


TÉLÉCOM NANCY

Projet MyNmap



```
Starting Nmap 6.00 ( http://nmap.org ) at 2012-05-17 12
Nmap scan report for scanme.nmap.org (74.207.244.221)
Host is up (0.00031s latency).
Not shown: 997 closed ports
PORT      STATE SERVICE      VERSION
22/tcp    open  ssh          OpenSSH 3p1 Debian 3ubuntu7
| ssh-hostkey: 1024 0a:d6:67:54:9d:0a:00:00:00:00:00:00:00:00:00:00
|_2048 79:f8:00:00:00:00:00:00:00:00:00:00:00:00:00:00
80/tcp    open  http         Apache/2.2.3 ((Ubuntu))
|_http-ti
9929/tcp  open  http         Apache/2.2.3 ((Ubuntu))
Device type: general purpose
Running: Linux 2.6.X|3.X
OS CPE: cpe:/o:linux:kernel:2.6 cpe:/o:linux:kernel:3
OS details: Linux 2.6.32 - 2.6.39, Linux 2.6.38 - 3.0
Network Distance: 2 hops
Service Info: OS: Linux; CPE: cpe:/o:linux:kernel
```

Chocot BAPTISTE
Houbre MAËL

Examinatrice : Mme
ISABELLE CHRISMENT

1 Introduction

Cette année le sujet de Réseaux et Systèmes Avancés portait sur un analyseur de port de type Nmap. Ce projet était découpé en plusieurs parties afin d'avoir des objectifs précis à atteindre en fonction de l'avancement du projet. Il concernait la détection des machines actives sur un réseau, un scanner de port TCP, TCP SYN/ACK, UDP et enfin l'utilisation d'une machine zombie pour un scan. Ce rapport détaillera le raisonnement de ces différentes parties en abordant le raisonnement utilisé ainsi qu'une explication d'une partie du code.

La plupart des fonctions sont documentées dans un style proche de celui de la *JavaDoc*, et définies dans les *headers*.

2 La détection de machines actives sur le réseau

2.1 Structure de données choisie

Pour stocker et afficher les résultats de nos scans, nous avons implémenté la structure **ActiveDevice** suivante :

```
typedef struct ActiveDevice {  
    char *address;  
    PortList *list;  
} ActiveDevice;
```

La structure **PortList** est une liste chaînée d'entiers. Les variables de types **ActiveDevice** sont stockées dans une **ActiveDeviceList**, qui est également une liste chaînée. L'attribut *address* correspond à l'adresse IPv4 de la machine. Toutes ces structures possèdent leur set de fonctions (ajout, *free*, debug, etc) et des fichiers de tests ont été implémentés pour vérifier le bon fonctionnement de ces structures.

2.2 Le ping

En ce qui concerne la détection de machines actives (que nous appellerons désormais **ping**), il est d'abord nécessaire de déterminer les adresses des différentes machines à scanner. Pour cela, nous avons codé une fonction *parse*.

```
char** parse(char* network)
```

Cette fonction, prenant en argument une adresse sous la forme "x.x.x.x/n", la sépare, en dégage le nombre de bits n utilisés pour coder l'adresse du réseau local, puis reconstitue la première adresse de ce réseau "x.x.x.x". On obtient donc un nombre d'adresses égal à 2^{32-n} .

Une fois le nombre d'adresses déterminé, on incrémente petit à petit l'adresse d'origine afin d'obtenir les différentes adresses à scanner que l'on stocke dans un pointeur de chaînes de caractères. Une fois que nous avons toutes les adresses que nous voulons scanner, on crée un thread qui va héberger le processus d'écoute des sockets que l'on aura envoyé. Puis pour chaque adresse que nous avons, nous envoyons un socket *raw* avec le *flag* ICMP_ECHO levé. Pour chaque réponse reçue, nous créons une instance de structure **ActiveDevice** contenant l'adresse de la machine ayant répondu et une liste vide de ports. Cette liste sera remplie au fur et à mesure des différents scans.

3 Scan TCP SYN

3.1 Un peu de théorie

Le principe du scan TCP SYN est d'envoyer une trame dans laquelle le *flag* SYN du header TCP est levé. En théorie, l'appareil scanné peut répondre de deux manières. Si le port est ouvert, l'appareil répond par une trame où les *flags* SYN et ACK sont levés. Sinon, il renvoie une trame où RST est levé.

On définit donc une socket utilisant le protocole TCP, et on modifie directement les valeurs des champs qui nous intéressent.

3.2 En pratique

Nous avons utilisé le *multi-threading* pour mener à bien ce scan. Nous lançons d'abord un *thread* annexe d'écoute, qui analyse les trames que la machine reçoit et qui met à jour la structure de données. Le *thread* principal va lui envoyer toutes les requêtes aux différents ports. Cela a pour avantage d'être rapide, et l'utilisation d'autres *threads* est même nécessaire si on souhaite scanner la totalité des ports d'un appareil, à savoir 65535.

Pour ce faire, on utilise les structures *iphdr* et *tcphdr*. Nous avons décrit le contenu du socket par un `char *`. Comme on sait que le *header* IP est en première position, on déclare un pointeur pointant sur le début du buffer, comme ceci :

```
struct iphdr *iph = (struct iphdr *) buffer;
```

On remplit ensuite les champs du *header* IP (adresses, etc). On fait de même avec le *header* TCP :

```
struct tcphdr *tcph = (struct tcphdr *) buffer +
    sizeof(struct iphdr);
```

On se place au début du *header* TCP, qui est situé juste après le IP (ceci est dû à l'encapsulation). On lève simplement le *flag* SYN (on le passe à 1), on calcule le *checksum* et on envoie.

3.3 Résultat

Par chance, ce scan a tout de suite été fonctionnel. Il a juste été optimisé par la suite et adapté pour coller avec l'ensemble du programme. On peut constater via *Wireshark* qu'on reçoit des réponses conformes à nos attentes pour les ports ouverts :

```
aspirateur.home livebox.home TCP 81 search-agent(1234) -> netrjs-2(72) [SYN] Seq=0 Win=5840 Len=27
aspirateur.home livebox.home TCP 81 search-agent(1234) -> netrjs-3(73) [SYN] Seq=0 Win=5840 Len=27
aspirateur.home livebox.home TCP 81 search-agent(1234) -> netrjs-4(74) [SYN] Seq=0 Win=5840 Len=27
livebox.home aspirateur.home TCP 58 domain(53) -> search-agent(1234) [SYN, ACK] Seq=0 Ack=1 Win=5840
```

4 Scan TCP SYN/ACK

4.1 Un peu de théorie

Le scan TCP SYN/ACK est légèrement différent du scan précédent. La trame envoyée vers l'appareil à scanner à cette fois les *flags* SYN et ACK de levés. Si la machine est active sur le port scanné, elle ne répond pas. Si elle ne l'est pas, elle renvoie une trame avec RST levé. Comme nous ne sommes pas sûr d'obtenir de réponse à chaque tentative, le *thread* d'écoute n'a plus d'intérêt ici. On attend à chaque requête une réponse, ou son absence.

4.2 En pratique

Pour ce qui concerne l'envoi de la trame, la fonction utilisée est la même que la précédente, appelée avec des arguments différents.

En revanche, pour l'écoute, le raisonnement est un poil différent. On sait que, lorsque la machine cible ne répond pas, elle est active sur le port scanné. Ainsi, pour éviter de bloquer le programme, nous avons utilisé des *timeouts* (structures *timeval*) réglés sur une centaine de millisecondes pour garantir un minimum d'efficacité.

4.3 Résultat

Nous avons pu constater que ce scan fonctionne bien moins que le précédent. En effet, il affiche un nombre incroyable de ports ouverts. Après une analyse sur *Wireshark*, nous avons constaté que les appareils n'envoient pas systématiquement une trame de réponse lorsque leur port est fermé :

```
118 2018-05-17 20:20:15.5902119... 192.168.1.16 192.168.1.1 TCP 81 search-agent(1234) → mcidas(112) [SYN, ACK] Seq=0 Ack=
119 2018-05-17 20:20:15.5982172... 192.168.1.16 192.168.1.1 TCP 81 search-agent(1234) → auth(113) [SYN, ACK] Seq=0 Ack=1
120 2018-05-17 20:20:15.6021849... 192.168.1.16 192.168.1.16 TCP 54 auth(113) → search-agent(1234) [RST] Seq=1 Win=0 Len=
121 2018-05-17 20:20:15.6024826... 192.168.1.16 192.168.1.1 TCP 81 search-agent(1234) → 114 [SYN, ACK] Seq=0 Ack=1 Win=51
122 2018-05-17 20:20:15.6192148... 192.168.1.16 192.168.1.1 TCP 81 search-agent(1234) → sftp(115) [SYN, ACK] Seq=0 Ack=1
123 2018-05-17 20:20:15.6191721... 192.168.1.16 192.168.1.1 TCP 81 search-agent(1234) → ansanotify(116) [SYN, ACK] Seq=0
124 2018-05-17 20:20:15.6264442... 192.168.1.16 192.168.1.1 TCP 81 search-agent(1234) → uuwp-path(117) [SYN, ACK] Seq=0
125 2018-05-17 20:20:15.6341991... 192.168.1.16 192.168.1.1 TCP 81 search-agent(1234) → sqlserv(118) [SYN, ACK] Seq=0 Ack=

Time 120: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
Ethernet II, Src: Sagemcom_43:21:18 (40:c7:29:43:21:18), Dst: HonHaiPr_6e:f5:eb (40:49:0f:6e:f5:eb)
Internet Protocol Version 4, Src: livebox.home (192.168.1.1), Dst: aspirateur.home (192.168.1.16)
Transmission Control Protocol, Src Port: auth (113), Dst Port: search-agent (1234), Seq: 1, Len: 0
Source Port: auth (113)
Destination Port: search-agent (1234)
[Stream index: 112]
[TCP Segment Len: 0]
Sequence number: 1 (relative sequence number)
Acknowledgment number: 0
0101 .... = Header Length: 20 bytes (5)
Flags: 0x004 (RST)
```

On constate en observant les trames que les rares réponses sont celles des ports que *nmap* détecte (qu'ils soient *open* ou *closed*). Néanmoins, les réponses ont toujours le *flag* RST levé. Nous avons donc conclu que des sécurités doivent empêcher ce scan de fonctionner correctement.

5 Scan TCP IDLE

5.1 Un peu de théorie

Le scan TCP IDLE a pour principe de scanner une machine par le biais d'une autre. Pour le réaliser, il faut utiliser une machine tierce sur un port dont on sait

déjà qu'il est fermé. On lui envoie un simple TCP SYN/ACK pour que cette machine nous renvoie un RST avec un certain id de paquet, qu'on stocke. Ensuite, on envoie un SYN à la machine à scanner avec, en champ source, l'adresse de la machine zombie. La clé est, que si le port est ouvert, elle va répondre à la machine zombie et son id de paquet va être incrémenté. Sinon, il restera inchangé. Finalement, on renvoie un SYN/ACK à la machine zombie et on observe la différence d'id (2 = port ouvert, 1 = port fermé).

5.2 En pratique

Aucune fonction n'a dû être ajoutée pour cette partie. Nous avons juste ajouté un *flag* à lever sur la fonction d'écoute SYN/ACK pour récupérer l'id de la réponse. Le reste avait déjà été réalisé dans les deux parties précédentes.

5.3 Résultat

Le scan précédent n'étant pas fonctionnel, celui-ci ne peut l'être. On constate en le lançant que ces résultats sont incohérents.

6 Scan FTP

Nous n'avons pas du tout traité cette partie.

7 Scan UDP

7.1 Un peu de théorie

Le scan UDP repose globalement sur les mêmes principes que pour le scan TCP, en définissant les *header* IP et UDP. La différence majeure est la façon de déterminer si un port est actif ou non. Pour ce type de scan, on envoie une socket UDP et on écoute avec une socket ICMP. On part du principe qu'un port est actif. Si jamais on reçoit dans un temps raisonnable une réponse de type ICMP_PORT_UNREACHABLE, c'est alors que le port est fermé.

Nous avons choisi un délai d'une seconde d'écoute car, en échangeant avec d'autres groupes, nous avons appris que le délai d'émission de réponse en UDP est d'une seconde. Ainsi, on devrait en théorie obtenir une réponse à chaque fois.

7.2 En pratique

Là encore, le *thread* d'écoute n'a pas lieu d'être ici car, en cas de réponse, on n'est pas sûr d'être capable de récupérer le port (si la réponse est de type ICMP, il n'y a pas de notion de port). On scanne donc port par port, en analysant la réponse, avec un *timeout* d'une seconde.

7.3 Résultat

Encore une fois, le résultat fût surprenant. On ne détecte quasiment pas de ports ouverts en utilisant ce scan. En observant le réseau sur *Wireshark*, on constate que les ports non détectés par *nmap* ne répondent pas, et que les ports détectés réponse systématiquement par une trame ICMP, avec le *flag* ICMP_PORT_UNREACHABLE levé.

```
aspirateur.home livebox.home UDP 70 50701 → microsoft-ds(445) Len=28
livebox.home aspirateur.h... ICMP 98 Destination unreachable (Port unreachable)
```

On remarque que parfois, la trame est envoyée en temps que **QUIC** (Quick UDP Internal Connection).

8 Conclusion

Ce projet fût de loin un des plus intéressants que nous avons jamais réalisé. Au delà du côté un peu "exotique" de l'application (qui peut être considérée comme un logiciel malveillant), la manipulation des *socketsraw* et le fait de pouvoir manipuler comme on le souhaite les entêtes des sockets avait un côté satisfaisant, du fait du contrôle sur les objets qu'on a manipulé.

De plus, le fait d'avoir à manier les entêtes, de devoir calculer les somme de contrôle des *sockets* et de devoir comprendre comment fonctionne les appareils suivant les trames qu'ils reçoient nous a permis d'approfondir nos connaissances en réseau.

9 Sources

Ci-dessous les sources qui nous ont servi afin de mener à bien le projet.

- Pour le ping : <https://www.cs.utah.edu/~swalton/listings/sockets/programs/part4/chap18/myping.c>
- le manuel et les fichiers de la libc
- www.stackoverflow.com

10 Travail effectué

Les tâches indiquées incluent leurs tests.

	B.Chocot	M.Houbre
Gestion de projet et documentation		
Stand-Up Meetings	2h	2h
Recherche	2h	8h
Structure de données		
Conception et implémentation	10h	0h
Ping		
Ping	8h	20h
Scan TCP		
Scan SYN	10h	0h
Scan SYN/ACK	10h	0h
Scan IDLE	1h	0h
Scan FTP	0h	0h
Scan UDP		
Scan UDP	5h	10h
Rapport		
Rédaction	2h	2h
Total		
Total	52h	42h