos clients sont encore en 32 bits, il suffit de rajouter l'option `-m32` aux lignes 7 et 10 après avoir installé le kit de développement 32 bits bien entendu.

Et voilà un petit *plugin* sans aucune prétention à utiliser côté client. La structure est exactement la même si on veut travailler côté serveur, il faut juste penser à bien choisir les événements auxquels on veut répondre, certains n'étant pas utilisables sur un client. La suite est en fonction de vos besoins et de votre imagination. Sous Debian, vous trouverez deux *plugins* dans `/usr/lib/openvpn` :

- `openvpn-plugin-auth-pam.so` qui permet d'authentifier un utilisateur via PAM ;
- `openvpn-plugin-down-root.so` qui permet de retrouver les droits root pour exécuter un script lors de l'arrêt du serveur. En effet, OpenVPN effectue une diminution de privilèges qui fait que lorsque le serveur s'arrête, il tourne encore sous un compte utilisateur sans droits spécifiques. Ce *plugin* permet de pallier à cela.

CONCLUSION

Comme on vient de le voir, l'écriture d'un *plugin* pour OpenVPN est d'une simplicité à toute épreuve. Une chose à savoir tout de même, quand un *plugin* et un script interceptent le même événement, le *plugin* est exécuté en premier et, s'il ne retourne pas d'erreur, le script en second. Maintenant, il me reste à vous souhaiter bon codage :) ▉

VISITEZ NOTRE BOUTIQUE ET DÉCOUVREZ NOS GUIDES !

RENDEZ-VOUS SUR **www.ed-diamond.com**

POUR DÉCOUVRIR TOUTES LES GUIDES DE VOS MAGAZINES PRÉFÉRÉS !

DÉBUTEZ

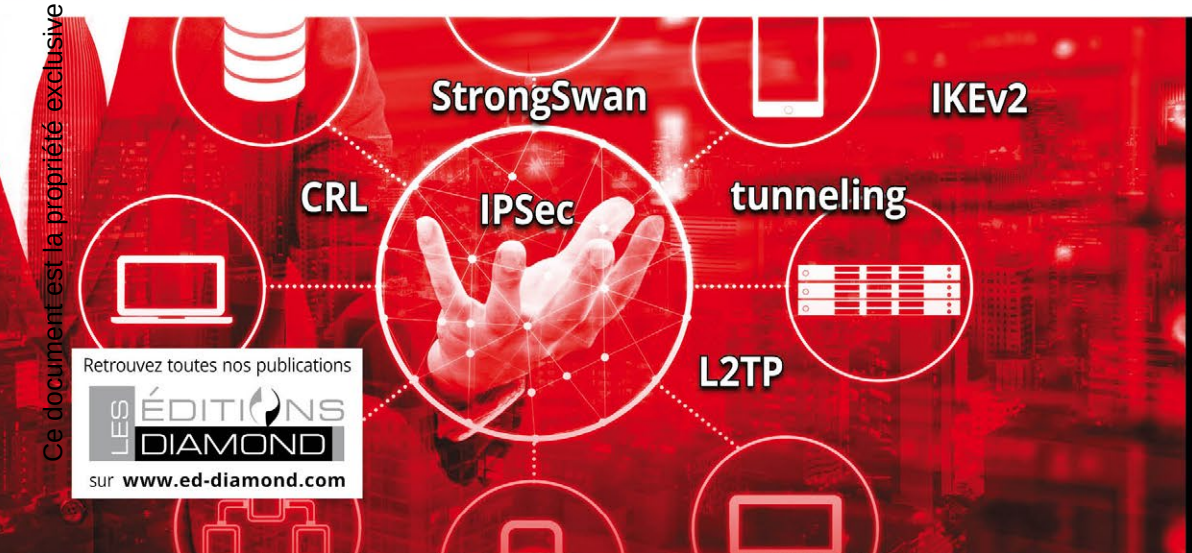
avec le protocole TCP/IP, les nouveautés d'IPv6 et le fonctionnement d'un VPN

OPENVPN

Configurez-le finement et installez des clients y compris sur des systèmes non libres

ALLEZ PLUS LOIN

en développant vos plugins OpenVPN et en y intégrant un firewall



IPSEC

Apprenez à configurer l'« autre » solution de tunneling