Découverte de réseaux IPv6

Nicolas Collignon

HSC - Hervé Schauer Consultants
4 bis, rue de la gare
92300 Levallois-Perret, France
nicolas.collignon@hsc.fr

Résumé
Le protocole IPv6 a été conçu il y a déjà plus de dix ans. Il est pourtant très faiblement présent sur les réseaux d'entreprise. Le monde opérateur et l'Asie poussent progressivement à l'adoption d'IPv6 par manque d'adresses. De nombreux chercheurs attendent impatiemment le *label* de *Microsoft* pour certifier les *modems routeurs* qui supportent les différentes technologie de cohabitation IPv4 - IPv6 (annoncé au « *French IPv6 Worldwide Summit 2006* »). La technologie se développe mais les outils d'audit et d'attaques la supportent très rarement. Ce constat s'explique notamment par le fait que les méthodes de *scan* doivent être implémentées différemment qu'avec IPv4.

Ce document présente un rappel sur les concepts de découverte réseau liés au protocole IPv6 et introduit l'architecture et l'implémentation du *framework sherlock*, un outil permettant entre autres de réaliser des *scans* IPv6.

1 Introduction

L'espace d'adressage IPv6 est nettement plus vaste que son homologue IPv4. Le point important à noter est le fait que la taille des réseaux alloués aux sociétés est généralement un /28 ou un /48. L'époque où l'on se battait pour obtenir une seule adresse publique est révolue, c'est désormais plusieurs /64 qui sont alloués pour un seul équipement réseau tel un téléphone portable. Les méthodes traditionnelles de *scan* réseau étant mises en échec par la taille de l'espace d'adressage, il est souvent considéré qu'un *Worm* ne pourrait pas se propager aussi rapidement sur un réseau IPv6 que sur un réseau IPv4. La découverte d'un réseau IPv6 nécessite une approche différente de celle d'un réseau IPv4. La taille de l'espace d'adressage rend quasi-impossible une approche linéaire du *scan* de machines. Ce document a pour but d'introduire les différents mécanismes utilisables pour découvrir la cartographie d'un réseau IPv6 et de présenter la suite d'outils *sherlock*. Le *framework sherlock* s'attaque à la problématique de traiter une tâche coûteuse sur une architecture répartie. Ce document présente un bref rappel des méthodes de *scan* IPv6, suivi d'une présentation des implémentations de ces méthodes ainsi que certaines optimisations.

2 Concepts

Ce chapitre introduit les trois différents types de *scans* réalisables avec le protocole IPv6. C'est-à-dire comment *scanner* le réseau local, comment *scanner* un réseau distant et mieux, comment *scanner* un réseau local à distance ?

2.1 Scan sur le lien

Étant donné qu'il est nécessaire de posséder l'adresse de couche OSI 2 pour contacter une adresse IP (IPv4 ou IPv6) sur un lien réseau (réseau local), la découverte d'adresses sur le lien se résume souvent à émettre des requêtes de traduction d'adresses couche OSI 2 / couche OSI 3.

Avec IPv4, des messages ARP [1] *who-has* sont envoyés sur le lien réseau afin de localiser toutes les adresses IPv4 utilisées. Les concepts de traduction d'adresses couche OSI 2 / couche OSI 3 ont été unifiés avec le protocole IPv6. Le protocole générique *Neighbor Discovery* (NDP [2]) basé sur ICMPv6 est désormais utilisé à la place d'un protocole spécifique à chaque type de transmission (ex : ARP pour les réseaux *Ethernet*). Un *scan* sur le lien peut être réalisé en envoyant des messages *Neighbor Solicitation* à destination de l'adresse *Multicast all-nodes* (`ff02::1`).

Les adresses *Link-Local*[3] sont très fréquemment calculées à partir d'un identifiant unique et spécifique à l'équipement réseau utilisé. L'identifiant d'interface de ces adresses (suffixe de 64 bits) peut être utilisé pour des adresses d'autres périmètres. Sur la majorité des implémentations IPv6, la configuration par défaut associe à chaque carte réseau *Ethernet* un identifiant de 64 bits dérivé de l'adresse MAC. Ces identifiants sont réutilisés pour calculer les adresses de périmètre lien mais aussi les adresses globales. Trouver une adresse *Link-Local* utilisée sur un réseau permet souvent de découvrir une adresse globale associée au même équipement réseau.

Example 1.

`fe80::` **`215:58ff:fe7c:274`**

⇓

`2001:aaa:bbb::/48`

⇓

`2001:aaa:bbb::` **`215:58ff:fe7c:274`**

Une approche passive de la découverte d'hôtes sur le lien passe nécessairement par l'écoute des messages *Neighbor Solicitation* envoyés sur le réseau. Ces messages sont généralement envoyés par les hôtes avant de s'approprier une adresse afin de vérifier qu'elle n'est pas déjà associée à un équipement réseau.

2.2 Scan distant

Nous entendons par *scan* « distant », un *scan* réseau à destination d'adresses IPv6 *Unicast* globales, c'est-à-dire un *scan* d'adresses *routables* sur Internet IPv6. Il y a techniquement très peu de différences entre un *scan* IPv4 et un *scan* IPv6 hormis l'espace d'adressage et évidemment, la structure de l'en-tête IP. Généralement le protocole ICMP [4] [5] est utilisé pour tester la validité d'une adresse IP, cela avec IPv4 ou IPv6.

Les problématiques de cartographie réseau liées au filtrage sont les mêmes avec IPv6 et IPv4. D'autres protocoles comme TCP, UDP et SCTP peuvent également être utilisés comme support de *scan* pour la découverte d'hôtes ou la découverte de services.

[1] RFC 826 : « *An Ethernet Address Resolution Protocol* »
[2] RFC 2461 : « *Neighbor Discovery for IP Version 6* »
[3] Adresses utilisées sur le réseau local, de périmètre *lien*
[4] RFC 792 : « *Internet Control Message Protocol (ICMP)* »
[5] RFC 2463 : « *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification* »

2.3 Scan de lien distant

Le *scan* de lien distant est une méthode permettant de découvrir le plan d'adressage privé utilisé sur un réseau distant. Avec le protocole IPv4, il est par exemple possible dans certains cas, de détecter la présence d'adresses privées à travers des relais applicatifs mal configurés (ex : *Web Proxy*). Avec IPv6, on ne s'intéresse plus vraiment à la notion d'adresses publiques ou privées mais à la notion plus générique de périmètre des adresses. *Scanner* un lien distant IPv6 revient en quelque sorte à tester la présence d'adresses d'un périmètre inférieur à global (lien, site, organisation) à travers des adresses globales : les champs « *adresse source* » et « *adresse de destination* » appartiennent au réseau `2000::/3`.

Certaines implémentations du protocole IPv6 acceptent de traiter les paquets utilisant un *Routing Header* (RH [6] [7]) contenant une liste de routeurs de périmètre variable. C'est à dire qu'il est parfois possible d'utiliser des adresses IPv6 globales et des adresses de périmètre lien dans un paquet. Les paquets envoyés ne donneront pas suite à une réponse car la pile IPv6 qui traitera les requêtes refusera de forger un paquet mêlant des adresses de périmètres différents. Si obtenir une réponse n'est pas possible, on peut en revanche détecter la présence d'adresses de périmètres différents. Pour cela, il suffit que l'adresse source et l'adresse de destination du paquet soient de périmètres identiques au niveau de l'hôte qui doit générer le paquet réponse.

Example 2. Pour contacter C de périmètre X via A et B de périmètres Y, l'en-tête doit contenir :

Adresse IP source : A
Adresse IP destination : B
Routeurs intermédiaires (RH type 0) : B, C, B

Ce mécanisme peut également être utilisé pour effectuer une prise d'empreinte sur l'implémentation du protocole IPv6 d'un hôte. À titre d'exemple, l'implémentation de *Linux*, accepte par défaut le routage de paquets à destination de l'adresse *localhost* (`::1`). Bien que l'évasion de périmètre semble peu intéressante si on ne peut pas recevoir de réponse, elle illustre les effets de bord des en-têtes de routage.

3 Optimisations

Il n'est plus possible de tester linéairement la validité de chaque adresse présente dans l'espace d'adressage avec le protocole IPv6. Il est nécessaire d'aller droit au but, de *scanner* le réseau avec une certaine logique afin d'obtenir le plus rapidement possible des résultats. Ce chapitre présente des méthodes permettant d'optimiser un *scan* et de trouver des sources d'informations utiles pour la découverte de réseaux IPv6.

Étant donnée la diversité des vecteurs d'optimisation, une implémentation d'un *scanner* de réseau IPv6 doit forcement passer par une parallélisation des tâches et une intelligence qui ajuste le processus de *scan* en fonction des informations recueillies.

[6] RFC 2460 : « *Internet Protocol Version 6* » → « 4.4 Routing Header »
[7] IPv6 : Les dangers des « Routing Headers Type 0 »
`http://www.hsc.fr/ressources/breves/ipv6-rt0.html.fr`

3.1 DNS Mapping

Le service DNS (*Domain Name System*) est souvent une précieuse source d'informations sur un réseau. Il est parfois possible de récupérer une bonne partie de l'espace d'adressage d'un réseau avec une simple requête de transfert de zone DNS (requête AXFR [8]). Même si c'est rarement le cas lorsque l'on interroge un serveur DNS à travers Internet, les serveurs DNS répondent souvent aux requêtes AXFR si l'émetteur vient du réseau de l'entreprise. Le serveur DNS est donc un outil très important pour une découverte de réseau IPv6 effectuée en internet. Le service DNS est souvent plus sollicité sur un réseau IPv6 que sur un réseau IPv4 car les adresses IPv6 sont difficiles à manipuler : longues à écrire, non-intuitives et moins facilement mémorisables.

Dans tous les cas, il est possible de faire une attaque de type *bruteforce* sur les noms d'hôtes afin d'essayer de trouver des adresses IP. L'attaque consiste généralement à essayer de traduire une liste de noms d'hôtes issue d'un dictionnaire. Plusieurs outils disponibles sur Internet permettent de réaliser cette opération ([2], [3], [4] ...). Il est intéressant de procéder à une recherche de motifs pour chaque nom d'hôte découvert pendant un *scan*.

Example 3. Le nom d'hôte contient parfois des indications concernant d'autres noms probablement valides :

srv-**1** → srv-**2**, srv-**3**, srv-**4** ...
web → www, ftp, proxy ...

Example 4. Certaines portions des noms d'hôte sont parfois dérivées de leur adresse IP :

srv-**2-3** → `2001:aaa:bbb::/48` → `2001:aaa:bbb::2:3`

L'utilisation du service DNS dans la découverte de réseau IPv6 ne se limite pas à la traduction d'adresses. La traduction inverse (*Reverse* DNS) est également utile : les fournisseurs d'accès Internet configurent souvent la traduction inverse de manière à ce que le nom d'hôte d'un client soit calculé à partir de l'adresse IP et de la localisation géographique de son équipement réseau (ex :*Modem*).

L'utilisation des DNS peut servir à découvrir des noms d'hôtes mais peut également servir à obtenir des informations concernant les services exposés par un hôte. À titre d'exemple, il est fort probable qu'un serveur avec comme nom « www.hsc.biz » soit un serveur Web, et donc, dispose d'un serveur HTTP en écoute sur le port 80/TCP.

3.2 Utilisation des adresses Multicasts

Le concept des adresses *Broadcast* n'existe plus avec le protocole IPv6. Les adresses *Multicast*, plus faciles à maîtriser, les remplacent. L'utilisation des *Multicast* s'est vue généralisée et les services qui les utilisent sont de plus en plus nombreux. L'utilisation des protocoles de gestion des abonnements *Multicast* comme *Multicast Listener Discovery* (MLDv2 [9] [10]) permettent d'apprendre des informations sur le rôle, source ou client, des adresses abonnées à un groupe *Multicast*. La structure des adresses *Multicasts* est généralement dérivée d'adresses globales *Unicast*. On retrouve également des adresses *Unicast* dans les *Rendez-vous Point* (RP [11]) de certaines adresses *Multicasts*.

[8] RFC 1034 : « *Domain Names - concepts and facilities* »
[9] RFC 2710 : « *Multicast Listener Discovery (MLD) for IPv6* »
[10] RFC 3810 : « *Multicast Listener Discovery Version 2 (MLDv2) for IPv6* »
[11] RFC 3956 : « *Embedding the Rendezvous Point (RP) Address in an IPv6 Multicast Address* »

Les groupes *Multicasts* sont la source d'information la plus rapide pour récupérer une liste des hôtes connectés sur un réseau, de façon active ou passive. Ils sont fréquemment utilisés sur le lien avec les adresses de périmètre restreint : lien, site ou organisation. L'outil de *scan* réseau fournit avec la suite [1] utilise les adresses *Multicasts*.

3.3 Réduction de l'espace d'adressage

Une des principale optimisation au *scan* IPv6 est la réduction de l'espace d'adressage à balayer. Les fournisseurs d'accès IPv6 mettent généralement à disposition de leurs clients des réseaux de taille /28 à /48. La réduction de l'espace d'adressage consiste donc à limiter la quantité de bits à tester (128 – 28 = 100 bits pour un réseau /28). Les méthodes pour réduire l'espace d'adressage dépendent de la façon dont ont été générées les adresses : par un procédé automatique ou par un procédé manuel. Les adresses générées par un procédé automatique sont généralement calculées à partir d'un algorithme connu. Les adresses conçues par un procédé manuel suivent une certaine logique « humaine ».

Adressage automatique : 6in4, 6to4, Dual-Stack

L'intégration du protocole IPv6 dans une architecture réseau implique souvent la cohabitation avec IPv4 sur de nombreux segments réseau. Les hôtes possèdent dans ce cas plusieurs adresses IPv4 et IPv6 permettant une utilisation des services réseau indépendamment du protocole utilisé. Les adresses IPv6 utilisées pour les tunnels *6in4*, les passerelles *6to4* ou sur les hôtes *Dual-Stack* sont quasiment toujours calculées à partir d'une adresse IPv4.

Example 5. En connaissant l'adressage IPv4 utilisé sur un réseau distant, il est envisageable de déterminer quelles adresses IPv6 seront associées. On retrouve principalement 4 motifs (avec `xxxx` la représentation hexadécimale d'une adresse IPv4 et `pppp` un préfique réseau de taille variable) :

- Les différents formats d'adresses *6to4* :
  - `2002:xxxx::xxxx`
  - `{ 2002:xxxx::1`
  - `{ 2002:xxxx:xxxx::`
- Les adresses de compatibilité IPv4 :
  - `pppp::xxxx`
  - `{ pppp::ffff:xxxx`

Scanner les adresses IPv6 associées à leurs homologues IPv4 nécessite l'envoi de cinq fois plus de paquets sur le réseau. *Scanner* les adresses IPv6 associées à un réseau IPv4 /24 à une vitesse de 1000 tests par seconde représente un travail de seulement 5 minutes.

Adressage automatique : EUI-64

Scanner l'intégralité d'un réseau /64 avec une vitesse de 1000 tests par seconde demande environ **585 millions d'années**. L'espace d'adressage IPv6 est énorme mais la répartition des adresses dans cet espace est loin d'être linéaire. La proportion des adresses EUI-64 [12] dans un réseau est très

[12] cf. « *Guidelines for 64-bit global identifier (EUI-64) registration authority* »

variable (10 à 90% sur un échantillon de 250 adresses IPv6 collectées sur des réseaux opérationnels), mais il est très rare qu'aucune adresse EUI-64 ne soit utilisée sur un réseau IPv6. EUI-64 est un mécanisme qui génère un suffixe de 64 bits (l'identifiant d'interface) à partir d'un identifiant unique de 48 bits, issu d'un équipement réseau. Le cas le plus fréquent est celui des adresses EUI-64 générées à partir d'une adresse MAC pour une carte réseau *Ethernet*. L'espace d'adressage à tester est donc réduit de 64 bits à 48 bits. Néanmoins, tester 48 bits d'espace d'adressage à raison de 1000 tests par seconde représente tout de même un travail de **9000 années**.

L'identifiant unique d'un équipement réseau (EUI-48) est composé d'un identifiant (OID : *Object IDentifier*) spécifique au constructeur et au modèle de l'équipement. En considérant les équipements les plus déployés sur les réseaux, on retrouve environ 12 fabricants, soit environ 450 OID différents. La durée réelle de parcours de l'espace d'adressage EUI-64 avoisine les **86 jours**. La réduction de l'espace n'étant pas suffisante, en considérant que la marque d'un fabricant d'équipements réseau utilisée sur la cible est connue, le paramètre OID fixé et utilisé pour le calcul EUI-64 ramène l'espace d'adressage à tester à 24 bits. On obtient un *scan* de **4 heures** pour un OID donné à 1000 tests par seconde.

Adressage automatique : Auto-configuration avec état

Les services d'attribution d'adresses dynamiques utilisés avec IPv4 tel que DHCP [13], ont leur équivalent sur les réseaux IPv6. Les serveurs DHCPv6 [14] allouent souvent les adresses de façon linéaire de la même manière que les GGSN (*Gateway GPRS Support Node*) sur les réseaux mobiles. Il est souvent facile de configurer l'espace d'adressage dédié à l'allocation dynamique d'adresses mais il est rarement possible de choisir le procédé utilisé pour générer des adresses. Une implémentation d'un *scanner* IPv6 a tout intérêt à tester la validité des adresses voisines de chaque adresse découverte.

Adressage « manuel »

Le terme adressage « manuel » sous-entend des adresses qui sont attribuées selon un plan d'adressage défini par des êtres humains `:)`. Avec les adresses globales IPv4, il n'y avait pas réellement de problème à ce niveau car la taille des réseaux alloués par les fournisseurs d'accès restait très modeste. Avec IPv6, les 64 bits d'information dédiés à la partie « Identifiant d'interface » laissent plus de liberté concernant le plan d'adressage. Cependant, il est très rare de voir des adresses choisies aléatoirement. L'entropie de la répartition entre des bits à 0 et les bits à 1 est souvent faible.

En prenant en échantillon de 200 adresses IPv6 *Unicast* non formées à partir de la méthode EUI-64, collectées sur des réseaux opérationnels de taille variable (/28 à /48), nous avons réalisé un graphe (cf. Fig. 1) des probabilités d'apparition de bit mis à 1 dans le suffixe (les 96 derniers bits). Ces résultats exposent une réduction de l'espace d'adressage de **36%**. 35 des 96 bits disponibles ont toujours été mis à 0. Les adresses suivent souvent des motifs. La découpe d'une adresse IPv6 en blocs de 16 bits montre que les bits de poids fort et les bits de poids faible sont majoritairement mis à 1. La découpe d'une adresse IPv6 en bloc de 8 bits montre que les bits de poids fort sont très majoritairement mis à 0. Les adresses IPv6 sont notées en hexadécimal mais les habitudes d'IPv4 nous ramènent souvent à utiliser uniquement une notation décimale : les chiffres sont largement plus utilisés que les lettres.

[13] RFC 2131 : « *Dynamic Host Configuration Protocol* »
[14] RFC 3315 : « *Dynamic Host Configuration Protocol for IPv6 (DHCPv6)* »

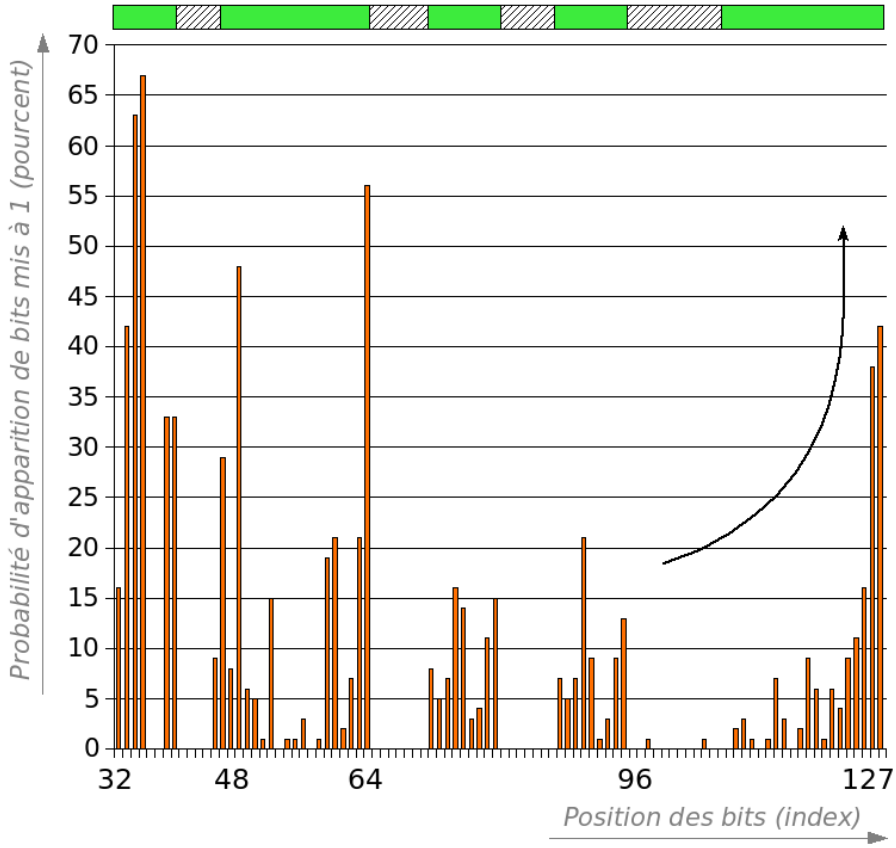

FIG. 1: Probabilité d'apparition de bits mis à 1 dans un suffixe de 96 bits

Les résultats de ce graphe peuvent être utilisés de deux façons complémentaires pour optimiser un *scan* :

- en réduisant l'espace d'adressage de 96 à 61 bits,
- en choisissant judicieusement des adresses parmi l'espace d'adressage réduit, selon des probabilités prédéfinies.

Il devrait être possible d'ajuster les probabilités selon le type d'infrastructure réseau à *scanner*, c'est à dire moduler les valeurs en fonction de la taille du réseau alloué par le fournisseur d'accès (information obtenue avec une requête *whois*).

Une analyse plus complète de l'échantillon d'adresses permet aussi de constater que :

- 40% des adresses possèdent 48 bits consécutifs mis à zéro,
- 50% des adresses possèdent 32 bits consécutifs mis à zéro,
- 10% des adresses se terminent par 8 bits mis à zéro,
- mois de 5% des adresses contiennent des lettres (a-f) en notation hexadécimale.

La proportion d'adresses EUI-64 est très variable suivant les réseaux. Sur certains réseaux où la politique de sécurité tend à limiter au maximum la fuite d'information, on retrouvera très peu

d'adresses EUI-64. À l'inverse, sur les réseaux où la configuration IP des hôtes est plus automatisée, on retrouve une proportion d'adresses EUI-64 de l'ordre de 70 à 80 %. Parmi les motifs récurrents, on retrouves aussi des termes familiers : « *beef* », « *cafe* », « *babe* », etc.. Tous ces constats renforcent l'idée qu'un *scan* IPv6 passe forcement par une sélection judicieuse des adresses à *scanner* afin d'éviter un balayage linéaire de l'espace d'adressage.

3.4 En-têtes de routage

En partant du principe que les hôtes d'un réseau supportent l'extension IPv6 *Routing Header*, il est éventuellement possible d'accélérer un *scan* ICMPv6 ou UDP. Pour un *scan* classique, à chaque paquet envoyé, la validité d'une seule adresse IP est vérifiée. Avec IPv6, en ajoutant un *Routing Header* (RH) de type 0 à chaque paquet envoyé, on peut tester la validité d'une ou plusieurs adresses, en utilisant un mécanisme similaire au *Source Routing*[15] IPv4. La transparence du réseau est un objectif recherché par les créateurs du protocole IPv6, il est donc relativement rare de voir des pare-feu IPv6 filtrant le protocole ICMPv6. Au lieu de tester la présence d'un seul hôte par paquet envoyé, il est possible de tester jusqu'à K adresses par paquet dans le RH si le MTU (*Maximum Transmission Unit*) du réseau permet de stocker les adresses de K relais IPv6 dans l'en-tête IPv6 sans fragmentation. Cette méthode présente plusieurs inconvénients :

- Si les paquets avec entêtes de routage sont filtrés, le *scan* ne produira aucun résultat ;
- La taille des paquets envoyés est supérieure à celle d'un *ping* ICMPv6 classique sans en-tête de routage, le débit d'émission est donc réduit ;
- L'efficacité de l'algorithme est directement liée à la probabilité d'utilisation d'adresses valides dans les entêtes de routage.

Supposons qu'un *scanner* souhaite détecter la présence des hôtes A, B et C sur le réseau. Au lieu de simplement envoyer un *ping* à destination de chaque machine, le *scan* peut être optimisé en rajoutant une liste de routeurs intermédiaires avec une en-tête de routage type 0. Ainsi, si l'hôte A répond, les adresses de A, B et C sont valides. Si un paquet ICMPv6 *Unreachable* est renvoyé, l'adresse de l'émetteur du paquet réponse est le dernier relais valide.

Example 6. Un *ping* vers A, lui même *routé* par R, permettant de détecter la présence de B et C sur le réseau :

- si C répond « *Echo Reply* » $\rightarrow$ A, B et C sont valides
- si B répond « *Unreachable* » $\rightarrow$ A et B sont valides
- si A répond « *Unreachable* » $\rightarrow$ A est valide
- si R répond « *Unreachable* » $\rightarrow$ A est invalide

Une implémentation performante de cette optimisation de *scan* est assez complexe à développer car l'ordre de sélection des adresses est crucial pour obtenir un réel gain en vitesse de découverte d'adresses.

4 Sherlock

Sherlock est un *framework* conçu pour traiter des tâches longues et lourdes en calcul à travers un modèle réparti. L'architecture est conçue pour découper le calcul d'une tâche de complexité infinie

[15] RFC 791 : « *Internet Protocol* »

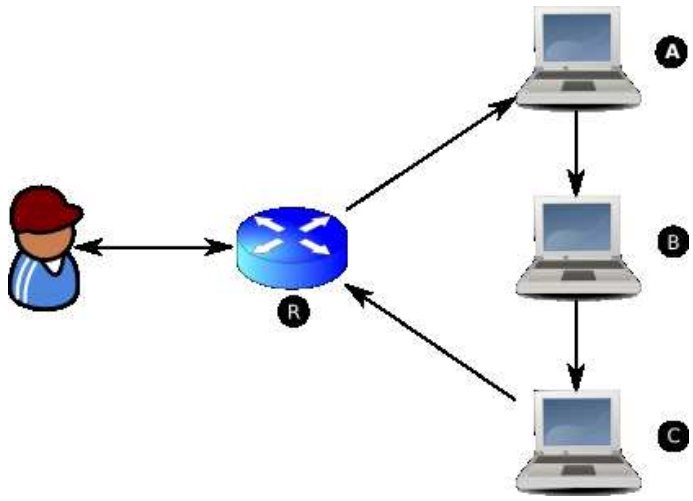

FIG. 2: Trajet des paquets ICMPv6 *Echo Request/Reply* utilisant les RH type 0

en plusieurs tâches de complexité abordable. L'objectif initial lors du développement était de réaliser un *scanner* IPv6 mais l'architecture a été généralisée pour être évolutive vers différents types de ressources autres que le réseau : processeur, mémoire, stockage.. La ressource réseau permettant la découverte de réseau IPv6 est implémentée sous l'appellation *sherlock-net*.

L'objectif principal de l'implémentation de *sherlock-net* n'est pas de *scanner* la totalité d'un réseau IPv6 dans un temps réduit, mais de trouver le plus rapidement possible, le plus grand nombre d'hôtes connectés sur un réseau IPv6. C'est-à-dire en évitant de procéder à un balayage séquentiel des adresses dans l'espace d'adressage. Le processus de découverte d'adresses IP ne se termine presque jamais (sauf si explicitement demandé). Le but est que les informations découvertes par le *scan* alimentent elles-même le moteur de *scan*. *Scanner* la totalité d'un réseau IPv6 est irréaliste dans la majorité des cas mais découvrir 90% des adresses utilisées sur un réseau semble être une tâche abordable.

Il est important de préciser que le *framework* n'a en aucun cas été conçu pour former des *botnets*. Réaliser plusieurs calculs en parallèle n'implique par forcément la distribution de la charge sur plusieurs machines. Cependant pour des facilités d'implémentations (*processes vs. threads*) et étant donné l'orientation réseau du projet, la répartition des tâches peut être effectuée sur plusieurs machines distinctes.

Ce chapitre a pour but d'introduire l'architecture et l'implémentation du *framework sherlock* et de la ressource *sherlock-net* associée à la découverte de réseau IPv6.

4.1 Architecture

Modèle réparti

L'architecture de *sherlock* fonctionne autour d'un modèle réparti (N → 1) où un serveur attend des connexions provenant des différents clients. Le serveur a trois rôles principaux :

- traiter les requêtes des clients,
- enregistrer et propager les évènements notifiés,

– gérer une base de donnée d'informations spécifiques aux ressources enregistrées.

L'intelligence du modèle est totalement déportée chez les clients. Les différents clients enregistrent auprès du serveur un masque d'évènements à surveiller afin d'être tenus au courant du changement d'état d'une tâche. Cela permet également de connecter des automates (*bots*) au *framework* qui vont interpréter les résultats et agir en conséquence. Trois types de clients sont distingués : les clients finaux type interface utilisateur (*ui*), les robots (*bots*) et les travailleurs (*workers*).

Gestion des tâches

Pour chaque ressource définie dans le *framework sherlock*, les *workers* peuvent définir une limite de montée en charge au niveau de la quantité de travail ou au niveau de la durée nécessaire pour réaliser une tâche. La ressource *sherlock-net* permet par exemple d'imposer une limite en terme de débit de transmission réseau (paquets ou octets par secondes) et une limite sur la durée d'un *scan* réseau. Lorsqu'une tâche est affectée à un *worker*, le serveur associe un délai de réalisation à celle-ci afin de pouvoir libérer la réservation si un *worker* est déconnecté du réseau pendant une période prolongée.

La majorité des quantités traitées au niveau de la gestion des tâches dans le *framework sherlock* sont manipulées avec des grands nombres au sens informatique du terme. Les limites des entiers 32 bits ou 64 bits sont outrepassées avec des fonctions permettant le calcul sur des quantités variables. Le modèle est conçu pour réaliser des tâches longues en traitement. Il est donc assez évolutif concernant la gestion de tâches fastidieuses comme le *scan* d'un réseau IPv6 de taille /28. Un réseau de cette taille peut contenir jusqu'à 2^{100} adresses IP (2^{128-28}). Cette quantité ne peut pas être représentée par un entier 32 ou 64 bits car 100 bits d'informations sont utilisés.

La répartition de la charge de travail est déterminée par un ordonnanceur. À chaque tâche peut être affectée un ordonnanceur différent, actuellement : *fair_work* (charge équitable), *fastest* (rapidité maximale) ou *random* (répartition aléatoire).

Architecture « sherlock-net »

L'architecture de l'implémentation de la ressource *sherlock-net* est organisée autour de 5 acteurs :

– le cœur, *sherlockd*, couplé à la ressource *sherlock-net* ;
– un client d'administration, *sherlock-sh* ;
– une sonde (*sniffer*), *sherlock-sniff* ;
– l'émetteur pour lancer les *scans* réseaux, *sherlock-send* ;
– des automates (*bots*) pour accélérer et automatiser les *scans*.

L'objectif initial est de mettre en place une architecture répartie mais il est tout à fait possible d'utiliser le *framework* dans un mode indépendant (*standalone*), permettant une utilisation similaire à celle des outils de découverte réseau disponibles sur la toile ([5], [6], etc..). L'intérêt de cette architecture est la possibilité d'utiliser plusieurs émetteurs pour lancer de multiples *scans* en parallèles et de *sniffer* différents points de routage des paquets. Il est également possible de développer des automates de *scans* indépendants sans toucher au cœur du programme.

Les sondes sont généralement placées sur des segments réseau communs aux émetteurs mais ce n'est pas une obligation. Les sondes peuvent être placées en amont sur le trajet des paquets. Il est également possible de *scanner* un hôte avec les *Routing Headers* afin de faire en sorte que la machine à *scanner* ne soit pas le destinataire final des paquets mais un routeur intermédiaire. Sur

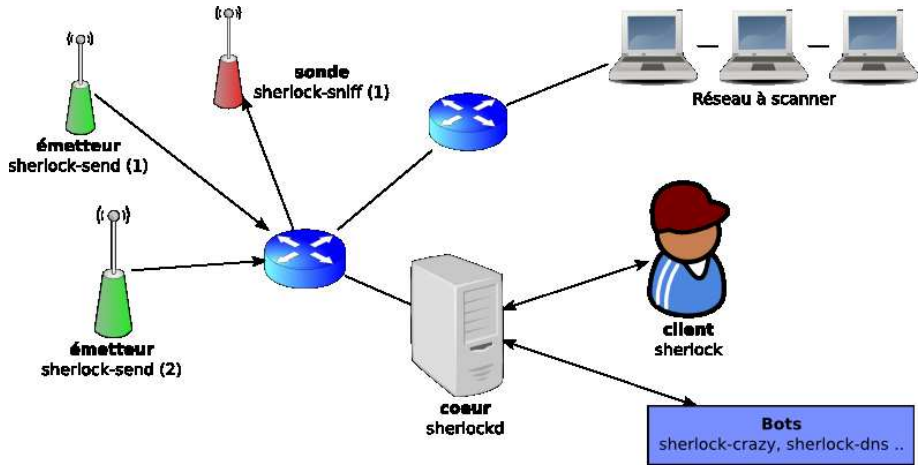

FIG. 3: Architecture de *sherlock-net*

les rares réseaux où le filtrage *anti-spoofing*[16] n'est pas encore mis en place, les émetteurs peuvent simplement usurper les adresses des sondes.

Le concept derrière le fonctionnement de la ressource *sherlock-net* est que l'utilisateur fournit une liste d'indices sur un réseau : nom d'hôtes, adresses IPv6 et/ou IPv4. La base de connaissances du réseau s'agrandit. L'utilisateur peut par la suite, lancer manuellement des *scans* ou connecter un *bot* à *sherlock* afin que la découverte du réseau soit plus ou moins automatisée.

4.2 Implémentation

La majorité des codes sources contenus dans le *framework sherlock* a été développée avec le langage C par choix personnel et principalement parce qu'avoir une base en C permet le développement de modules dans des langages de plus haut niveau par la suite. À long terme, il serait intéressant de pouvoir développer les automates avec *Python*, *Perl*, *Ruby* ou encore *ECMA Script*. Les clients (*workers*, *ui*, ou *bots*) peuvent se connecter au serveur de différentes méthodes : TCP, TLSv1, *socket* UNIX ou encore via des tubes locaux (*pipe*). Le stockage des informations aux niveau du serveur est actuellement en mémoire mais il est prévu de pouvoir enregistrer l'état du *framework* dans un fichier. Il est également envisageable de stocker les informations dans un *backend* SQL.

Implémentation de la ressource « sherlock-net »

Le *worker sherlock-sniff* est une sonde réseau implémentée avec la bibliothèque *libpcap* [7]. Il collecte les résultats des paquets envoyés par le *sherlock-send*. Il n'y a pas de communication directe entre *sherlock-sniff* et *sherlock-send* permettant au premier de déterminer si un paquet réponse correspond bien à une requête émise par le second. La sonde doit interpréter le paquet réponse pour signaler au serveur si un test est validé. Déterminer avec précision la nature du test effectué nécessite parfois la transmission de données significatives dans la charge utile des

[16] RFC 2827 : « *Network Ingress Filtering - Defeating Denial of Service Attacks which employ IP Source Address Spoofing* »

paquets. L'implémentation actuelle ne sécurise pas ces données par un procédé cryptographique de chiffrement ou de signature. Il est donc actuellement possible de tromper le moteur de *scan* en injectant des données arbitraires sur les réseaux où les sondes écoutent. *sherlock-sniff* possède un cache des informations qu'il communique au serveur afin de minimiser la sollicitation du réseau pour le trafic de contrôle.

Le *worker sherlock-send* se connecte au serveur et attend des ordres. Deux types de *scans* IPv6 sont actuellement implémentés dans *sherlock-send* : *scan* ICMPv6 *Echo Request* (*ping*) et *scan* TCP SYN. L'implémentation des différentes méthodes de *scan* est assez similaire à celle des outils classiques de *scan* IPv4 ou IPv6. La seule différence est que le constructeur de paquets IP intègre la notion de zones variables dans la structure des paquets. Chaque zone variable est calculée selon une expression régulière fournie au préalable.

Support des expressions régulières

Tout d'abord, pourquoi les expressions régulières dans un programme de *scan* IPv6 ? Parce qu'elles permettent d'exprimer des *scans* réseaux relativement complexes de manière synthétique.

Example 7. Un *scan* de port TCP avec des paquets TCP SYN utilisant des ports sources aléatoires (entre 1024 et 2048) à destination des services FTP, SSH, telnet et HTTP pour toutes les adresses EUI-64 de périmètre lien, formées à partir des OIDs constructeurs IBM et Dell :

```
synscan --sport $rand(1024,2048) --dport (2[1-3]|80|8080) \
        fe80::$eui64((@ibm_laptops@|@dell_laptops@)::)
```

Example 8. Un *traceroute* ICMPv6 à destination de toutes les machines du sous-réseau `2000:d:e:f::/96` en utilisant les adresses `2000:a:b:c::1` à `2000:a:b:c::10` comme routeurs intermédiaires :

```
ping --ttl [2-12] --hops 2000:a:b:c::$rand(1,10) 2000:d:e:f::/96
```

Le besoin de support des expressions régulières se fait sentir à tous les niveaux : noms d'hôte, adresses IPv4, adresses IPv6, ports. L'utilisation classique des expressions régulières se ramène à vérifier qu'une chaîne de caractères correspond à une expression régulière. Dans le cadre de l'implémentation de *sherlock-net*, les objectifs sont différents :

- parcours séquentiel de l'espace de recherche,
- estimation du nombre d'itérations nécessaires pour trouver une chaîne donnée par un processus de recherche séquentielle,
- génération d'une chaîne aléatoire correspondant à une expression régulière.

Étant donnée l'immensité de l'espace d'adressage IPv6, l'ordre des requêtes envoyées peut jouer un grand rôle dans la durée d'un *scan* IPv6. Les simulations de *scans* présentent deux grands intérêts. Le premier est que **l'administrateur réseau peut estimer approximativement la durée nécessaire pour *scanner* son réseau**. La deuxième, plus utile du point de vue de *sherlock*, est la possibilité de calculer combien de temps un bot prendrait pour *scanner* un réseau. Dans un premier temps, une phase de collecte d'adresses est menée. Une seule adresse prélevée est fournie comme indice à un bot. L'efficacité du bot peut ainsi être mesurée sur 2 critères : la quantité d'adresses découvertes et la vitesse de découverte des adresses.

La syntaxe (à débattre) des expressions régulières supportées par *sherlock-net* n'est pas identique à la syntaxe classique (ex : *Perl*). Les différents symboles disponibles sont résumés dans le tableau ci-dessous (Tab. 1). L'intérêt du symbole $\backslash n$ est de pouvoir générer des caractères identiques afin de

former des motifs comme « *xyyx* », « *xxyy* », « *xyxy* » avec les expressions « `??\1\2` », « `?\1?\1` », « `??\2\1` ». La seule fonction actuellement implémentée est la fonction eui64 qui génère un suffixe réseau de 64 bits à partir de 48 bits selon l'algorithme EUI-64.

Symbole	Signification
`?`	Joker de 4 bits si l'expression est en hexadécimal. Joker de taille variable si l'expression est décimale.
`*`	Joker qui cherche à occuper le maximum de place dans le contexte de l'expression régulière. La taille maximale est 8 bits pour une adresse IPv4 et 16 bits pour une adresse IPv6.
`[x-y]`	Intervalle de caractères autorisés.
`[xyz]`	Liste de caractères autorisés.
`(xy|yz|z)`	Liste de chaînes de caractères autorisées (taille variable).
`@chemin@`	Dictionnaire.
`$fonction(args..)`	Fonction de génération automatique.
`{n}`	Répétition du dernier fragment de l'expression régulière.
`\n`	Permet de répéter le *n* dernier fragment de l'expression régulière.
`/n`	Limite la validité de l'expression aux *n* premiers bits (masque IP).

TAB. 1: Syntaxe des symboles utilisés dans l'implémentation des expressions régulières de *sherlock-net*

Processus de scan

La méthode de *scan* employée par *sherlock* est totalement dépendante de la configuration des bots qui décident quelles tâches doivent être accomplies. Les tâches peuvent être exécutées en parallèle. Néanmoins, l'objectif final est de suivre plus ou moins le processus suivant :

1. Récolte des informations DNS
   (a) Identification de la zone et des services DNS (requêtes SOA, NS, MX)
   (b) Tentative de transfert de zone DNS (requête AXFR)
   (c) Attaque par dictionnaire des noms d'hôtes (www, ftp, etc ..)
   (d) Attaque par force brute des Reverse DNS
2. Récolte des informations de routage
   (a) Découverte *whois*
   (b) *traceroute*
3. *Scans* à forte probabilité de succès
4. *Scans* à faible probabilité de succès

Parallèlement à ce processus, une analyse constante des informations recueillies est menée afin de modifier l'ordonnancement des tâches. À la découverte d'un motif particulier, le bot *crazy* privilégiera par exemple les *scans* à destination d'adresses ayant le même style de motif. Étant données

les méthodes actives de découverte utilisées, il arrive parfois que des adresses hors du périmètre initial soient trouvées. Un bot peut être amené à automatiquement demander un *scan* sur ces adresses. Pour éviter ces débordements, il devrait être possible de configurer les *bots* pour qu'ils ne s'intéressent pas aux adresses issues de sous-réseaux différents.

4.3 Implémentation des bots

Les bots connectés à *sherlockd* analysent les évènements recensés par les différentes sondes et ajoutent des nouvelles tâches à exécuter. Actuellement 3 automates sont en cours de développement : *crazy*, *bunny* et *bobby*.

Le bot « crazy »

L'objectif de l'automate *crazy* est de mettre des tâches courtes dans la file d'attente du serveur en fonction des informations collectées. Le principe est que l'automate reste passif jusqu'à ce qu'il soit notifié d'un événement tel que :

- détection d'une adresse EUI-64,
- détection d'une adresse *6in4* ou *6to4*,
- détection des numéros de ports intégrés aux adresses.

À titre d'exemple, après avoir détecté une adresse globale EUI-64, le *bot* va essayer de trouver des adresses basées sur le même OID (même constructeur) sur le réseau. De plus, si l'adresse EUI-64 trouvée est de périmètre global, l'automate essayera de déterminer si la machine *scannée* possède une adresse EUI-64 de périmètre lien avec le même identifiant d'interface. L'automate *crazy* est également capable de détecter des motifs simples de 16 bits utilisés dans les adresses découvertes « *xyyx* », « *xxyy* », « *xyxy* ».

Le bot « bunny »

L'automate *bunny* est actif contrairement aux deux autres. Les adresses à *scanner* sont déterminées en se basant sur un modèle de répartition des adresses défini au préalable. Plusieurs expressions régulières sont utilisées pour générer des adresses IPv6. De plus le concept de réduction d'espace d'adressage par probabilité introduit dans le chapitre « 3.3 Réduction de l'espace d'adressage » est utilisé. *bunny* génère en permanence une liste d'adresses IPv6 à *scanner* en respectant les contraintes imposées par les probabilités définies.

Le bot « bobby »

L'objectif global de l'automate *bobby* est de récupérer automatiquement certaines informations sur les hôtes détectés comme la topologie réseau ou les noms DNS :

- détermination de la taille des sous-réseaux avec des requêtes *whois*,
- récupération des noms d'hôtes, reverse et zones DNS,
- *traceroute* multiprotocole,
- prise d'empreintes de l'implémentation de la pile IPv6.

Les différentes tâches que cet automate entreprend sont des tâches relativement courtes en temps de traitement. La prise d'empreinte se résume actuellement à vérifier si les hôtes distants sont des implémentations Linux. Cette détection est réalisée en envoyant un paquet *ping* avec un *Routing Header* indiquant que le paquet doit transiter par l'adresse *localhost* (`::1`) de l'hôte à tester.

5 Conclusion

L'efficacité de *sherlock-net* n'est pas encore au niveau de nos espérances. Beaucoup de travail reste à faire concernant l'implémentation des automates. Cependant les résultats des *scans* réalisés avec *sherlock-net* sont toujours plus performants en vitesse de découverte du réseau que plusieurs *scans* linéaires ou aléatoires manuels. L'important dans la cartographie de réseau IPv6 n'est pas réellement le débit disponible mais plutôt l'ordonnancement des tâches. La découverte d'une seule adresse IPv6 est souvent le déclic qui va entraîner la découverte de nombreuses autres. Le problème est « Combien de temps est-il nécessaire pour trouver les adresses intéressantes ? ». À titre d'exemple, il est peu rentable de *scanner* toutes les adresses EUI-64 d'un réseau sans connaître un des OID utilisés par les équipements de ce réseau. Par contre une fois un des équipements utilisés trouvés, il est plus simple de supposer les OIDs utilisés.

Complexifier les adresses IPv6 permet sûrement d'allonger le temps de découverte d'un réseau, mais ce coût peut facilement se répercuter sur tout le travail d'administration et de maintenance. Pour pallier ce problème, les serveurs DNS arrivent à la rescousse. Le problème est que ces serveurs DNS deviendront à leur tour une source d'informations dangereuse. Les adresses temporaires[17] sont actuellement faiblement utilisées sur la toile IPv6 mais il ne faut pas oublier ce facteur qui pourrait complexifier la découverte de postes client sur un réseau. Étant donné que l'identifiant d'interface (suffixe de 64 bits) de ces adresses est généré par un procédé cryptographique, l'entropie des 64 derniers bits est très forte. Cependant l'avantage du protocole IPv6 est que chaque hôte possède très souvent plusieurs adresses IP.

Dans tous les cas, un *scan* IPv6 sera toujours plus long qu'un *scan* IPv4 et le volume de données envoyées sera toujours plus important. Ce n'est pas pour autant que les *Network Intrusion Detection System* (NIDS) détecterons plus facilement les attaques. Avec IPv6, il est très simple pour une personne malveillante de posséder autant, voir plus d'adresses que le réseau qu'il souhaite *scanner*. Une adresse IPv6 sera donc rarement employée plus d'une fois pour la cartographie réseau. Mettre en quarantaine chaque adresse une par une est impensable, mais déterminer les sous-réseaux utilisés pour une attaque n'est pas une tâche facile.

Quand verra-t-on ce jour où monsieur X trouvera la cafetière (`fe80::cafe:0:1`) et le réfrigérateur (verb+fe80 : :f00d+) du voisin lors d'un *scan* ICMPv6 via le réseau IPv6 de sa XXXbox ?

Références

1. THC IPv6 Attack Toolkit — `http://www.thc.org/thc-ipv6`
2. blindcrawl.pl — `http://sec.angrypacket.com/code/blindcrawl.pl`
3. Bnsr00t.tar.gz
4. Those eXtra Domain NameS 2.0 — `http://www.txdns.net/`
5. Nmap — `http://insecure.org/nmap/`
6. Amap — `http://www.thc.org/thc-amap`
7. tcpdump / libpcap — `http://www.tcpdump.org/`

– draft-ietf-v6ops-scanning-implications-03.txt
– draft-ietf-v6ops-icmpv6-filtering-recs-03.txt

[17] RFC 3041 : « *Privacy Extensions for Stateless Address Autoconfiguration in IPv6* »