SEMINAIRE DE DETECTION D'INTRUSIONS

Port Scanning (NMAP)
Les différentes techniques de scans et outils de détection

Deuxième épreuve ingénieur civil informaticien

Année académique 2001–2002

Prof. M. Dacier

Laurent Barsin
Olivier Delcourt
Amaury Duchesne
Phat Piron

Table des matières

Première partie

Les différentes familles et techniques de port scanning

1 Introduction

La première partie de ce rapport énumère plusieurs techniques de port scanning. Le port scanning est utilisé pour observer l'état d'un hôte ou d'un réseau. Il permet de déceler si certains ports de communication sont en attente de connexion; autrement dit, de savoir si ils sont dans un état ouvert (listening). Ces ports représentent des voies de transmissions potentielles. Tracer leur existence peut être utile à un administrateur système afin de détecter si certains ports sont ouverts et les filtrer en cas de besoin. Le port scanning est aussi utilisé par les hackers pour les mêmes raisons mais ceux-ci utiliseront ces ports ouverts pour s'infiltrer et explorer l'environnement d'un réseau ou encore récupérer des informations sur le système.

Le port scanning a donc pour but de découvrir des voies de communications exploitables. L'idée est de sonder autant de voies que possible et de garder celles qui sont réceptives ou particulièrement utiles. On peut faire une analogie entre le port scanning et une politique agressive de publicité. En effet, certains publicitaires envoient des quantités impressionnantes de mails indésirables à un très grand nombre d'adresses. Et il y aura toujours des personnes pour répondre. C'est à ce moment là que vous pouvez tendre vos filets. Pour le port scanning, la technique est similaire si ce n'est que les boîtes à mails ont été remplacées par des ports de communication.

Le plus souvent un port scan est comparable à une attaque frontale. En effet, le hacker doit tester chacun des ports de la machine cible et ce, les uns après les autres. Il faut savoir que tout système possède 65536 ports TCP et 65536 ports UDP dont les 1024 premiers sont réservés à des applications serveur lancées par l'administrateur. L'attaquant envoie une demande de connexion à chacun des ports de la machine cible et attend qu'ils répondent. Si ceux-ci répondent, c'est qu'ils sont ouverts. L'attaquant pourra donc par la suite établir une connexion sur ces derniers. En fait, il s'agit la plupart du temps d'une attaque de reconnaissance. Elle est souvent suivie d'une véritable intrusion. C'est pour cette raison qu'on qualifie souvent le port scanning de pré-attaque. De même, le port scanning est une des techniques les plus populaires pour découvrir et déterminer les services qui écoutent un port spécifié. Ainsi, en utilisant ces techniques, un hacker peut ensuite créer une liste de faiblesses et de vulnérabilités suite au résultat obtenu et ce, dans le but de pouvoir exploiter et compromettre un hôte distant.

Les techniques de port scanning peuvent être principalement regroupées en trois catégories.

- Open scan.
- Half-open scan.
- Stealth scan (ou scan furtif)

Chacune de ces techniques implémente des attaques permettant de localiser les ports ouverts/fermés d'une cible. Les scans dit ouverts sont facilement loggés par la cible et sont donc facilement détectables. Cependant, nous verrons qu'ils ont l'avantage de produire des résultats positifs (ports ouverts) sur lesquels on peut se fier. A l'inverse, utiliser des scans furtifs peut éviter certains IDS et passer outre certains firewalls. Mais, comme nous le verrons également, des résultats signalés comme positifs (ports ouverts) ne pourront pas toujours être digne de confiance. Autrement dit, ces techniques dites furtives peuvent produire des fausses alarmes (*false positive response*). Quant aux techniques de scans demi-ouverts, elle peuvent constituer un juste milieu entre les deux catégories qui viennent d'être décrites. On introduira également une quatrième catégorie appelée «*balayage par ping*» (*ping sweep*) qui ne constitue pas en soi une catégorie de port scan. Comme nous le verrons, il s'agit d'une étape préliminaire à tout port scan qui consiste à déterminer si un hôte est «up» ou non. Pour finir, une section sera également consacrée aux scans UDP. En définitive, on peut définir un port scanning de la façon suivante.

Port scanning – L'art de détecter quels systèmes sont *en vie* et accessibles via l'Internet et de savoir quels services ces systèmes offrent [1], en utilisant des techniques comme le *ping sweeps*, le *port scans* et l'*OS identification*, est appelé *port scanning*. Le port scanning peut être comparé à un voleur vérifiant toutes les portes et fenêtres d'une maison qu'il s'apprête à pénétrer.

Un des outils permettant de lancer un grand nombre de port scans décrits plus bas est NMAP. D'ailleurs, chaque description de port scan sera précédée de sa syntaxe sous NMAP. Outre le port scanning, NMAP permet de faire de l'authentification d'OS par empreinte de la pile TCP/IP de la cible ou encore d'identifier les propriétaires des process écoutant sur les ports scannés. Plusieurs sections de ce rapport y seront consacrées. A titre illustratif, voici un exemple de résultat donné par NMAP.

```
$nmap -sS -O www.xxx

Starting nmap V. 2.54BETA30 ( www.insecure.org/nmap/ )
Interesting ports on xxx (xxx.xxx.xxx.xxx):
(The 1518 ports scanned but not shown below are in state: closed)
Port       State      Service 7/tcp      open       echo
9/tcp      open       discard 13/tcp     open       daytime
19/tcp     open       chargen 21/tcp     open       ftp
25/tcp     filtered   smtp 37/tcp     open       time
53/tcp     open       domain 80/tcp     open       http
110/tcp    open       pop-3 111/tcp    open       sunrpc
161/tcp    filtered   snmp 513/tcp    open       login
514/tcp    open       shell 515/tcp    open       printer
540/tcp    open       uucp 587/tcp    open       submission
1524/tcp   open       ingreslock 2049/tcp   open       nfs
4045/tcp   open       lockd 6000/tcp   open       X11
6112/tcp   open       dtspc 6699/tcp   filtered   napster
7100/tcp   open       font-service 8888/tcp   open
sun-answerbook 32771/tcp  open       sometimes-rpc5 32772/tcp
open       sometimes-rpc7 32773/tcp  open       sometimes-rpc9
32774/tcp  open       sometimes-rpc11 32776/tcp  open
sometimes-rpc15 32780/tcp  open       sometimes-rpc23

No exact OS matches for host (If you know what OS is running on
it, see http://www.insecure.org/cgi-bin/nmap-submit.cgi). TCP/IP
fingerprint:
SInfo(V=2.54BETA30%P=i686-pc-linux-gnu%D=2/24%Time=3C79126E%O=7%C=1)
TSeq(Class=RI%gcd=1%SI=C5AC%IPID=I%TS=100HZ)
TSeq(Class=RI%gcd=2%SI=363A%IPID=I%TS=100HZ)
TSeq(Class=RI%gcd=1%SI=79CD%IPID=I%TS=100HZ)
T1(Resp=Y%DF=Y%W=60DA%ACK=S++%Flags=AS%Ops=NNTNWM)
T2(Resp=N) T3(Resp=N) T4(Resp=N)
T5(Resp=Y%DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
T6(Resp=N) T7(Resp=N)
```

1. Autrement dit, de savoir si des services tournent ou sont dans un état LISTENING en se connectant sur certains ports TCP ou UDP du système cible.

```
PU(Resp=Y%DF=Y%TOS=40%IPLEN=70%RIPTL=148%RID=E%RIPCK=E%UCK=E%ULEN=134%DAT=E)

Uptime 136.861 days (since Wed Oct 10 21:38:52 2001)

Nmap run completed -- 1 IP address (1 host up) scanned in 66
seconds
```

2 Open scan

Afin de bien comprendre cette technique, nous allons rappeler le fonctionnement de TCP ainsi que le principe d'établissement d'une connexion TCP en trois étapes (*Three-way handshake*).

La paire *(adresse IP, numéro de port)* est appelée un *socket* et représente un *end point* d'une connexion TCP. Pour obtenir un service TCP, une connexion doit être explicitement établie entre un socket sur une machine et un socket sur une autre. La connexion TCP est alors identifiée par ces deux *end points* (socket1, socket2). Chaque point peut échanger des *segments*. Un segment TCP est constitué d'un en-tête suivi de données. Un en-tête TCP peut inclure les bits (flags) suivants [2]:

- SYN Synchronise les numéros de séquence pour initier une connexion.
- FIN L'émetteur à terminé d'envoyer des données.
- RST La connexion est réinitialisée.
- ACK Le numéro d'acquittement est valide.

TCP est un protocole de transport fiable et orienté connexion. Le fait que ce protocole est orienté connexion signifie que deux applications utilisant TCP doivent établir une connexion TCP avant d'échanger des données. En assignant un numéro de séquence à chaque segment transféré, et en demandant un ACK au récepteur, TCP peut ainsi garantir une certaine fiabilité. Examinons l'établissement d'une connexion en trois étapes.

1	émetteur	→	SYN XX	→	récepteur
2	émetteur	←	SYN YY|ACK XX+1	←	récepteur
3	émetteur	→	ACK YY+1	→	récepteur

Une connexion s'effectue donc de la manière suivante.

1. L'émetteur E envoie un message TCP (une requête SYN) au récepteur R en mettant son flag SYN (SYNchronise les numéros de séquence) à 1. Une requête SYN spécifie le numéro de port de R que E veut atteindre et un numéro de séquence initial (XX dans notre exemple).
2. E répond avec un message SYN contenant son propre numéro de séquence initial (YY) ainsi qu'avec un ACK en spécifiant quel sera le numéro de séquence du prochain segment attendu (XX+1).
3. E acquitte le message SYN de R, et le transfert de données peut débuter.

Par contre, si un port ne peut accepter la connexion, il la refuse selon le schéma suivant.

1	émetteur	→	SYN	→	récepteur
2	émetteur	←	RST|ACK	←	récepteur
3	émetteur	→	RST	→	récepteur

La plupart des implémentations TCP/IP suivent les règles ci dessous [3].

- Quand un segment SYN (ou un FIN) arrive sur un *port fermé*, TCP jette le segment et *envoie* un RST.

2. Il y a en fait six bits possibles. URG et PSH ne sont pas décrits.
3. A noter que ces règles ne sont que des recommandations que toutes implémentations n'adoptent pas forcement.

- Quand un segment RST arrive sur *port ouvert* ou un *port fermé*, il est simplement jeté.
- Quand un segment contenant un ACK arrive sur *port ouvert*, il est jeté et un RST est envoyé.
- Quand un segment SYN arrive sur un *port ouvert*, le *three-way handshake* de TCP se poursuit en répondant un SYN|ACK.
- Quand un segment FIN arrive un *port ouvert*, il est simplement jeté. En fait, le comportement d'un port devant un FIN est le suivant: port fermé = RST, port ouvert = jeté.

Les techniques que nous allons discuter dans cette section sont dites «ouvertes» parce qu'elles déterminent la présence de ports qui écoutent en ouvrant une connexion complète.

2.1 TCP connect()

Syntaxe `nmap`

`nmap -sT` *cible*

Descriptif

Cette technique de scan est la plus simple que l'on puisse mettre en oeuvre. En effet, il s'agit simplement d'ouvrir une connexion complète, classique et tout à fait légitime, avec certains ports de la machine cible. Si le port est ouvert (en écoute), il acceptera simplement la connexion selon le schéma en trois étapes de TCP. Sinon, il refusera la connexion selon le deuxième schéma présenté ci-dessus. Cela voudra dire que le port est fermé (déjà occupé).

Cette méthode est très rapide, puisqu'on peut effectuer plusieurs scans en parallèle. De plus, elle ne requiert aucun privilège *root* pour fonctionner, puisqu'il s'agit d'une simple demande de connexion classique. C'est-à-dire, la connexion est initiée avec l'appel système *connect()* qui est accessible par n'importe quel utilisateur. Si l'appel de *connect()* retourne la valeur 1, le port et ouvert, sinon, le port est fermé. Evidemment, cette technique est facilement identifiable parce qu'ell ouvre une connexion complète. Cependant, parce que l'appel système *connect()* utilise l'accord en trois étapes de TCP, il est très fiable pour détecter les ports ouverts/fermés.

Puisque cette technique se base sur l'accord en trois étapes de TCP pour se connecter à un hôte arbitraire, une connexion spoofé est impossible puisque le protocole d'accord est corrompu.

On peut déjà esquisser une méthode de détection pour ce type de scan. Les fichiers *log* des programmes qui gèrent les services vont tous noter l'IP de l'attaquant. Il suffit donc de repérer les connexions qui se sont ouvertes, n'envoient aucune donnée et interrompent leur connexion. D'un autre côté, certains IDS ne verront pas cette attaque, puisque comme nous l'avons dit, c'est une demande tout à fait légitime. Il se trouve, en effet, que certains IDS se sont surtout concentrés sur la détection de scans *half-open*(voir plus loin). Ceux-ci ne loggeront pas cette attaque parce que la connexion est complète, alors que si on n'envoie pas le ACK final du *three-way handshake*, ils la loggeront.

Avantages

Rapide, facile, fiable, ne requiert pas de privilèges *root*.

Inconvénient

Facilement détectable et loggable.

Vitesse: rapide.

Stealth: faible.

2.2 Reverse ident scanning

Syntaxe `nmap`

`nmap -I` *cible*

Descriptif

Jusqu'ici, un scan permet de trouver quels ports sont ouverts, mais une autre fonction intéressante des scans est qu'ils permettent, par exemple, d'identifier les caractéristiques et comportements des process qui écoutent les ports des machines cibles. En effet, il peut paraître plus intéressant d'attaquer quelqu'un qui dispose de privilèges *root* plutôt qu'un pauvre petit user tout gentil!

Pour remplir cet objectif, on peut utiliser l'*identification protocol* qui permet de révéler le nom de l'utilisateur propriétaire de n'importe quel process utilisant TCP même si celui-ci n'a pas initié de connexion. Le démon correspondant à ce protocole se nomme *identd (identification protocol daemon* et est accessible par défaut par n'importe quel utilisateur.

Le scanner (NMAP) met à profit ce protocole de la façon suivante. En se connectant à un port TCP (en employant un *TCP connect()*), si le port est ouvert, le scanner envoie une *ident request* au port 113 résevé au daemon *identd* de la machine cible. Pour faire sa requête à la machine, la connexion doit fournir les numéros de ports source et destination, c'est pourquoi une connexion complète est absolument nécessaire. Ce type de scan est connu sous le nom de *reverse ident* parce que, bien qu'à l'origine, la RFC aie suggéré ce protocole pour aider les serveurs à identifier leurs clients, l'implémentation actuelle autorise l'inverse (le client peut identfier un serveur). Comme sous-entendu, cette méthode requiert l'ouverture d'une connexion complète puisqu'elle vient se greffer sur un scan *TCP connect()*. Elle est donc facilement identifiable.

Pour détecter ce genre de violation, on peut se référer au paragraphe correspondant de *TCP connect()*. Sinon, un bon moyen de se prémunir contre ce genre d'intrusion est de simplement désactiver cette fonctionnalité sur votre OS.

Avantages

Identique à *TCP connect()*, il est rapide, sans privilège root mais permet en plus d'obtenir des informations intéressantes.

Inconvénients

Facilement détectable.

Vitesse: rapide.

Stealth: faible.

3 Half-open scan

Le qualificatif «demi-ouvert» s'explique par le fait que le client met un terme à la connexion avant que le three-way handshake de TCP ne soit complété. Ainsi, cette méthode de scan sera souvent indétectable par les IDS se basant sur une connexion complète, et retournera des résultats assez corrects (reconnaissance plutôt fiable des ports ouverts/fermés).

3.1 TCP SYN scan

Syntaxe `nmap`

`nmap -sS` *cible*

Descriptif

L'implémentation de cette méthode de scan est similaire à la connexion complète d'accord TCP en trois étapes à l'exception du fait que, au lieu de renvoyer un ACK lors de la réception d'un SYN|ACK, la connexion est interrompue. Appelée scan demi-ouvert car il ne s'agit pas d'une vraie connexion, cette technique consiste à envoyer un SYN à un port de la machine cible, et si celle-ci renvoie en réponse un SYN|ACK, c'est que ce port est ouvert (LISTEN). Il faut ensuite envoyer immédiatement un RST afin d'annuler cette connexion. Voici le schéma de la demi-connexion quand le port est donc ouvert.

attaquant	→	SYN	→	cible
attaquant	←	SYN|ACK	←	cible
attaquant	→	RST	→	cible

A noter que le SYN envoyé par «l'attaquant» est un SYN différent d'un SYN traditionnel en ce sens qu'il a été créé non pas par le noyau lors d'un appel *connect()* mais par l'utilisateur. De plus, l'envoi du RST n'est pas obligatoire vu que le bit RST est orienté noyau. En fait, en recevant le SYN|ACK de la cible, le noyau s'aperçoit qu'il n'a pas initié la connexion avec un SYN (comme lors d'un *connect()*). La pile TCP/IP du noyau renvoie donc un RST automatiquement.

Inversement, un port fermé répondra avec RST|ACK à la demande de synchronisation de l'attaquant. Puisqu'aucun service ne tourne sur ce port, la cible retourne en fait une demande de reset (RST). Voici le schéma de la demi-connexion quand le port est donc fermé.

attaquant	→	SYN	→	cible
attaquant	←	RST|ACK	←	cible

L'avantage de cette technique de scan vis à vis d'un simple *connect()*, c'est que potentiellement moins de cibles la loggeront en raison du caractère semi-ouvert de la connexion. Il faut savoir qu'il y a encore quelque temps, cette méthode de scanning était difficilement décelable car il fallait détecter au niveau du noyau chaque SYN entrant.

Cependant, il en est tout autrement aujourd'hui. Cette technique est en fait devenue facilement détectable par de nombreux IDS, en raison du fait qu'un grand nombre de programmes de dénis de service (DoS) basent leurs attaques sur l'envoi d'un excès de paquets SYN (TCP SYN flooding). C'est sans aucun doute pour cette raison que les systèmes de détection d'intrusion courants sont capables de logger ces scans demi-ouverts. Ainsi, leur efficacité s'est dégradée ces dernières années. Le SYN scan a, en quelque sorte, subi la trop grande popularité du TCP SYN flooding et ce à son détriment. Les IDS détectant ce dernier font d'une pierre deux coups et détectent également ce scan.

A noter que la façon la plus simple de détecter ce scan serait sans doute de repérer un nombre important de paquets SYN non suivi de connexion. Il suffit alors de traduire cette affirmation sous forme de règles pour un IDS tel que SNORT, pour n'en citer qu'un.

Avantages

Rapide, fiable, évite les IDS primaires ou trop basiques, évite le protocole d'accord du TCP three-way handshake.

Inconvénients

Requiert le privilège root, jeux de règles implémentées dans de nombreux IDS empêchant ainsi beaucoup d'essais de SYN scan.

Vitesse: rapide.

Stealth: moyen.

3.2 IP ID header scanning – «dumb scan» – scanning par témoin

Syntaxe `nmap`

`nmap -sI` *dumb* `-P0` *cible*

Descriptif

Comme déjà dit dans la section précédente, le SYN scan est maintenant aisément détectable. D'autres méthodes de scans dites «furtives» seront présentées dans les sections qui vont suivre. Cependant, aussi furtives que soient ces méthodes, des parades ont été trouvées. Avec le scanning par témoin (ou «dumb» scan), nous franchissons une étape supplémentaire dans la furtivité car l'IP de la machine attaquante n'est jamais visible ! Voyons comment cela est possible.

Encore appelée scan d'en-tête ID (nous verrons pourquoi par après), cette technique de scan est encore plutôt méconnue et fait intervenir les particularités d'implémentation de la pile TCP/IP de la plupart des systèmes d'exploitation. A l'origine, cette technique à été découverte par Salvatore Sanfilippo antirez, qui l'a décrite en détails dans un post sur bugtraq. L'implémentation de ce scan se base sur celle du SYN scan, à ceci près que le dumb scan entraîne l'implication d'un troisième hôte (en plus de l'attaquant et de la cible) en qualité de source factice. En fait la méthode consiste à scanner une machine cible à l'aide de deux autres machines au lieu d'une seule.

Avant d'aller plus en avant, il est important de définir ce qu'est un hôte appellé «muet» (dumb). A l'inverse d'une machine classique, un hôte silencieux ou muet est un serveur qui envoie et reçoit peu ou pas de traffic du tout, d'où l'origine du nom caractéristique dont il est doté. Cette caractéristique est indispensable. Il est important que ce serveur génère peu ou pas de traffic sur son port de même numéro que celui origine du scan. Cependant, il est bon de noter que localiser un de ces hôtes requiert pas mal d'efforts et peut s'avérer contraignant. Les efforts déployés à cette recherche peuvent tout simplement ne pas en valoir la peine et c'est probablement là l'un des plus gros inconvénients de ce scan. Néanmoins, la technique n'en demeure pas moins remarquable sur plusieurs abords.

Le dernier point à aborder avant de présenter le principe du scan est qu'il existe un champ d'en-tête IP nommé ID appelé par la suite IPID pour plus de clarté. Un extrait du RFC 791 stipule que *[. . .] Le choix d'un identificateur de datagrammes est motivé par la nécessité de pouvoir distinguer de façon unique les fragments appartenant à un datagramme particulier [. . .]*. Le RFC 791 conseille que cette IPID soit incrémentée uniquement en cas de fragmentation, ce qui n'est pas le cas dans la réalité. En effet sur beaucoup d'architectures, il est incrémenté linéairement à chaque envoi de paquet IP et notamment à l'envoi de paquet SYN ou RST. Par exemple, on a une incrementation de 256 sous Windows et de 1 pour certains UNIX, 0 pour les Linux 2.4.x. Comme nous allons le voir, le dumb scan utilise ce défaut d'implémentation. Ainsi, comme énoncé, le scénario que nous allons développer implique trois machines:

- A: machine de l'attaquant.
- B: machine muette (ou encore témoin).
- C: machine cible su scan.

Le principe du scan est le suivant.

- Il faut tout d'abord trouver une machine témoin B qui satisfasse l'implémentation incrémentale de l' IPID. Il faut juste qu'elle ne soit pas trop chargée pour permettre de bien mesurer l'incrémentation. Un petit calibrage avant le scan est donc nécessaire en envoyant une dizaines de paquets SYN. Ainsi, la machine n'est pas obligée de n'avoir *aucun* traffic sur le port cible du scan. Il suffit juste que le traffic ne fluctue pas trop et qu'il soit relativement faible. L'hôte A envoie alors à l'hôte B une serie de pings et analyse le champ ID de l'en-tête IP des réponses de l'hôte B. Un hôte muet (dumb) verra l'IPID de sa réponse incrémenté de 1 à chaque séquence de pings[4]. Pour un hôte muet, la situation est donc la suivante (`id=1`+ signifie l'incrémentation du champ `id`).

```
60 bytes from BBB.BBB.BBB.BBB: seq=1 ttl=64 id=+1 win=0 time=96 ms
60 bytes from BBB.BBB.BBB.BBB: seq=2 ttl=64 id=+1 win=0 time=88 ms
60 bytes from BBB.BBB.BBB.BBB: seq=3 ttl=64 id=+1 win=0 time=92 ms
```

- L'hôte A envoie un packet SYN spoofé à l'hôte C en utilisant l'adresse source de l'hôte B. Le port distant que l'attaquant désire tester peut être choisi arbitrairement parmi les 65535 disponibles et ce, afin de découvrir son état (ouvert/fermé).
- L'hôte C, croyant avoir affaire à B lui répondra par une des deux réponses possibles:
  - Une réponse sous forme de SYN|ACK indiquera que le port cible de C est ouvert (listening). L'hôte B renverra donc un RST (ou plutôt un paquet avec le bit RST mis à 1) vu qu'il n'a pas envoyé de SYN. Comme déjà dit dans la section précédente, cette procédure est automatisée par le noyau.
  - Une réponse sous forme de RST|ACK indiquera que le port cible de C est fermé (non-listening), et l'hôte B ignorera ce paquet et ne renverra rien en retour.

Maintenant, comment l'hôte A pourrait-il connaître ce qui a été envoyé à l'hôte B par l'hôte C? En d'autres mots, comment l'hôte A pourrait-il connaître l'état du port cible de l'hôte C?

En fait, en supposant que le port était *ouvert* sur le serveur cible C, les séries de pings parallèles que l'hôte A a envoyés vers B alors que les paquets SYN spoofés sont en cours d'envoi vers C contiendront la réponse à notre question. Analysons le champ IPID de ces réponses pings.

```
60 bytes from BBB.BBB.BBB.BBB: seq=25 ttl=64 id=+1 win=0 time=92 ms
60 bytes from BBB.BBB.BBB.BBB: seq=26 ttl=64 id=+3 win=0 time=80 ms
60 bytes from BBB.BBB.BBB.BBB: seq=27 ttl=64 id=+2 win=0 time=83 ms
```

Remarquez que les IPID des réponses des paquets deux et trois affichent une valeur incrémentale supérieure à 1. Par conséquent un port ouvert a été repéré. Durant cette période, tout incrément supérieur à 1 dans les réponses de l'hôte B indique un port (de C) ouvert. A l'origine, l'incrément était de 1, mais puisque que l'hôte A envoie un SYN spoofé sur un port ouvert de l'hôte C, B doit

4. L'application utilisée qui permet de connaître les IPID se nomme hping et est disponible sur http://www.kyuzz.org/antirez. Il est à noter que les systèmes BSD utilisent un algorithme pseudo-aléatoire générant les numéros de séquences (IPID) au lieu d'un algorithme incréméntal.

renvoyer à l'hôte C un paquet RST|ACK, ce qui incrémente le champ ID.

A l'inverse, si un port est *fermé* sur l'hôte C, l'hôte B ne sera pas tenu de renvoyer un paquet à l'hôte C. Par conséquent, le champ ID dans une réponse ping ne sera incrémenté que de 1.

```
60 bytes from BBB.BBB.BBB.BBB: seq=30 ttl=64 id=+1 win=0 time=90 ms
60 bytes from BBB.BBB.BBB.BBB: seq=31 ttl=64 id=+1 win=0 time=88 ms
60 bytes from BBB.BBB.BBB.BBB: seq=32 ttl=64 id=+1 win=0 time=87 ms
```

Vous remarquez que le champ ID est toujours borné à 1. C'est pour cette raison qu'un hôte «muet» est requis. Ainsi le traffic entrant et sortant reste cantonné à une valeur minimum et ce, afin de diminuer les réponses dites «false-positive» (fausses alarmes). Il est d'ailleurs conseillé de tester plusieurs fois chaque port pour éviter les falses-positives dues à une variation résiduelle du traffic sur l'hôte muet B. Nous obtenons ainsi la liste des ports ouverts sur une machine sans que celle-ci n'ai jamais vu notre IP.

Il est à noter que remplacer les paquets TCP de la machine A vers B par des paquets ICMP, n'est pas possible car l'incrémentation de l'IPID n'est maintenue qu'entre les différents paquets de même type. La combinaison TCP/ICMP a été testée sans succès contre des Windows 2000. Il est cependant possible qu'il existe des OS qui conservent la cohésion de l'IPID quel que soit le type de paquets envoyé.

A ce stade, il est important de remarquer que cette technique de scan par témoin est aussi facilement détectable qu'un SYN scan. En effet, le scan par témoin reste un SYN scan. Ce qui par contre n'est pas détectable, c'est l'IP de l'hôte A (attaquant) qui est totalement invisible pour les hôtes B et C. En effet, le paquet SYN spoofé envoyé sur la cible C ne peut pas être retracé. De plus, la vérification du numéro de l'IPID par l'hôte A n'est pas loggée par l'hôte B. Le scan semble venir de l'hôte B avec TCP *connect()*, mais l'hôte B ne logge rien.

En fait, une variété de méthodes de scan peuvent être utilisées en faisant apparaître un hôte muet. Ce type de scan n'est pas limité à la technique du SYN scan. Toute méthode contraignant un hôte B à répondre au port d'un hôte A pourrait être mise en pratique. C'est notamment le cas pour les FIN scans décrits dans la section qui suit.

Avantages

Le dumb scan étant un SYN scan, les avantages sont identiques. L'avantage supplémentaire non négligeable est que l'identité de l'attaquant est tout à fait indétectable.

Inconvénients

Les inconvénients déjà cités pour le SYN scan restent d'application. Des inconvénients supplémentaires dus à l'utilisation d'un troisième hôte muet surviennent cependant. Ainsi, la méthode étant relativement récente et complexe, elle demeure très lente. De plus, comme déjà mentionné, un de ses plus gros désavantages est de trouver un bon hôte muet (ou témoin). On pourrait envisager d'augmenter la vitesse de scan en trouvant plusieurs de ces hôtes.

Vitesse: très lente.

Stealth: très forte concernant la non-divulgation de l'identité de l'attaquant.

4 Stealth Scan

Il faut savoir, qu'auparavant, étaient qualifiés de scans furtifs toute technique permettant de contourner les IDS, et les fichiers d'enregistrements de connexions (log). Les techniques ayant évolué, un scan furtif ne se limite pas seulement à cela. Les scans remplissant cette fonction sont maintenant connus sous le terme de scan «demi-ouvert» (half-Open). Aujourd'hui, un scan est considéré comme furtif quand il remplit quelques-unes des caractéristiques suivantes.

- Paramètre les drapeaux (ACK, FIN, ...).
  - Ne fait intervenir aucun drapeaux (Null Scan).
  - Fait intervenir tous les drapeaux (Scan Xmas).
- Passe à travers les filtres, firewalls, routeurs.
- Apparaît comme du traffic réseau fortuit.

Les techniques expliquées ci-dessous utilisent, d'une manière générale, le principe inverse des scans «demi-ouverts». Dans les techniques dites «demi-ouvertes», on était attentif aux réponses des ports ouverts. Dans le cas des scans furtifs, seules les réponses des ports fermés nous importent. On déduit le nombre de ports ouverts suivant les ports que l'on suppose comme fermé.

La technique exploite une faiblesse du protocole TCP/IP expliquée dans la RFC 793. En effet, un port fermé recevant un paquet IP (avec FIN, ACK, PSH ,URG ,... activé) est censé répondre par un RST. Tandis qu'un port ouvert, ne fera que l'ignorer et ne renverra rien. Puisque l'on ne reçoit rien, on fait donc l'hypothèse que le port est ouvert. Ce type de supposition est connu sous le nom de *inverse mapping*. Elle n'est pas sans poser quelques difficultés comme nous le verrons plus loin. A noter que les scans furtifs sont difficilement loggables, étant donné qu'ils n'ouvrent pas de connexion complète selon le schéma habituel du *Three-way handshake*.

4.1 Stealth FIN scan

Syntaxe `nmap`

`nmap -sF` *cible*

Descriptif

Le principe de base est très simple, un paquet avec un drapeau FIN activé est envoyé vers la machine. Un port fermé aura comme comportement de répondre par un paquet dont le drapeau RST activé. Les ports ouverts, eux, ne répondent rien. On soupçonnera donc comme ouverts, tous les ports qui n'ont pas répondu. Voici une négociation pour un port ouvert.

attaquant	→	FIN	→	cible
attaquant	←	-	←	cible

Le système d'exploitation de la machine jette le paquet. Inversement, un RST est renvoyé si un SYN atteint un port fermé.

attaquant	→	FIN	→	cible
attaquant	←	RST	←	cible

On peut dire qu'il y a deux façon de tester un port ouvert. La première est de recevoir une liste des réponses des ports fermés et de la soustraire à la liste des ports scannés. Par exemple, on envoie trois paquets sur les ports 1, 2 et 3 d'un hôte distant. Si la réponse est un RST pour les ports 1 et 3, nous comparerons alors la liste originale 1, 2, 3 au RST reçu (port 1 et 3), et en déduirons que 2 est ouvert par comparaison.

Le second test implique l'utilisation d'un timeout pour la réponse des paquets. Si le timeout est atteint avant la réception du paquet en question, alors nous supposerons le port est ouvert. Evidemment cette méthode devrait être évitée autant que possible, en raison du problème des fausses alarmes (false-positive response). En effet, si un firewall ou un filtre arrête le paquet FIN, nous n'aurons pas de réponse de la part du port cible, et par conséquent, on le considèrera comme ouvert, alors qu'il est peut-être fermé.

Les systèmes d'exploitation style de type Windows ne réagissent pas comme prévu (ainsi que certains systèmes Unix), mais renvoient systématiquement un RST que le port soit ouvert ou fermé . Ceci dit, on peut exploiter ce résultat afin d'identifier l'OS utilisé par la machine cible.

Finalement, le FIN est plutôt intéressant, car il pourra passer à travers un plus grand nombre de firewalls ou filtres que le SYN scan classique puisque les IDS implémentent moins la détection de FIN.

Avantages

Evite le protocole d'accord TCP en trois étapes, évite pas mal de firewalls et IDS basiques.

Inconvénients

Problème évoqués des fausses alarmes, ne fonctionne pas sur tous les OS

Vitesse: moyenne.

Stealth: moyen-fort.

4.2 Xmas Scan

Syntaxe `nmap`

`nmap -sX` *cible*

Descriptif

En se basant sur le principe que le renvoi d'un RST de la part d'un port fermé ne fonctionne pas uniquement avec un drapeau FIN activé, mais aussi avec PSH, URG, ACK, SYN, il est facile d'implémenter d'autres méthodes.

La méthode qui active tous les drapeaux de l'en-tête TCP se nomme Xmas, en rapport avec l'effet décoratif que donne l'activation de tous les drapeaux nous faisant penser ainsi, à un sapin de Noël... Son comportement est identique au FIN scan en ce sens qu'à l'arrivée du paquet le serveur le jette, si le port était ouvert. A l'inverse, un RST sera retourné si le port était fermé. La différence est que le Xmas scan passera les filtres FIN et d'autres artifices susceptibles de bloquer les FIN scans. Ceci dit, on est toujours pas à l'abris de fausses alarmes. Et les systèmes Windows ainsi que certains systèmes Unix ne répondent pas comme spécifié par les normes, empêchant dès lors la détection des ports. On peut donc, aussi, dans ce cas-ci se baser sur ce problème pour déduire le système d'exploitation de la machine cible.

Les points avantages, inconvénients, rapidité et stealth sont identiques à ceux du FIN scan.

4.3 Null Scan

Syntaxe `nmap`

`nmap -sN` *cible*

Descriptif

Cette méthode est l'opposée du Xmas scan. En effet, elle consiste en la désactivation de tous les drapeaux. Le résultat est identique aux deux méthodes citées précédemment. Elle permet de passer un autre type de firewalls ou de filtres mais apporte les même difficultés au niveau des fausses alarmes, et des systèmes d'exploitation ne répondant pas comme voulu.

Les points avantages, inconvénients, rapidité et stealth sont identiques à ceux du FIN scan.

4.4 Fragmentation

Syntaxe `nmap`

`nmap -s[F|X|N] -f` *cible*

Descriptif

La fragmentation n'est pas une technique à part entière. Elle est à combiner avec les trois méthodes vues précédemment. Il s'agit de fragmenter les paquets IP envoyés, pour rendre la tâche des filtres et firewalls plus ardue. En effet, ils ne défragmentent pas les paquets, tâche qui leurs prendrait beaucoup trop de temps et de ressources, et qui avant tout est dévolue au machine se trouvant derrière ce firewall.

Cependant, certains réseaux, machines ou systèmes d'exploitations ont des problèmes de gestion des paquets fragmentés. Un comportement inattendu peut alors se produire (plantage, redémarrage, ...). Mais un avantage non-négligeable de cette méthode est que les systèmes de détection d'intrusions se basent de plus en plus sur les signatures (pour pouvoir détecter les FIN, ...). La technique de fragmentation peut donc être utile pour contourner ces IDS.

Avantages

Permet de contourner les IDS, firewalls et filtres.

Inconvénients

Parfois, un comportement inattendu de la machine cible.

5 Les balayages

On peut utiliser des paquets ICMP pour déterminer si une machine cible est en vie ou non. On envoie simplement un paquet *ICMP ECHO request* (ICMP type 8) à la cible. Si l'on reçoit en réponse un message *ICMP ECHO reply* (ICMP type 0), l'hôte est en vie. Sinon, on le considère off-line.

Interroger de nombreux hôtes par l'intermédiaire de cette technique est appelé un *balayage par ping* ou encore un *Ping Sweep*. C'est en somme une étape préliminaire à tout scan plus perfectionné et notamment à tout *port scan*. En effet, à quoi bon lancer un scan sur les milliers de ports d'une machine si elle est *"down"*.

5.1 ICMP ECHO Ping Sweep

Syntaxe nmap

`nmap -sP -PI` *cible*

Descriptif

C'est le *sweep* détaillé plus haut, basé sur l'envoi de messages *ICMP ECHO request*.

Bloquer les balayages ICMP est facile puisqu'il suffit de ne pas prendre en compte les requêtes ICMP ECHO venant de l'extérieur de son propre réseau. Microsoft.com est par exemple impossible à pinger par cette méthode. NMAP conclut après quelques secondes que ce serveur est *"down"*.

On peut par ailleurs envoyer une requête ICMP ECHO à une adresse broadcast, ce qui permet de scanner encore plus simplement un réseau cible. La requête sera broadcastée à tous les hôtes en vie sur le réseau cible, et ceux-ci répondront rapidement à l'attaquant qui a pu se contenter d'envoyer quelques paquets. Remarquons cependant qu'une machine Windows ne répond pas à une requête envoyée à une adresse broadcast.

Avantages

Première reconnaissance du réseau cible, simplicité de mise en oeuvre du broadcast ICMP.

Inconvénients

La plupart des firewalls sont configurés pour bloquer le trafic ICMP ECHO.

Vitesse: très dépendante des outils employés.

Stealth: faible.

5.2 TCP Sweeps

5.2.1 TCP ACK Sweep

Syntaxe nmap

`nmap -sP -PT` *cible*

Descriptif

A la place d'envoyer des requêtes ICMP echo, on envoie des paquets TCP ACK et on attend les réponses. Un hôte présent doit répondre par un RST. Ceci permet de passer à travers des réseaux n'autorisant pas le ping. On peut par ailleurs spécifier vers quel port ces paquets TCP ACK sont envoyés. Par défaut, il s'agit du port 80 (`-PT80 ou -PT` dans Nmap) (le moins souvent filtré). Les ports 21, 22, 23 et 25 sont aussi judicieux, les autres étant normalement protégés par un pare-feu.

Remarquons cependant que la réponse de l'hôte est très dépendante du système d'exploitation, des pare-feux, des routeurs, et autres dispositifs de filtrage de paquets. A noter que microsoft.com est toujours aux abonnés absents.

Avantages

Pas aussi facile à bloquer qu'une requête ICMP Echo.

Inconvénients

Réponse du système distant variable.

Vitesse: se référer à la section consacrée au scan du même nom.

Stealth: se référer au scan du même nom.

5.2.2 TCP SYN Sweep

Syntaxe `nmap`

`nmap -sP -PS` *cible*

Descriptif

On envoie un SYN plutôt qu'un ACK. Toujours par défaut sur le port 80 (`-PS80`). L'hôte présent doit aussi répondre par un RST (très rarement par un SYN|ACK).

Les points avantages, inconvénients, rapidité et stealth sont identiques à ceux du point précédent.

Remarque – NMAP utilise par défaut une combinaison du ping TCP et du ping ICMP avant de procéder à un port scan à moins que l'option `-P0` soit mise (elle demande à nmap de ne pas pinger l'hôte avant de le scanner).

6 Divers

6.1 UDP/ICMP Error

Syntaxe `nmap`

`nmap -sU` *cible*

Descriptif

Il s'agit de découvrir les ports UDP ouverts sur une machine. Pour cela, on envoie un paquet UDP de 0 octets sur chaque port de la machine. Si on reçoit un message *ICMP port unreachable* (type 3, code 3), alors le port est fermé. Autrement, il est considéré comme ouvert (on ne reçoit rien).

Beaucoup de services très vulnérables écoutent sur de l'UDP: snmp, tftp, NFS, ... Ce type de scan n'est donc certainement pas inutile.

Ceci dit, les scans UDP présentent de nombreux désavantages.

- UDP est un protocole sans connexion et sans contrôle de flux. Les paquets que nous envoyons ont de bonnes chances de se perdre. On conclut alors que beaucoup de ports sont ouverts alors qu'ils n'ont en fait jamais reçu nos paquets. Il faut donc retransmettre plusieurs fois nos paquets de manière à donner à nos tests une certaine pertinence.
- Par ailleurs, d'un point de vue technique, un port fermé ne doit pas nécessairement envoyer le message *ICMP port unreachable*. Ce qui remet à nouveau en cause la fiabilité du scan.

- De plus, ce scan peut s'avérer très lent face à certains systèmes d'exploitations. Linux respecte la RFC 1812 qui stipule qu'il faut limiter la vitesse de transmission des messages d'erreur ICMP à 80 toutes les quatre secondes (et appliquer une pénalité en cas d'excès). Solaris est encore plus lent. Microsoft, par contre, n'a pas trouvé judicieux de respecter la RFC 1812 et l'on peut donc scanner d'un trait les 64K ports d'une machine Windows.
  Remarque – La mise en oeuvre d'un tel scan est donc plus complexe qu'il n'y paraît. Nmap détecte une telle limitation sur la vitesse d'émissions de paquet ICMP et ralentit plutôt que d'inonder tout le réseau.
- Il faut avoir les privilèges *root*, car il faut pouvoir accéder au raw ICMP socket pour lire le port inaccessible.

Avantages

Détection d'OS, éviter les IDS pour TCP.

Inconvénients

Peu fiable, requiert les privilèges ROOT.

Vitesse: lente.

Stealth: moyenne.

6.2 FTP bounce attack

Syntaxe `nmap`

`nmap -b` *cible*

Descriptif

Avant d'introduire la méthode proprement dite, exposons la raison pour laquelle elle est possible. Les RFC ont proposé que les serveurs FTP disposent d'une fonctionnalité «proxy». A l'origine, la RFC 959 avait suggéré cette option pour permettre à un client d'établir simultanément deux connexions FTP avec différents serveurs et ensuite transférer les données d'un serveur à l'autre. Seulement, la plupart des implémentations pouvaient être contournées afin qu'un serveur FTP envoie des données presque partout sur l'Internet. Cette vulnérabilité peut être exploitée pour implémenter un «*FTP proxy scan*».

Soit S la machine qui scanne, C la machine cible, et F un serveur FTP accessible des deux machines. Voici les détails de la méthode.

$$\text{Scanner (S)} \quad \longleftrightarrow \quad \text{Serveur FTP (F)} \quad \longleftrightarrow \quad \text{Cible (C)}$$

- S, pour ouvrir une connexion FTP doit fournir deux adresses : une pour le serveur FTP, et l'autre qui indique l'adresse à laquelle doivent être envoyées les données du serveur vers la machine cible. Dès lors, en utilisant la commande PORT, S initie une connexion avec F, en précisant un certain port de C, soit p_C, comme étant le port dont on a besoin pour le transfert «proxy».
- S essaye de commencer un transfert de données proxy, en utilisant la commande LIST (donne le contenu du répertoire).
- La réponse est alors envoyée au port spécifié, c'est-à-dire p_C. Dès lors, soit p_C est en train d'écouter (port ouvert) et la connexion est acceptée (réponse 150/126). Soit le port est fermé, et on reçoit le message «`425 can't open data connection`».

Il suffit alors de répéter cette méthode pour chaque port, et de cette façon, on listera tous les ports ouverts.

Le principal intérêt de cette méthode, est qu'il est pratiquement impossible que la machine cible repère le scan au travers du serveur disposant d'une fonctionalité proxy. La furtivité du scan est encore accrue si le serveur FTP dispose lui-même d'un firewall. Cependant, le serveur garde toujours une trace des logs de demande de connexion à caractère «proxy» pendant quelques jours.

L'autre côté pratique de cette méthode est qu'il permet de passer outre certains firewalls puisque le port FTP est souvent un de ceux qui reste ouvert à une connexion éventuelle. Une autre possibilité intéressante, est de scanner le serveur lui même en lui donnant l'adresse de rebouclage «127.0.0.1». C'est un point sur lequel certains concepteurs de firewalls ne se sont pas penchés. Ce rebouclage permet alors de scanner la machine en ignorant le firewall. Certains serveurs, néanmoins, désactivent cette possibilité de bouclage. Un autre défaut, lors de l'utilisation de ce scan, est qu'il ne permet pas d'obtenir les *bannières* des process scannés, ni d'implémenter des programmes qui permettront d'exploiter ces données (voir section 2.2).

La méthode la plus évidente pour parrer à ce genre d'attaque, pour le serveur FTP, est de vérifier que l'adresse destination fournie par le client est bien conforme. Cela empêche déjà que le serveur puisse servir de «point de rebond» pour ce type d'attaque. Dans ce cas, le scan renverra toujours port fermé. Enfin pour ne pas être la cible de ce genre d'intrusion, la meilleure possibilité est de désactiver le port 20, qui est censé gérer les connexions FTP.

Avantages

Contourne certains firewalls, rend la source quasiment intracable.

Inconvénients

Plutôt lent, maintenant connu et patché.

Vitesse: lente.

Stealth: fort.

7 Détection d'OS

7.1 Objectif

Les failles de sécurité sont souvent propres à un OS. C'est la raison pour laquelle l'identification de l'OS est de première importance.

7.2 Authentification par l'empreinte de la pile TCP/IP

Cette technique exploite les différences entre les implémentations de cette pile dans les OS.

L'idée est d'envoyer des paquets TCP spécifiques à la cible et d'observer la réponse. La réponse permettra de cerner un groupe d'OS voire même un seul OS. La réponse varie parce que l'implémentation de la pile TCP/IP n'est pas la même d'un concepteur de logiciel à l'autre. Ceci vient d'interprétations différentes des lignes de conduites suggérées par les RFC au moment où la pile en question a dû être écrite.

7.3 Tactiques exploitées par NMAP

Ci-dessous, on trouvera un aperçu des six principales tactiques exploitées par NMAP pour découvrir l'OS distant. NMAP parvient grâce à elle à récupérer des informations qu'il compare avec sa propre base de données[5], sans cesse réactualisée. Voici ces six techniques.

1. *Test FIN* – Il consiste à envoyer un paquet FIN à un port ouvert. La RFC793 stipule que le comportement adéquat est de ne pas répondre à un tel paquet. Cependant, de nombreuses implémentations de pile TCP/IP renverront un RST. C'est le cas de MS Windows, BSDI, CISCO, HP/UX, MVS ou IRIX.
2. *Echantillonnage TCP ISN (Initial Sequence Number)* – On tente de déduire des choses intéressantes du numéro de séquence initial choisi par l'implémentation TCP distante quand elle répond à une demande de connexion. Essentiellement deux techniques utilisées par OS récents
   - Incréments aléatoires: Solaris, IRIX, FreeBSD, Digital UNIX, Cray, Linux, OpenVMS, AIX,...
   - Modèle dépendant du temps où l'ISN est incrémenté périodiquement d'une certaine valeur. C'est le cas de Windows.
3. *Fenêtre TCP initiale* – On peut facilement tirer des conclusions de la taille de la fenêtre TCP choisie initialement par l'implémentation TCP distante, car celle-ci est plus ou moins constante suivant les types d'OS. Il s'agit d'un test utile car certains OS sont identifiables par cette seule tactique. AIX est le seul OS utilisant la valeur 0x3F25. OpenBSD et FreeBSD utilisent Ox402E.
4. *Citation de message ICMP* – Les RFC stipulent qu'un message d'erreur ICMP doit toujours citer une partie du message ICMP à l'origine de l'erreur. Pour un message *ICMP port unreachable*, presque toutes les implémentations renvoient l'entête IP + 8 octets. Cependant, Solaris renvoie un peu plus et Linux encore un peu plus. Nmap parvient donc à reconnaître trivialement Linux et Solaris par cette méthode même si ils n'ont pas de ports à l'écoute!
5. *Intégrité des messages d'erreurs ICMP* – Comme signalé juste au-dessus, les machines doivent joindre une partie du message original quand elles retournent un message d'erreur ICMP. Certaines machines utilisent les en-têtes IP comme espace de travail et les altèrent donc. AIX et BSDI renvoient un champ IP "taille totale" trop grand de 20 octets. BSDI, FreeBSD, OpenBSD, ULTRIX et VAX altèrent le champ "ID" envoyé. Alors que le checksum doit changer à cause de la modification du champ TTL (Time To Live), certains OS (AIX, FreeBSD,...) renvoient un checksum inconsistant ou nul. Mêmes constatations avec le checksum UDP. NMAP fait 9 tests différents sur les erreurs ICMP.
6. *Options TCP* – Elles fournissent des informations très précieuses. Plusieurs points positifs les concernant
   - Toutes les machines ne les implémentent pas.
   - Celles qui les implémentent le signalent. Quand on envoie une requête avec une option mise à une cible, celle-ci placera ladite option dans la réponse uniquement si elle supporte cette option.
   - On peut positionner beaucoup d'options dans un seul paquet pour tout tester en une seule fois. En effet, quand on reçoit la réponse, on regarde quelles options sont retournées et donc supportées. Certaines machines en supportent plus que d'autres. C'est donc un excellent moyen de discernement. Et même si plusieurs OS supportent le même ensemble d'options, on peut parfois les distinguer par la valeur de ces options. Et si ces valeurs coïncident aussi, on peut encore différencier les OS par l'ordre dans lequel les options sont données.

7.4 Tactiques non employées dans NMAP

Voici encore d'autres techniques qui permettent d'obtenir des informations sur le système d'exploitation de la machine cible. Elles ne sont pas implémentées dans NMAP pour les quelques raisons que

5. fichier nmap-os-fingerprints

nous auront l'occasion d'évoquer ci-dessous.

1. *Le test du flag buggé* – Il consiste en l'envoi d'un paquet SYN avec placement d'un flag non défini. Les machines Linux (noyau <2.0.35) conservent se flag dans leur réponse. Certains OS réinitialisent (envoi d'un RST) la connexion quand ils reçoivent ce paquet buggé.
2. *Bit Don't Fragment* – Pour améliorer les performances, certains systèmes d'exploitation placent systématiquement le bit *Don't Fragment* à 1 sur certains des paquets qu'ils envoient. L'examen de ce comportement peut donner à l'attaquant une information supplémentaire sur l'OS distant.
3. *Message d'erreur ICMP* – La RFC 1812 suggère de limiter le débit auquel les messages d'erreurs ICMP sont envoyés. Seuls quelques systèmes d'exploitation suivent cette RFC. Un attaquant peut donc envoyer des paquets UDP à un port UDP aléatoire et compter le nombre de messages d'erreur reçu en un temps donné. Ce test rendrait cependant la procédure de détection d'OS plus longue, puisqu'il faut envoyer un lot de paquets et attendre leur retour. De plus, il faut encore gérer toute la problématique liée à la non fiabilité d'UDP. Des paquets peuvent se perdre et nos mesures s'en trouveront faussées. Tout ceci fait que cette approche n'est pas utilisée dans NMAP. NMAP est cependant bien conscient de cette limitation dans la vitesse d'envoi de messages d'erreur ICMP puisqu'il est capable de la détecter lors d'un scan UDP et de s'adapter à la vitesse de la cible.
4. *Valeur ACK* – Les implémentations de la pile TCP/IP peuvent même différer dans la valeur utilisée pour le champ ACK. Lors de l'envoi d'un FIN|PSH|URG à un port fermé, la plupart des implémentations mettent le numéro d'acquit dans le paquet retourné à la même valeur que le numéro de séquence reçu. Windows répondra par contre avec une valeur de ACK égale au numéro de séquence + 1.
5. *Type de service (TOS)* – Quand un message *ICMP Port Unreachable* est envoyé, un attaquant peut examiner le champ Type de Service. Presque toutes les implémentations utilisent 0 pour la valeur de ce champ. Mais Linux utilise 0XC0.
6. *Gestion de la fragmentation* – Les différentes implémentations gèrent souvent différemment les fragments IP se recouvrant. Certaines recouvrent la vieille partie avec la nouvelle, d'autres procèdent inversement.

8 Techniques de Port Scanning

Certaines *techniques* de scans peuvent être ajoutées aux différentes familles de scans que nous venons de décrire. En voici les principales. A noter que ces techniques ne sont pas toutes combinables avec tous les scans précédemment définis.

8.1 Port scan «aléatoire»

Beaucoup de systèmes de détection d'intrusion (IDS) commerciaux et de pare-feux recherchent des tentatives de connexion séquentielles. Rendre aléatoire la séquence de ports testés peut donc être intéressant.

8.2 Scan lent

Les systèmes de détection d'intrusion peuvent déterminer si une IP spécifique essaie de scanner les ports des ordinateurs du réseau qu'ils protègent. Et ceci, en analysant le trafic réseau sur une certaine période de temps. Cette période de temps est encore appelée *seuil de détection*.

Certains hackers sont très patients et font usage de scanners de réseau qui étalent le scan sur une très longue période de temps. Le débit de scan peut être aussi faible que 2 paquets par jour par réseau cible. Si l'attaquant peut obtenir le *seuil de détection* de sa cible, il peut réduire les chances de détection à zéro, pour peu qu'il n'inclue pas une signature spécifique à ses paquets qui alerterait d'une autre façon l'IDS.

8.3 Scan incluant de la fragmentation

Cette section est juste une redite de la section 4.4. En fait, l'anonymat de scans SYN, FIN, XMAS ou NULL peut être améliorer en fragmentant les paquets IP envoyés. De cette façon, les en-têtes TCP sont elles-mêmes fragmentée parmi plusieurs paquets rendant plus difficile la détection de ces scans par les filtres ou IDS concernés.

8.4 Decoys

Certains scanners de réseaux permettent d'utiliser des decoys ou leurres. Cela permet de noyer sa propre IP parmi toute une série d'adresses spoofées. Ce qui conduira les systèmes de détection d'intrusions à penser que le réseau qu'ils gèrent est scanné par tous ces hôtes, et déterminer le réel attaquant est quasi impossible.

La tactique qui permettait jadis aux IDS de détecter les leurres reposait sur les valeurs du champ TTL dans les paquets scannés. Si tous les paquets entrants ont la même valeur pour le champ TTL, il est plus que vraisemblable qu'ils viennent de la même source. Les outils modernes de scan n'ont bien sûr pas eu beaucoup de mal à trouver la parade. NMAP, par exemple, génère des paquets disposant d'une valeur du champ TTL choisie aléatoirement entre 51 et 65.

Cependant, les IDS pourraient toujours tenter de retracer la route vers l'IP source réelle [6]. Si l'attaquant utilise des IP non routables ou que les IP appartiennent à des hôtes «down», il y a probablement anguille sous roche.

Les leurres peuvent, dans NMAP, être utilisés aussi bien dans la phase préliminaire de ping scan que dans le port scan proprement dit. Ils peuvent également être utilisés lors de la procédure de détection d'OS.

8.5 Scans coordonnés

Même un scan lent peut parfois être détecté. Quand un groupe de hackers travaillent ensemble à essayer d'obtenir l'accès à un réseau cible, on a affaire à une *attaque coordonnée.*

Si de multiples IP scannent un réseau cible, chacune d'elle scanne un service déterminé sur une certaine machine à des périodes de temps différentes et il est dès lors presque impossible de détecter ces scans. Les attaques coordonnées bien conçues sont pour la plupart impossible à détecter à l'heure actuelle. Sans rentrer dans les détails, elles permettent aux attaquants de disposer de plus de furtivité, de «puissance de feux» et de récupérer un plus grand nombre de données.

6. Routeurs traçant la route.

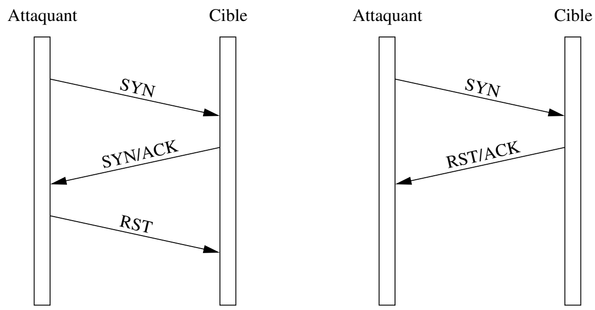
Deuxième partie

Détection de SYN et de FIN scans

Après une présentation des différentes techniques de portscanning lors de cette première partie, nous avons décidé de nous concentrer principalement sur la détection de deux types de scan: TCP SYN scan et TCP FIN scan. La section qui va suivre rappelle brièvement le principe de leur fonctionnement. Pour plus de détails concernant ces scans, nous reportons le lecteur aux sections 3.1 et 4.1 de la première partie.

1 Rappels

Le SYN scan (`nmap -sS`) est un des scans TCP les plus populaires. L'outil de scanning génère simplement un paquet SYN en direction d'un port choisi sur une machine cible. Si ce paquet atteint un port ouvert, la cible lui répondra avec un paquet SYN/ACK. Cela signifie que le port cible est dans un état d'écoute (autrement dit, il est ouvert). Puisque c'est tout ce que le scanneur a besoin de savoir, un RST est renvoyé au port cible pour mettre fin à la connexion. Par contre, si le port est fermé la cible répondra par une demande de reset (RST). Finalement, si il n'y a pas de réponse de la cible, cela suggère que le port visé est protégé par un firewall situé entre la machine cible et le scanneur. La figure 1 résume la situation.



(a) Port cible ouvert (b) Port cible fermé

FIG. 1 – *Demi-connexion lors d'un TCP SYN scan.*

Ce type de scan est communément appelé un scan *semi-ouvert* puisqu'aucune connexion complète ne sera jamais établie. En fait, puisque l'outil de scanning ne complète pas le *three way handshake* de TCP et ce, même si un SYN|ACK est retourné, l'appel *accept()* du socket de la cible n'arrivera pas à complétion. Par conséquent, le scan ne sera (généralement) pas loggé sur la machine cible. L'inconvénient de cette technique est que le scanneur doit lui même construire l'entièreté du paquet IP renfermant le premier SYN. Des privilèges *super-user* sont donc requis.

Quant au FIN scan (`nmap -sF`), son principe de fonctionnement est tout aussi simple. Le scanneur envoie un paquet FIN sur un port cible. Si ce paquet arrive sur un port fermé, l'implémentation TCP de la cible ignore le paquet **et renvoie** un RST. Dans le cas contraire, lorsqu'un paquet FIN est envoyé sur un port ouvert, il est tout simplement laissé tombé (aucun RST n'est renvoyé). Puisque le scanneur ne reçoit rien, il fait l'**hypothèse** que le port est ouvert (cf. *reverse mapping* de la section 4 de la première partie). Cela cause bien évidemment un très grand risque de *false positives* puisque le scanneur ne peut être sûr que le paquet FIN qu'il a envoyé est arrivé à destination. Il peut avoir été jeté en chemin par un firewall ou tout simplement avoir été perdu. Le scanneur considérera alors, peut-être à tort, le port cible comme étant ouvert. La figure 2 illustre la situation.

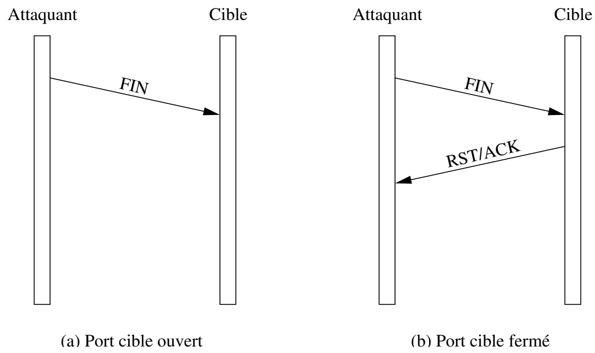


FIG. 2 – *Échange lors d'un TCP FIN scan.*

2 Le point de vue du scanneur

Il y a, en général, deux objectifs qu'un attaquant essaye d'atteindre lorsqu'il effectue un portscan. Le premier, et le principal, est de collecter un maximum d'information sur l'accessibilité et le statut de certaines combinaisons d'adresses IP et de ports (TCP, dans notre cas). Le second objectif consiste à noyer les systèmes de détection d'intrusions avec des alertes et ce, dans le but de «distraire» les défenseurs du réseau ciblé ou tout simplement de les empêcher de faire leur job.

Nous utiliserons par la suite le terme d'*empreinte du scan*[7] pour caractériser l'ensemble des combinaisons port/IP pour lesquelles l'attaquant marque un intérêt. Il faut bien faire la distinction entre l'empreinte d'un scan et son *script*, qui se réfère à la période de temps durant laquelle l'attaquant a essayé d'explorer une certaine empreinte. Une empreinte est tout à fait indépendante des différents aspects d'un script. Un script révélera par exemple la rapidité d'un scan, si il est aléatoire, etc. L'empreinte d'un scan, elle, représente l'information qu'un attaquant a collecté selon les exigences du scan. Le script, lui, rencontrera ces exigences ou peut-être d'autres conduisant à la collecte d'information non désirée par l'attaquant.

Jusqu'à présent, le type le plus commun d'empreinte de scan reste ce qu'on appelle un *scan horizontal*. Il faut entendre par ce terme que l'attaquant dispose d'un *exploit* pour un service particulier et que celui-ci est intéressé par tout hôte qui exploiterait ce service. Ainsi, l'attaquant scanne le port derrière lequel tourne ce service et ce, sur une grande quantité d'adresses IP pour lesquelles il marque aussi un certain intérêt. Ce type de scan est principalement réalisé de manière séquentielle sur un port tel que le port 23 (DNS). D'autres ports sont également dignes d'intérêt comme le port 139 (incluant le fichier NETBIOS et le partage d'imprimante), le port 98 (linuxconf) ou encore le port 23 (telnet) pour n'en citer que quelques uns. La distribution des ports «intéressants» change au gré de la popularité des différents exploits dans la communauté des attaquants.

Le pendant du scan horizontal est le *scan vertical*. On parle de ce type de scanne lorsque l'attaquant scan tous ou une partie des ports d'une seule machine. Dans ce cas, l'attaquant est intéressé par la machine en particulier et tente de découvrir les services qui tournent sur cette machine, peut-être dans l'idée de déterminer quel exploit tenter contre celle-ci. A noter que dans certains cas, le scan peut se

7. L'empreinte d'un scan ne doit pas être assimilée à sa signature, notion qui sera abordée par la suite.

limiter à une faible quantité de ports bien ciblés. On peut également imaginer de combiner les deux types de scans précédemment cités. On parle alors de *scan en bloc* visant un grand nombre de services sur de nombreuses machines.

3 Rappels des techniques de contournement d'IDS

Dans cette section, nous essayons de rendre compte des techniques les plus évidentes dont un attaquant dispose pour contourner des systèmes de détection d'intrusions. Cette section constitue un rappel de la section 8 de la première partie.

Changement de l'ordre des scans – La plupart des scans balayent des adresses IP séquentiellement en allant de la plus petite à la plus grande. Cependant, si cette information est utilisée par les IDS, il est très fortement conseillé à l'attaquant de modifier son type de balayage. Rendre aléatoire l'ordre dans lequel les adresses IP et les ports sont scannés peut se faire très facilement. `nmap` est tout à fait capable d'effectuer cela.

Scan lent – En ralentissant le scan, un attaquant peut le rendre beaucoup plus difficile à détecter. Il peut contourner très facilement des détecteurs de scans en étalant le scan dans le temps afin que les «coups de sonde» sur les ports d'une machine apparaissent en dehors de la *fenêtre de détection* du détecteur [8]. Cela oblige les détecteurs à logger plus d'information. Cela les contraint également à devoir récupérer des empreintes de scan parmi un trafic normal plus important. Par conséquent, la détection n'en devient que plus ardue. Le prix à payer pour l'attaquant est qu'il prendra plus de temps pour obtenir les informations qu'il recherche. (Il peut donc s'avérer utile de caractériser les scans par le *taux d'information moyen* – cf. section suivante – que le scan désire atteindre; autrement dit, combien de bits d'information par seconde il découvre en moyenne).

Rendre aléatoire le temps s'écoulant entre deux coups de sonde – Des délais déterministes entre deux coups de sonde subis par un port peut aider certains algorithmes de détection. C'est pourquoi il est préférable pour les attaquants de rendre aléatoires ces délais.

Rendre aléatoire les champs moins prioritaires – Des champs comme ceux correspondant au numéro de séquence, au numéro de l'acquit, à l'IP id et au port source dans un paquet émanant d'un scanneur sont souvent «hard codés» avec des valeurs fixées par le scanneur (`nmap` s'est longtemps exposé à ce genre de problème). Ils peuvent aussi être générés par des algorithmes déterministes. Cela rend la détection plus facile. Il est donc indispensable pour l'attaquant de rendre aléatoire les valeurs de ces champs.

Spoofing – L'adresse source est plus difficilement falsifiable pour un attaquant. Elle constitue par conséquent une information de premier choix pour un détecteur de portscans. Cependant, dans la plupart des cas, l'attaquant doit utiliser une véritable adresse IP puisque il a besoin de récolter les paquets que la cible génère en réponse à son scan et ce, afin de connaître quels ports sont véritablement ouverts sur la machine cible. Malgré tout, une façon pour l'attaquant de masquer son adresse serait d'utiliser un grand nombre de *decoys* (introduits à la section 8 de la première partie).

Scan distribué ou coordonné – Un attaquant qui peut initier ses scans à partir d'un nombre important de véritables adresses IP peut rechercher différentes parties de l'empreinte de son scan à partir de plusieurs endroits physiques différents. Cela complique le tâche des IDS.

Par la suite, nous supposerons que toutes ces techniques sont connues des attaquants et qu'ils les mettent en oeuvre.

8. Nous introduirons par la suite la notion de fenêtre de détection d'un détecteur de portscans (cf. préprocesseur de snort).

4 Le point de vue du détecteur

Il existe pas mal de polémiques entourant le portscanning. L'une d'entre-elles consiste à savoir si le scan de réseaux distants sans la permission de leurs administrateurs est en soi une activité illégal ou du moins répréhensible. L'activité n'a en fait rien d'illégale; elle n'en reste pas moins hautement répréhensible. L'expérience des administrateurs de réseaux montre que les portscans non sollicités, qu'ils détectent dans la pratique, sont pour la plupart utilisés dans le but de compromettre le réseau ciblé une fois le scan effectué. Ainsi, il semble raisonnable de considérer des portscans comme des attaques hostiles, du moins potentiellement.

Du point de vue de la détection et dans un premier temps, il peut être important de remarquer que tous les scans qui impliquent des combinaisons de flags illégales (l'ensemble des *stealth scans* dont le TCP FIN scan fait partie) sont extrêmement simple à détecter. Des règles qui produisent simplement une alerte lorsque tout paquet comprenant une combinaison non standard de flags apparaît, détecteront de tels scans avec un risque de false positives presque nul. Ainsi, bien que ces techniques soient souvent référencées comme étant des «scans furtifs», elles ne sont pas très furtives du point de vue de la détection. Paradoxalement, un système de détection d'intrusions est beaucoup plus facilement mis à mal par un TCP connect scan (qui réalise une connexion complète), par un TCP SYN scan (qui ouvre une demi-connexion) ou encore par un scan UDP où, dans chacun de ces cas, les paquets émanant des scans pourraient paraître comme du trafic normal et fortuit. C'est pour cette raison que nous préférons nous pencher sur ce type de scan (et plus particulièrement sur le TCP SYN scan).

Il nous sera très utile par la suite d'être capable de caractériser l'«étendue» ou la «grandeur» d'une empreinte de scan, ou encore l'étendue de la portion de l'empreinte d'un scan qui englobe un domaine particulier d'adresses IP. Cela permettra de cerner avec quelle difficulté un scan pourra être détecté, et donc de mesurer l'efficacité de la détection d'un scan. Il est donc très utile de pouvoir déterminer la taille d'une empreinte de scan. Par exemple, un scan par bloc d'un très grand nombre de ports sur l'entièreté d'un réseau de classe C conduit l'attaquant à vérifier 16.646.144 combinaisons IP/port. C'est assez difficile à cacher. Au contraire, si un attaquant décide simplement de caractériser l'état d'un unique port sur une machine bien déterminée d'un réseau donné, il lui sera beaucoup plus facile de contourner un éventuel système de détection.

La méthode la plus simple pour mesurer une empreinte consiste tout simplement à compter le nombre de combinaisons IP/port que l'attaquant a besoin de tester. On appelle cela la *taille totale* de l'empreinte. Du point de vue de l'attaquant, c'est une relativement bonne métrique pour caractériser la taille de son scan, puisqu'elle peut être directement reliée au nombre de bits d'information que l'attaquant essaye de collecter au sujet du réseau cible. Cependant, il n'en est pas toujours ainsi. Parfois, l'attaquant est intéressé par l'obtention de bits d'information provenant de chaque combinaison IP/port. La détection d'OS rentre dans cette catégorie. Par conséquent, il peut s'avérer utile de parler d'*information totale* de l'empreinte qui est le nombre total de bits d'information que le scan essaie d'obtenir.

Cependant, il existe d'autres types de métriques qui peuvent être utiles du point de vue de la détection. Pour motiver celles-ci, nous considérons que certains scans peuvent être plus facilement détectables que d'autres. Ainsi, si un scan décèle un port ouvert, il est beaucoup moins aisé pour nous de déterminer si il s'agissait bien d'un scan que dans le cas où le scan aurait atteint un port fermé. En effet, si l'on considère qu'aucune erreur de configuration n'a été faite, du trafic normal n'atteindra presque jamais un port fermé. Généralement, un utilisateur normal sur Internet ne tentera pas d'identifier un service en initiant une connexion vers un port d'une certaine adresse IP pour savoir si le service est derrière ce port. Au lieu de cela, il se basera sur d'autres types de services (comme un DNS par exemple) pour savoir si un tel service est présent sur un port bien déterminé. Par conséquent, **des connexions en direction de ports fermés sont de façon inhérente beaucoup plus suspectes que des connexions sur des ports ouverts**. Cela ne garantit pas cependant qu'elles soient hostiles puisque des erreurs provenant d'utilisateurs peuvent toujours survenir (et conduire à des false positives). Ainsi, une autre mesure de l'empreinte d'un scan est sa *taille de ports fermés* c'est à dire le nombre de combinaisons IP/port que le scan tente d'atteindre et dont les ports sont en fait fermés au moment de l'exécution du scan. Ces différentes constatations sont à la base de la technique de détection mise en oeuvre lors de la section 6.

Parallèlement, une approche de détection de scan peut être de détecter le «paquet scannant» (le SYN d'un TCP SYN scan par exemple). On parle alors de *forward scan detection*. Une autre approche, plus judicieuse cette fois, est d'observer les paquets qui pourraient être des paquets faisant écho à des sollicitations d'un scan et provenant de ports fermés (ce sont des paquets de réponses); TCP renvoie par exemple un RST dans le cas d'un SYN scan (et un paquet ICMP «port unreachable» dans le cas d'un scan UDP). On parle alors de *backward scan detection*. L'avantage de la deuxième méthode de détection est que les paquets sont, de façon inhérente, beaucoup plus anormaux que des paquets épiés lors d'une détection de type *forward scan detection*. Ces deux types de détection peuvent, en fait, s'avérer être complémentaires. A noter que la technique de détection par *backward scan detection* est à la base du raisonnement de la technique de détection présentée à la section 5.

En fait, les détecteurs de portscans parmi les IDS actuels sont encore assez primitifs à l'image du préprocesseur de snort dont une version épurée (détection de SYN et de FIN scan uniquement) sera présentée à la section 7. Il se base sur la signature classique d'un portscan: «**Un portscan est généralement réalisé sur un grand nombre de ports durant un temps relativement court**». A partir de cette signature, l'algorithme mis en oeuvre par le préprocesseur est le suivant: *Si le préprocesseur repère N paquets envoyés vers N combinaisons différentes de port/IP dans un temps de moins de M secondes, il s'agit d'un portscan.* Les paramètres N et M peuvent être choisi par l'utilisateur. La pratique montre que $N = 5$ et $M = 7$ sont les valeurs les plus acceptables. Il n'est pas compliqué de comprendre que cette algorithme est aisément contournable par un attaquant, ne serait ce qu'en implémentant un scan lent. Nous reviendrons sur le préprocesseur ainsi que ces avantages et inconvénients lors de la section 7.

Finalement, nous détaillerons un détecteur de portscans qui tente de ne pas se limiter à la signature classique que nous venons d'évoquer. Il s'agit de SPICE *(Stealthy Probing and Intrusion Correlation Engine)*. Cette méthode de détection implique le maintient d'un modèle probabiliste qui englobe l'activité totale du réseau à surveiller. Cela est effectué en encodant toute une série de tables de probabilités conjointes qui permettent de savoir approximativement quelle sera la probabilité d'avoir affaire à des paquets dits anormaux. Ils seront ainsi conservés plus longtemps suivant leur degré d'«anomalie». Les paquets seront ensuite regroupés au sein d'un même portscan supposé en s'aidant d'heuristiques développées à partir de vrais portscans. SPICE est en théorie capable de détecter toutes les techniques de scans actuelles en minimisant le nombre de false positives. Cette technique sera abordée lors de la section 8 où l'on définira également le degré d'anomalie d'un paquet.

5 Détection d'un nombre de SYN $\gg$ SYN/ACK

Une première approche de détection de SYN scan (les FIN scans étant facilement détectables) consiste à observer un nombre plus important de paquets SYN que de paquets SYN/ACK. En fait, la poignée de main de TCP est une opération tout à fait normale sur un réseau. Il ne faut pas être alarmé par les paquets émanant de cette poignée de main. En fait, ce qui est anormal, c'est de ne pas terminer la poignée de main et donc d'observer un nombre plus important de SYN vis à vis du nombre de SYN/ACK. Cette situation indique plus que probablement la présence d'un scan semi-ouvert. Mais de quels SYN/ACK parle-t-on? Ceux renvoyés par la cible (étiquettés (a) sur la figure 3) ou ceux qui (ne) sont (pas) renvoyés par l'attaquant lors de la complétion de la poignée de main TCP (étiquettés (b) sur la figure 3)? Les deux sections suivantes font la distinction entre ces deux types de SYN/ACK et listent les avantages et les inconvénients à compter un type plutôt que l'autre.

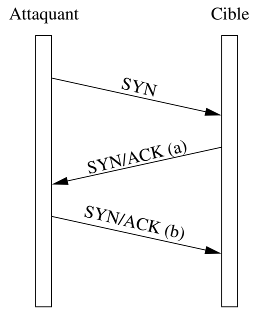


FIG. 3 – *Le three way handshake.*

5.1 SYN $\gg$ SYN/ACK (a)

Dans cette section, nous comptabilisons le nombre de SYN/ACK renvoyés par la cible lors d'une demande de synchronisation (SYN) contractée par l'attaquant. On calcule ensuite le rapport entre les SYN envoyés par l'attaquant et les SYN/ACK renvoyés par la cible pour obtenir un certain ratio.

Dans ce cas de figure, nous faisons une hypothèse assez importante et qui est en général vérifiée. Comme dit précédemment, un portscan est généralement effectué sur un grand nombre de ports durant un temps relativement court. Notre hypothèse consiste à dire que la plupart des ports d'une machine cible sont fermés et que, par conséquent, le scan atteint beaucoup de ports fermés sur un laps de temps assez restreint. Puisqu'un RST est renvoyé lorsqu'un SYN atteint un port fermé, il n'y aura pas de SYN/ACK réémis par la cible. Si le scan touche un grand nombre de ports fermés, il y aura un déficit important de SYN/ACK qui ne seront pas comptabilisés. Ainsi, à partir du ratio calculé précédemment, on peut déduire la présence ou non d'un portscan.

L'inconvénient majeur de cette type de méthode de détection est qu'elle est (et donc le ratio avec elle) très dépendante du trafic. Prenons l'exemple du cas d'une petite entreprise qui voit passer, sur une semaine de temps, 400.000 SYN. C'est déjà un trafic important. Disons qu'elle dispose d'un serveur WEB très fréquenté et que la plupart de ces demandes de connexions sont destinées au port 80 de son serveur. Si un SYN scan est effectué (mettons que le scan atteignent 1000 ports fermés) durant cette semaine, le rapport est de 400.000 SYN pour 399.000 SYN/ACK, soit un ratio de 1,0025. Il faut donc avoir un seuil de tolérance inférieur ou égal à 1,0025 et donc proche de 1 pour pouvoir détecter un scan (tout en sachant qu'un ratio de 1 reflète l'absence de SYN scans et est donc le témoin d'une activité normale). Au contraire, si le scan vise la machine d'un particulier exposé à peu de traffic, le ratio «seuil» devra être plus élevé puisque que si on enregistre 3 SYN pour 2 SYN/ACK (un seule tentative de scan donc) le rapport sera de 1,5. La tolérance devra être inférieure ou égale à 1,5 pour détecter le scan. Cette exemple illustre toute la difficulté de choisir un bon ratio «seuil» suivant le trafic auquel sont exposés les machines à protéger.

Avantage:

- Facilement implémentable.

Inconvénients:

- Difficulté de choisir un bon ratio de tolérance lié à l'influence du trafic. Il faudrait donc s'adapter systématiquement au trafic.
- Repose sur l'hypothèse que la plupart des ports sont fermés. Quid si la plupart sont ouverts ?
- Repose sur l'hypothèse que le scan touche beaucoup de ports sur un laps de temps assez court. Quid si le scan est lent ?

- Pour un ratio de tolérance trop élevé, il n'y aura pas de détection si un scan se fait sur un ou quelques ports bien ciblés. La méthode n'a que peu de chances de détecter des scans horizontaux qui cherchent à exploiter un exploit en particulier.
- L'attaquant peut gonfler artificiellement le nombre de SYN et de SYN/ACK comptabilisés afin d'éviter une quelconque détection (cf. la fin de la section suivante pour plus d'explications).

5.2 SYN $\gg$ SYN/ACK (b)

Une solution plus habile consiste à comptabiliser non pas les SYN/ACK renvoyés pas la cible mais ceux qui doivent être renvoyés par la machine (attaquant) qui a fait une demande de synchronisation et qui est nécessaire à la complétion de la poignée de main TCP.

Cette méthode peut en effet s'avérer plus fiable que la précédente. Revenons un peu sur la section précédente. En comptabilisant les SYN/ACK étiquetés (a) sur la figure 3, on rendait compte d'une anomalie sur les ports **fermés** uniquement et non sur les ports ouverts. En effet, comme déjà souligné, il est hautement anormal de recevoir des SYN sur des ports fermés. Par contre, il est tout à fait normal de recevoir des demandes de synchronisation sur des ports ouverts. Si on calcule le ratio précédent en ne tenant compte que des ports fermés, un ratio différent de 1 peut être jugé anormal (en fait, il sera même à coup sûr infini puisqu'aucun SYN/ACK (a) ne devrait être renvoyé). Par contre, le même ratio calculé en ne tenant compte cette fois que des ports ouverts sera immanquablement toujours de 1, alors que des scans peuvent très bien avoir touché des ports ouverts. Ils ne seront pas détectés par la première méthode !

Par contre, si l'on décide de comptabiliser dans notre ratio les SYN/ACK étiquetés (b) sur la figure 3, dans les deux cas de figure (ports ouverts ou fermés), il ne sera pas normal d'avoir un rapport de 1. En effet, sur un port ouvert, il ne sera pas normal de recevoir un RST au lieu d'un SYN/ACK (b). Et sur un port fermé, le simple fait de tenter de synchroniser le port (et donc d'envoyer un SYN) est considéré comme anormal. En fait, dans le cas d'un port ouvert et lors de notre première méthode de détection, on ne tient pas compte du RST que l'attaquant pourrait renvoyé à la suite de la réception d'un SYN/ACK (a). Il est tout à fait normal de renvoyer ce SYN/ACK (a); c'est après, lors de la dernière étape de la poignée de main TCP, que l'on peut se rendre compte d'une anomalie si le port cible était ouvert. En fait, cette méthode repère si la poignée de main TCP s'est complétée. Si ce n'est pas le cas, une anomalie est constatée.

Une parade peut cependant être envisagée contre ces deux systèmes de détection. Dans le cas du premier système de détection (SYN $\gg$ SYN/ACK (a)). Si l'attaquant décide, en connaissance de cause, de «flooder» la cible de SYN sur un port ouvert (qu'il aura préalablement déterminé), il lui suffit alors de faire monter le quota de SYN sur ce port ouvert puis de scanner les autres ports. De cette façon, si le nombre de SYN sur le port ouvert est suffisant, l'attaquant augmente artificiellement le nombre de SYN et de SYN/ACK (a) comptabilisés avant de scanner les autres ports. Cela a pour conséquence de garder le ratio du détecteur proche de 1 et d'ainsi éviter la détection.

Pour ce qui est de la deuxième méthode de détection (SYN $\gg$ SYN/ACK (b)), on pourrait imaginer un mécanisme identique mais cette fois en gonflant artificiellement les SYN et les SYN/ACK comptabilisés par des connexions complètes (TCP connect). A noter qu'un attaquant serait peu enclin à prendre ce genre d'initiative de peur d'être loggé. Dans le premier cas, on pouvait flooder la cible en utilisant soit TCP connect, soit en ouvrant une demi-connexion puisque la première méthode était insensible à ce qui pouvait se passer après la deuxième étape de la poignée de main TCP.

Avantages:
- Facilement implémentable.
- Indépendant d'une quelconque hypothèse concernant les ports fermés. Le ratio sera représentatif si le scan atteint des ports ouverts **ou** fermés. Précédemment, il l'était seulement sur des ports

fermés.
- Le choix du ratio de tolérance est plus évident et moins sujet au trafic puisque certaines hypothèses sont tombées.

Inconvénients:
- L'attaquant peut gonfler artificiellement le nombre de SYN et de SYN/ACK comptabilisés afin d'éviter une quelconque détection.
- Le problème des false positives reste entier dû au fait qu'un système mal configuré peut envoyer des SYN sans intentions de nuire.

5.3 Une implémentation en langage C

Nous avons rédigé un programme en C, nommé `synscan.c` (disponible sur http://www.student.montefiore.ulg.ac.be/~barsin/detection.html). Celui-ci tente d'appliquer les différentes méthodes mises en évidence précédemment. Ce programme est accompagné d'une règle snort simple, qui enregistre dans un fichier (dans notre cas, `alert`) trois types de paquets. Tout d'abord, les paquets SYN reçus par la machine surveillée (adresse IP passée en argument au programme synscan), ensuite, les SYN|ACK envoyés par la machine surveillée et enfin, les SYN|ACK reçus par cette dernière. Le programme parcourt le fichier `[file]` passé en argument (fichier de snort, dans lequel celui-ci a enregistré les paquets) toutes les `[x]` sec (également passé en argument). Il dénombre et classe les différents paquets. Au final, avec les résultats de l'analyse, il calcule les deux ratios introduits préalablement:

$$\frac{\#SYN}{\#SYN/ACK(a)} \quad \text{et} \quad \frac{\#SYN}{\#SYN/ACK(b)}$$

Cette méthode reste assez simple, et permet de détecter la plupart des SYN scans. Il reste le problème délicat du ratio qui est difficile à déterminer, et qui dépend fortement du trafic. Pour rappel, deux inconvénients peuvent être mis en évidence. Le premier est le *flooding*, qui reste une bonne façon de contourner le détecteur, et enfin, la quantité de false positives non négligeable. Remarquons qu'un scan lent sera détecté, vu l'enregistrement de tous les paquets SYN, et le parcours régulier du fichier par le programme. Reste à savoir la quantité de ressources que le détecteur est prêt à allouer pour logger ces paquets.

6 Détection de SYN sur des ports fermés

Comme déjà signalé à maintes reprises, il est hautement anormal de recevoir des SYN sur des ports fermés. Une technique de détection simple consiste à répertorier ces ports fermés sur la machine ou le réseau à protéger. Une fois les plages de ports fermés repérées, il suffit d'écrire des règles (snort par exemple) qui génèrent une alerte sur toute tentative de synchronisation sur un port fermé de la machine ciblée. On peut même inclure dans ces règles que, le fait de recevoir un paquet FIN sur un de ces ports fermés, constitue une tentative de portscan. Les règles générées sont alors du type:

```
Alert TCP any any -> $HOST head_port:tail_port (flags:*SF; msg:"message d'alerte";)
```

avec respectivement `head_port` et `tail_port`, les bornes inférieures et supérieures d'un intervalle de ports fermés.

Avantages:
- Indépendant du trafic.
- S'adapte assez facilement à la machine à surveiller.

Inconvénients:
- Nécessite une mise à jour fréquente des règles si la liste des ports fermés fluctue rapidement dans le temps. Les règles varient donc d'une machine à l'autre suivant les services offerts.

- Le nombre de règles peut s'avérer prohibitif si le nombre de ports ouverts est important. En effet, une règle est nécessaire pour chaque plage (ou intervalle) de ports fermés. Cet inconvénient peut cependant être évité en éliminant des règles pour des intervalles trop exigus et ce, au détriment de la qualité de détection.
- Cette méthode est beaucoup moins tolérante aux false positives que les précédentes car l'envoi d'un SYN sur un port fermé génère automatiquement une alerte alors que dans les sections précédentes, il fallait attendre l'établissement d'un certain ratio. La politique de détection est donc beaucoup plus agressive. Une amélioration consisterait à attendre la réception d'un nombre de SYN (ou FIN) (nombre géré par l'utilisateur) avant de déclencher une alerte.
- Le problème des fausses alertes demeure:
  - *False positives*. Comme déjà signalé, toute erreur de connexion (qu'il s'agit d'une mauvaise adresse IP ou d'un mauvais port), due à une mauvaise configuration d'un système, sera perçue comme une attaque. Le système n'est pas très tolérant.
  - *False negatives*. Les scans sur les ports ouverts ne seront pas détectés. Cette constatation peut cependant s'avérer souvent fausse puisqu'un scan (pour peu qu'il ne soit pas horizontal) touchera toujours des ports fermés.

6.1 Une implémentation en langage Perl

Nous avons écrit un petit script PERL qui implémente cette méthode de détection. Il consiste dans un premier temps à lister les plages de ports fermés. Pour ce faire, le script analyse le résultat de la commande UNIX `netstat` qui permet de répertorier les ports ouverts d'une machine. Le script trie ensuite ces ports ouverts par ordre croissant. Il suffit alors de déduire les plages de ports fermés de la liste des ports ouverts. Le reste du code consiste simplement à générer les bonnes règles SNORT sur les plages de ports fermés considérés.

Il est bon de noter que le script ne s'intéresse qu'aux ports *bien connus* (les 1024 premiers) afin d'éviter une explosion du nombre de règles générées sur certains systèmes. Ce choix se justifie par le fait que ce sont ces ports qui sont les plus susceptibles d'intéresser les attaquants (pour par exemple exploiter un exploit bien connu). Ce qui n'est pas le cas des ports supérieurs au 1024-ème qui sont ouverts au gré des utilisateurs.

Comme déjà énoncé dans la section précédente, ce script pourrait être amélioré en implémentant un seuil de détection qui rendrait la détection plus «tolérante». Il comptabiliserait ainsi un certain nombre de SYN sur des ports fermés avant de déclencher une alerte. Un autre inconvénient de cette méthode, c'est de générer des fichiers `log` assez volumineux. En effet, tout SYN sur un port fermé sera répertorié. Ce problème sera cependant réglé par le préprocesseur de portscan de SNORT qui est capable d'identifier la provenance d'un scan et ainsi, de ne générer qu'une et une seule alerte sur un SYN scan. Enfin, le script PERL `closed.pl` est disponible à l'URL suivante http://www.student.montefiore.ulg.ac.be/~barsin/detection.html.

7 Le préprocesseur de portscans de Snort

Syntaxe: `preprocessor portscan: IP/CIDR maxPorts maxTime LOG_DIR`

Les arguments sont donc les suivants:
1. `IP/CIDR`, le bloc réseau/CIDR à surveiller pour les scans de ports.
2. `maxPorts`, le nombre de ports accédés durant la période de détection.
3. `maxTime`, le nombre de secondes à compter durant laquelle la limite d'accès aux ports est considérée.
4. `LOG_DIR`, le répertoire/fichier où placer les alertes. Les alertes sont également écrites vers le fichier d'alerte standard.

7.1 Fonctionnement général

Le *préprocesseur* de portscans recherche un nombre `maxPorts` de paquets TCP (ou UDP) envoyés vers un nombre quelconque de combinaisons de ports et d'adresses IP **depuis une même source** en un nombre `maxTime` de secondes, où `maxTime` et `maxPorts` ont été détaillés ci-dessus. Une version épurée `spp_portscan.c` (tenant compte uniquement de parties se référant aux SYN scans et aux Stealth scans) est disponible à l'adresse http://www.student.montefiore.ulg.ac.be/~barsin/detection.html.

Tout paquet qui arrive est analysé et classifié dans une structure de donnée complexe détaillée à la section suivante. Si le paquet n'a aucun lien avec un quelconque des paquets déjà passés par Snort au préprocesseur, il devient le noeud de départ d'une nouvelle structure de données. Autrement, le compteur de paquets de la structure correspondante est incrémenté, et une vérification est faite pour déterminer si le seuil de `maxPorts` paquets en `maxTime` secondes a été dépassé. Si ces deux dernières conditions sont vérifiées, le scan est reporté à l'utilisateur. Une remarque s'impose concernant les *stealth scans*. Comme déjà signalé auparavant, ceux-ci sont en fait très faciles à repérer pour un détecteur d'intrusions. Une simple règle dans Snort suffit à les identifier. Ceci dit, le problème est qu'une alerte est générée pour chaque paquet. Le préprocesseur résout sans mal ce problème, et ce, toujours grâce à la structure de donnée complexe que nous détaillons dans la section suivante. Le préprocesseur ne générera donc qu'une seule alerte pour un SYN scan repéré.

7.2 Classification des paquets

Le *préprocesseur* utilise, pour classer les paquets, une structure de listes liées hiérarchisées. La liste «mère» est *scanList* (cf. figure 4). Il s'agit d'une liste de sources («*SourceInfo*»), qui reprend l'ensemble des sources qui ont établi une connexion avec les machines que Snort surveille. Dans les cellules de cette liste, nous disposons, entre autre, de l'adresse de la machine source, du nombre de destinations auxquelles elle s'est connectée, du nombre de connexions qu'elle a établies (ces nombres ne sont pas toujours précis, et sont surtout là pour donner une idée sur la sévérité du scan). Il y aussi des informations contenant le temps de connexion, et certains champs qui permettent de déterminer si il y a, éventuellement, un véritable scan. Enfin, il y a un lien vers la liste liée de toutes les destinations («*DestinationInfo*») avec lesquelles la source est en contact (si ces destinations sont dans le champ de surveillance de Snort). Les cellules de cette liste nous donnent l'adresse de la machine destination, le nombre de connexions en cours entre la source et la destination, et un pointeur vers la liste de ces dernières («*ConnectionInfo*»). Les cellules de cette liste nous donnent des informations concernant les ports par lesquels les connexions sont établies, le type de scan qui est réalisé (en regardant les flags, on sait si le paquet est éventuellement hostile) et différentes données sur les paquets qui ont transité (*flags*, temps d'arrivée, ...). Pour une vision synthétique de cette structure, on se référera utilement à la figure reffiglist.

7.3 Fonctions

Le préprocesseur utilise une grande quantité de fonctions pour créer et actualiser les listes liées, pour gérer les fichiers logs, ... Nous décrivons uniquement celles que nous considérons comme les plus importantes, c'est-à-dire celles qui permettent de déterminer si la connexion en cours fait partie d'un scan.

La fonction principale est `void PreprocFunction(Packet* p)`. Elle va prendre tous les paquets IP entrant, et déterminer si ces paquets peuvent faire partie d'un scan ou non. Sa première vérification consiste à voir s'il s'agit bien d'un paquet IP. Si le paquet reçu n'utilise pas ce protocole, elle ne le traitera pas. Ensuite, elle utilise la fonction `ScanType CheckTCPFlags(u_char th_flags)`, qui lui permet de repérer le type de scan auquel elle est confrontée. Si rien de suspect n'a été détecté, il est inutile de continuer, et on sort de la fonction, sinon il est temps d'analyser ce scan plus en profondeur.

```
if (not ip) then break
   else Vérifier si ce paquet peut faire partie d'un portscan
```

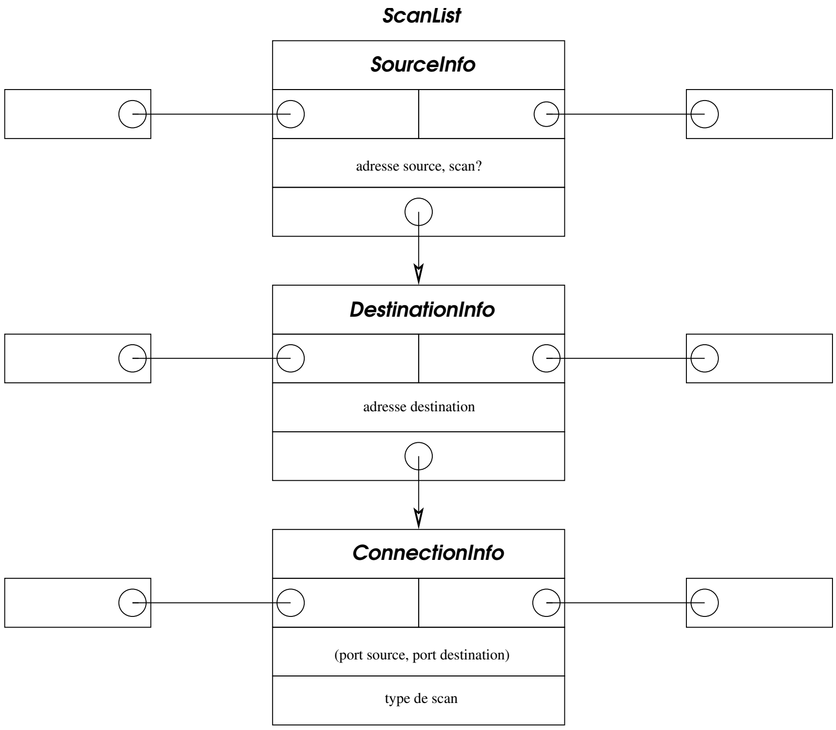


FIG. 4 – *Structure des listes liées.*

```
fi
if (not scan) then break
   else ...
fi
```

Nous savons maintenant que ce scan a quelque chose de suspect, et pourrait donc faire partie d'une attaque. Cependant, afin de ne pas crier trop vite au loup, il faut vérifier que dans notre structure de données, il n'y a pas des connexions qui n'ont plus d'activités intriguantes depuis plus de `maxTime`. On utilise pour cela la fonction

```
void ExpiresConnections (ScanList* scanList, struct spp_timeval watchPeriod,
                         struct spp_timeval currentTime).
```

Après cette étape, notre liste a été réactualisée et le risque de fausse alerte est moins important, on peut alors envoyer le paquet à `int NewScan (ScanList* scanList, Packet* p, ScanType scanType)` qui, en utilisant la liste liée *scanList*, va introduire cette nouvelle connexion dans cette liste et, de cette manière, voir le nombre de connexions qui ont déjà été établies à partir de cette source. Elle renverra donc le nombre de ports déjà «ouverts» entre cette source et une destination. Outre cette information, nous savons si nous sommes en présence d'un *Stealth Scan* (un champ étant prévu à cet effet). Si ce dernier n'est pas détecté, ou que le nombre de ports en communication est inférieur à notre tolérance, nous ne pouvons pas encore dire s'il s'agit d'un véritable scan ou non, et nous continuons donc à traiter les autres paquets. Dans le cas contraire, nous avons de bonnes raisons de penser que cette connexion fait partie intégrante d'une attaque. Nous devons alors regarder si cette source était déjà connue comme hostile, ou non. Si elle n'était pas encore connue, on envoie un message dans le fichier d'alerte, et on marque la source du sceau de l'infamie... Sinon, on passe cette étape.

Ensuite, on parcourt de nouveau *scanList*, et on regarde toutes les sources où on a conclu à un scan. Ensuite, il faut voir depuis quand on n'a plus fait de rapport à propos de cet attaquant. Si le dernier rapport est encore assez récent, on passe à la source suivante. Sinon, il faut voir si elle n'a pas terminé son attaque (plus de connexions en cours). Dans ce cas, on signale la fin du scan dans le fichier log, on retire le *flag* de la détection du scan, afin de permettre que, lors de la prochaine réactualisation de *scanList*, elle puisse être éliminée. Dans le cas contraire, on rapporte le scan dans le fichier log, on ajoute le nombre de connexions en cours au nombre de connexions totales de cette source, et on enlève ces connexions dépassées de notre structure. Et on répète cette opération pour toutes les sources de scans détectés. Grâce à ce système, on a deux points de vue. Un, synthétique, qui nous signalera qu'une telle source a réalisé un portscan sur telle destination, et l'autre, exhaustif, qui donnera le détail de toutes les connexions qui ont constitué ce portscan.

```
1  if (not ip) then break
2     else Actualiser scanList en effaçant toutes les connexions dépassées;
3          Vérifier si ce paquet peut faire partie d'un portscan;
4        if (not scanType) then break
5           else Combien de connexions ont déjà eu lieu entre cette source et cette destination;
6              if (#ports.maxports || stealthscan) then
7                 if (not scan_détecté) then Le signaler dans notre fichier alert
8                    else while (source_scanList) do
9                            if (Time-Out?) then
10                              if (#connections == 0) then
11                                 Signaler la fin du scan et permettre la suppression
12                                 else Mettre information dans log et supprimer les connexions
13                              fi
14                           fi
15                        od
16                fi
17             fi
18       fi
19 fi
```

Nous allons maintenant détailler comment le préprocesseur procède pour «*Vérifier si ce paquet peut faire partie d'un portscan*» (ligne 3 du précédent pseudo-code). Nous utilisons pour cela, comme déjà dit, la fonction `ScanType CheckTCPFlags(u_char th_flags)`. Elle analyse les *flag*s du paquet traité, et selon leur agencement décide si ce paquet pourrait provenir d'une source hostile. La première chose que l'on fait pour éviter un trop grand nombre de false positives, c'est scinder les paquets en deux groupes, ceux qui contiennent le *flag ACK*, et ceux qui ne l'ont pas. En effet, la plupart des port scanners, et en l'occurrence *Nmap*, n'associe le *flag ACK* qu'avec des types bien particuliers de scan (par exemple, XMAS)[9]. Cela nous permet déjà d'éviter un certain nombre de fausses alertes, plutôt que de repérer tout SYN entrant comme étant immédiatement un intrus. Si aucun ACK n'est trouvé parmi les drapeaux, c'est un paquet douteux, on regarde de quel type de scan il pourrait faire partie, et on le signale dans le champ prévu à cet effet.

```
if (ack) then Si c'est une association de flags que nous considérons comme douteuse, signalons le
   else Signaler tout ce qui semble douteux
fi
```

Nous allons maintenant analyser la méthode suivie pour déterminer «*Combien de connexions ont déjà eu lieu entre cette source et cette destination*» (ligne 5). Il faut alors utiliser la fonction

```
int NewScan(ScanList* scanList, Packet* p, ScanType scanType).
```

Il s'agit simplement d'une fonction de parcours de la structure «*ScanList*» qui va comparer, dans un premier temps, la source du paquet avec la source de chaque cellule de la liste. Si elle ne trouve pas de correspondance, elle crée une nouvelle cellule pour cette source, et elle y adjoint toutes les composantes nécessaires. Si, par contre, elle trouve une cellule contenant la source correspondante, il s'agit maintenant de chercher dans la deuxième liste liée (celle des destinations). On réitère alors notre

9. Il fut un temps où le ACK scan était aussi utilisé. Mais le problème qui le rendait efficace a été solutionné, et donc il n'est plus vraiment utilisé.

action : si on ne trouve pas de correspondance, on crée une nouvelle cellule, sinon on «entre» dans cette cellule, et on parcourt la dernière liste, celle des connexions. Et de nouveau, deux options s'offrent à nous : présence de cette connexion, auquel cas on actualise simplement le temps d'arrivée, sinon, on rajoute cette cellule avec les informations dont nous disposons. Donc, maintenant, nous connaissons tout l'historique de la source qui a établi cette connexion, et il suffit d'aller rechercher le nombre de connexions dans le champ de *SourceInfo* prévu à cet effet.

```
while (not (fin de ScanList)) do
   if (sourcepak == sourcecell) then
      while (not (fin de dest_list)) do
         if (dest_pak == dest_cell) then
            while (not (fin de con_list)) do
               if (con_pak == con_cell) then
                  Actualiser le moment de la dernière connexion
                  else Continuer, sauf si on arrive en fin de liste, alors on crée
                       un nouveau noeud, et on se rajoute au nombre de connexions
               fi
            od
            else Continuer, sauf si on arrive en fin de liste, alors on crée
                 un nouveau noeud, et on se rajoute au nombre de connexions
         fi
      od
      else Continuer, sauf si on arrive en fin de liste, alors on crée
           un nouveau noeud, et on se rajoute au nombre de connexions
   fi
od
```

Il faut bien entendu remarquer que «*Continuer, sauf si on arrive en fin de liste, alors on crée un nouveau noeud, et on se rajoute au nombre de connexions*» est différent selon l'étape où il est invoqué parce qu'il faut toujours que tous les champs que nous avons créés soient remplis de façon correcte.

Enfin, nous terminerons par expliquer la méthode utilisée pour «*actualiser scanList en effaçant toutes les connexions dépassées*» (ligne 2). Nous utilisons pour ce faire la fonction

```
void ExpiresConnections(ScanList* scanList, struct spp_timeval watchPeriod,
                        struct spp_timeval currentTime).
```

Comme la fonction précédente, elle ne fait que parcourir la liste liée principale, puis regarde si on a conclu à un scan ou non. Si oui, on ne doit pas perdre d'informations concernant cette source, et donc on passe à la source suivante. Sinon, on analyse chacune des connexions de chaque destination de cette source, et si la dernière activité est plus ancienne que `maxTime`, alors on supprime la connexion (la cellule qui la contient et les informations la concernant dans les champs parents de cette cellule), car on estime qu'un temps suffisant s'est écoulé, ce qui laisse penser que ce ne serait pas une attaque. Cela permet d'éviter un certain nombre de false positives. Une fois que toutes les connexions ont été contrôlées, on doit vérifier qu'une destination ne se retrouve pas sans plus aucune connexion, auquel cas on peut supprimer cette destination. Une fois toutes les destinations analysées, on vérifie que la source a encore au moins une destination, sinon on la supprime.

```
while (source_scanList) do
   if (scanDetected) then continue fi
   while (destination) do
      while (connexion) do
         if (time-out) then Enlever la connexion fi
      od
      if (#connexion == 0) then Enlever la destination fi
   od
   if (#destination == 0) then Enlever la source fi
od
```

7.4 Avantages

La raison principale qui rend cette méthode plus efficace que les précédentes est sans aucun doute l'«entretien» de la fameuse liste liée «scanList». En effet, le problème avec les autres méthodes étaient qu'elles dépendaient fortement d'autres paramètres, parce que l'on ne pouvait dissocier une connexion d'une autre. Ici, chaque connexion est identifiée et classée selon son origine et sa destination, ce qui permettra de déterminer plus facilement si nous sommes en présence d'un scan.

Ainsi, par exemple, si nous recevons dix SYN et que l'on a renvoyé quatre SYN|ACK, les deux méthodes précédentes concluraient simplement à la détection d'un scan, alors qu'ici on peut dire qu'un tel nous a scanné, et qu'un autre non (par exemple pour A, on a reçu 4 SYN et on a renvoyé 4 SYN|ACK, tandis que pour B, on a reçu 6 SYN et on a rien renvoyé, c-à-d 6 connexions sur des ports fermés). On peut aussi éviter des fausses alertes. En effet, prenons un exemple. Si toutes les ouvertures de connexions provenaient de machines différentes, il est fort probable qu'elles soient presque toutes dues à des erreurs, et donc il n'est pas encore nécessaire de signaler que nous avons été scanné. Sans cette structure nous aurions simplement regardé le rapport $\frac{SYN}{SYN/ACK}$, et nous aurions conclu à un scan, tandis qu'ici nous attendrons un peu avant de nous prononcer définitivement.

Un autre avantage de cette technique est que grâce à cette structure de données, on a un fichier spécifique, qui nous donnera, une et une seule fois, l'origine de chaque scan détecté comme tel. Ainsi, nous n'avons pas à parcourir tout le log de *Snort* pour découvrir qui est la source de ce scan.

7.5 Inconvénients

Malheureusement, bien que cette méthode comporte de nombreuses améliorations, elle reste une méthode assez statique, et dépendante de paramètres que nous lui imposerons. Ainsi, nous déterminons le nombre de ports maximums qui peuvent être ouverts dans un certain intervalle de temps. Pour contrer ces paramètres, il suffirait de faire un scan lent, qui dépasserait de peu l'intervalle en question, et le préprocesseur n'y verrait que du feu. Si pour parer ceci, nous décidions d'augmenter l'intervalle de temps, il suffirait à l'attaquant de faire de même. Le problème c'est qu'il arrivera un moment où l'intervalle sera tel que même les connexions tout à fait normales seront signalées comme des attaques. Un scan lent rendra donc le préprocesseur pratiquement obsolète.

Un autre problème que l'on pourrait rencontrer serait la présence de fragmentation lors de cette attaque, que ce préprocesseur ne gère pas. C'est pourquoi il est plus sûr de rajouter une règle dans *Snort* qui prendra ce point de vue en compte (préprocesseur `defrag`).

Enfin, reste le problème des scans coordonnés, c'est-à-dire provenant de multiples machines. Dans ce cas, une nouvelle fois, notre préprocesseur sera inutile. Ainsi si nous reprenons le deuxième exemple dont nous avons parlé dans la section précédente, nous recevons 10 paquets provenant de 10 machines différentes. Nous ne conclurons jamais à un portscan (paradoxalement, une méthode moins sophistiquée détecterait ces connexions comme un scan) alors qu'il est tout à fait possible que ces dix machines soient contrôlées par une seule et même entité.

8 Détection avec SPICE

8.1 Score d'anomalie

Avant toute chose, il importe d'introduire une métrique supplémentaire, qui sera utilisée intensivement ci-dessous. Supposons que la distribution de probabilité du trafic normal du réseau surveillé soit connue (en pratique, elle peut être estimée à partir d'échantillons, mais n'est jamais connue parfaitement). On dispose alors, pour toute combinaison x de port et d'adresse de destination, de la probabilité *P(x)* qu'un paquet du trafic normal possède ces caractéristiques. On peut dès lors donner un *score* *d'anomalie* à tout paquet entrant doté des caractéristiques x qui constitue simplement le logarithme changé de signe de *P(x)*, soit

$$A(x) = -\log(P(x)).$$

Si on considère maintenant tout un scan et non plus un paquet de scan, son empreinte est définie par un ensemble X de caractéristiques individuelles x propres à chaque paquet du scan. On obtient donc une grandeur caractérisant l'anomalie du scan

$$A(X) = \sum_{x \in X} A(x).$$

Plus l'information recherchée par l'attaquant se solde par le scan de combinaisons de ports et adresses IP peu courantes (en regard du trafic normal), plus il devrait être facile de le détecter.

Il est fondamental de comprendre que le score d'anomalie d'un scan est dépendant du réseau surveillé. Si deux réseaux ont exactement la même configuration d'hôtes et de services et qu'ils sont tous deux scannés de façon identique, le score d'anomalie des scans peut être différent si la distribution de probabilité de trafic diffère.

8.2 Avant-propos

Le but ultime de Spice (*Stealthy Probing and Intrusion Correlation Engine*) est de parvenir à détecter un scan alétoire et distribué, mené sur une période de temps très longue sur une cible qui gère une quantité phénoménale de trafic. Comme nous l'avons mis en évidence auparavant, les autres outils de détection sont dans l'impossibilité de détecter de tels scans. Il semble évident qu'il faut oublier le principe de *fenêtre de détection*. Une fenêtre de trop petite taille ne permettra pas la détection de scans lents. Et l'agrandissement de la fenêtre n'est même pas une solution, puisque, vu l'importance du trafic sur la cible, la quantité d'information à gérer (stockage, recherche et performance) deviendra beaucoup trop importante. Il est utopique de sauver toute l'information sur le trafic pendant une périodeconsidérable de temps.

Mais les objectifs d'un attaquant sont bien précis, il veut rassembler de l'information qu'il ne connaît pas déjà. Il n'approche par ailleurs pas sa cible comme un utilisateur normal approche un serveur. Donc, il est certain qu'au moins une partie des paquets utilisés dans le cadre d'un portscan correspondra à du trafic très anormal par rapport à la distribution de trafic «habituelle». Notre approche ne sera donc pas de logger de l'information concernant tous les paquets mais bien de logger de l'information en se basant sur le *score d'anomalie* des paquets. Plus les paquets apparaissent anormaux, plus il faudra les conserver longtemps. Ceci réduit considérablement les problèmes de stockage, de recherche et de performance précédemment évoqués.

Du point de vue de son architecture, Spice est essentiellement constitué de deux parties: un *senseur d'anomalie* et un *corrélateur*. Le senseur est chargé d'affecter un score d'anomalie à tout *événement* (c-à-d à tout paquet). Les événements suffisamment anormaux sont passés, avec leur score d'anomalie, au corrélateur. Le corrélateur est chargé de grouper les événements et de signaler la présence de scans.

8.3 Senseur d'anomalie

Comme nous l'avons déjà signalé plus haut, on attribue un score d'anomalie à un événement en exploitant la probabilité qu'un événement «normal» ressemble à cet événement. Rappelons qu'un événement est caractérisé par les champs d'entête suivants: adresse IP source, adresse IP destination, port source, port destination, protocole, et drapeaux (*flags*). Dans le cadre de l'approche adoptée ici, pour l'affectation d'un score d'anomalie à chaque événement, nous considérerons qu'un événement est caractérisé par des IP source et destination et des ports source et destination. Nous exploiterons les autres caractéristiques (*flags*,...) au niveau du corrélateur. Le senseur se doit de maintenir une ou des

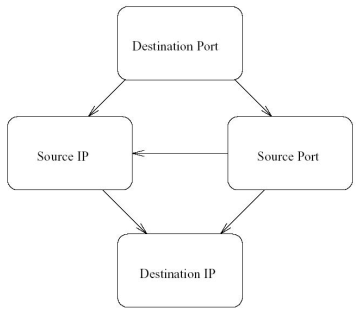


FIG. 5 – *Réseau Bayésien.*

tables de probabilités qui caractérisent la distribution de probabilité du trafic «normal».

La première approche est de maintenir un table de probabilités conjointes *P(destination port, destination IP, source IP, source port).* Il suffira pour un nouvel événement de regarder ses caractéristiques, d'aller chercher dans la table la probabilité correspondante et d'en prendre alors le logarithme changé de signe pour obtenir son score d'anomalie. Ca ne semble pas bien compliqué. La table de probabilités risque cependant d'être de tailleconséquente.

On préférera donc l'approche suivante. Il s'agit de construire une estimation de la probabilité conjointe précédente en se basant sur des probabilités marginales et conditionnelles. Ceci est possible grâce aux réseaux Bayésiens. Pour rappel, un réseau Bayésien est un diagramme qui est particulièrement adapté pour mettre en évidence l'interdépendance de variables. Le réseau Bayésien qui sera utilisé ici est repris à la figure (5). La règle générale d'application dans un réseau Bayésien est la suivante : un noeud est indépendant de tous ses non-descendants, à partir du moment où ses parents directs sont connus. Le réseau Bayésien de la figure (5) nous apprend donc que l'adresse IP de destination est indépendante du port destination si l'adresse IP source et le port source sont connus. On peut sans mal obtenir, à partir du réseau Bayésien, la relation suivante

$$P(Dport,DIP,Sport,SIP) = P(Dport)P(Sport|Dport)P(SIP|Sport,Dport)P(DIP|SIP,Sport)$$

Nous allons maintenant détailler la procédure suivie pour trouver que la relation entre les quatre caractéristiques de tout événement est bien celle exprimée par le réseau Bayésien. Cela se fait par l'intermédiaire de calculs d'entropie sur le trafic réel observé pendant plusieurs semaines (calculs d'entropie sur un bon million de paquets TCP SYN). On a par exemple pu montrer que $H(DIP|SIP,Sport)$, qui traduit l'incertitude résiduelle sur l'adresse IP de destination quand on connaît l'adresse IP source et le port source, était très faible, et presque égale à $H(DIP|SIP,Sport,Dport)$. Ce qui confirme ce que met en évidence notre réseau Bayésien à savoir que l'adresse IP de destination est indépendante du port de destination, si on connaît l'IP source et le port source. Les autres relations mises en évidence par le réseau Bayésien peuvent également être confirmées par les autres calculs d'entropie.

8.4 Corrélateur

Les événements qui ont un score d'anomalie plus élevé qu'un certain seuil sont passés par le senseur au corrélateur, qui va se charger de grouper les événements.

On désire faire usage d'heuristiques pour déterminer à quel groupe un événement appartient. Le problème est que l'on ne sait pas **à priori** quelles heuristiques sont utiles pour rassembler les paquets d'un scan. Pour chaque scan, des heuristiques différentes devraient être utilisées. Pour l'un, le fait que les champs ID des paquets IP sont les mêmes sera très utile, pour l'autre, cela pourrait n'avoir aucun intérêt.

L'autre problème qui se présente à nous est que l'on ne sait pas dans quel ordre et à quelle cadence les événements à grouper arriveront.

8.4.1 Graphe de corrélation

La métaphore qui motive notre approche est la suivante: les événements à corréler sont comme des atomes dans l'univers. Chaque heuristique traduit l'énergie de liaison entre atomes. On crée des liens entre ces événements quand l'attraction entre ceux-ci est la plus grande (quand l'énergie de liaison est la plus grande). Le *graphe de corrélation* est fait de noeuds qui correspondent aux événements et de liens entre ces noeuds qui traduisent la correlation entre ceux-ci. Donc, deux événements (noeuds) sont reliés dans le graphe s'il existe une correlation forte entre ceux-ci. A ce lien (connexion) est associé un certain poids qui traduit le degré de corrélation.

Des groupes d'événements correspondent dans le graphe de corrélation à des sous-graphes dans lesquels toutes les connexions sont étiquetées d'un poids supérieur à une certaine valeur. Les groupes sont les sous-graphes connectés qui persitent quand les liens plus faibles qu'un certain seuil sont supprimés.

8.4.2 Fonction d'évaluation

La force des liens entre événements est évaluée par l'intermédiaire d'une fonction d'évaluation. Celle-ci a la forme suivante

$$f(e_1,e_2) = c_1 h_1(e_1,e_2) + c_2 h_2(e_1,e_2) + \ldots + c_k h_k(e_1,e_2)$$

où e_1 et e_2 sont les événements dont on évalue la corrélation, c_i $(i = 1,\ldots,k)$ sont des constantes, et h_i $(i = 1,\ldots,k)$ sont des fonctions d'évaluation heuristique. Les fonctions d'évaluation heuristique traduisent le niveau de corrélation selon différents critères, et n'ont qu'une seule contrainte à respecter: produire un résultat entre 0 et 1 où 0 indique que l'heuristique n'est pas parvenu à trouver une quelconque corrélation entre les événements sur le dit critère, et 1 traduit l'existence de la plus forte corrélation entre les deux événements sur ledit critère. Il va sans dire qu'il faudrait prendre en considération au moins les heuristiques suivantes

- *Heuristique d'égalité de caractéristiques* – L'adresse IP source est-elle commune à plusieurs événements? Ou serait-ce le port destination? Ou le réseau de destination? Si c'est le cas, valeur 1. Sinon, valeur 0.
- *Heuristique de proximité de caractéristiques* – Dans quelle mesure les événements sont-ils temporellement proches? Les IP de destinations sont-elles proches les unes des autres? Plus on a cette proximité (temporelle, spatiale,...), plus la valeur de cette heuristique de proximité approche 1.
- *Heuristique d'éloignement de caractéristiques* – Cette heuristique tente de reconnaître des «trous classiques» entre les valeurs des caractéristiques des événements. Ces «trous classiques» peuvent correspondre par exemple à un délai d'une heure entre les événements, ou à un incrément systématique de un dans les adresses IP de destination.
- *Heuristique de covariance de caractéristiques* – Il s'agit de reconnaître des caractéristiques d'événements qui varient ensembles. Par exemple, cette heuristique est capable de détecter que le port source varie à la même vitesse que l'adresse IP de destination.

Quant aux constantes c_i $(i = 1,\ldots,k)$, elles permettront de pondérer l'importance apportée aux résultats des différentes heuristiques choisies.

8.4.3 Ajout d'événements dans le graphe de corrélation

Quand un nouvel événement est présenté au corrélateur, il doit être ajouté d'une manière ou d'une autre au graphe de corrélation. Il peut y avoir un très grand nombre (peut-être des milliers) d'événements dans le graphe de telle sorte qu'il est littéralement utopique de tester chaque nouvel événement avec tous ceux déjà présents.

Le nouveau noeud se voit affecter un partenaire de liaison (noeud courant) de façon alétoire. On évalue la force de liaison (par l'intermédiaire de la fonction d'évaluation f) entre le noeud courant et le nouveau noeud, on obtient la valeur *f(new_node,current_node)*. Un successeur possible est choisi aléatoirement parmi les successeurs du noeud courant. Si la fonction d'évaluation appliquée au nouveau noeud et au successeur possible donne une valeur plus grande que *f(new_node,current_node)*, le successeur possible devient le successeur effectif (noeud courant) et le nouveau noeud se retrouve à présent lié à celui-ci. Dans le cas contraire (valeur plus petite que *f(new_node,current_node)*), le successeur possible n'est choisi comme successeur qu'avec une probabilité P (qui diminue plus la valeur en question est petite par rapport à *f(new_node,current_node)*), sinon, l'état courant est conservé. Et le processus est réitéré de la sorte autant de fois qu'il faut.

On se référera utilement à la figure (6). Elle décrit quelques étapes de l'ajout d'un nouvel événement e_{100}. On lui donne d'abord un partenaire de liaison alétoire e_{17}. La fonction d'évaluation appliquée à ces deux événements donne une valeur de 1.7. e_{22} est choisi parmi les voisins de e_{17} (c'est d'ailleurs son seul voisin), et vu que la fonction d'évaluation entre e_{100} et e_{22} donne une valeur de 4.0, e_{22} est choisi comme noeud courant (e_{100} lui est donc lié). e_5 est choisi aléatoirement parmi les voisins de e_{22}. La fonction d'évaluation appliquée à e_5 et e_{100} donne une valeur de 2.1, ce qui est plus petit que quatre. Mais la probabilité P fait ici que e_5 est quand même choisi comme noeud courant. e_{74} est choisi alétoirement parmi les voisins e_5, la fonction d'évaluation appliquée à e_{74} et e_{100} donne une valeur inférieure à 2.1. e_{74} n'a pas autant de chance que e_5 et ne profite pas de la probabilité P, donc e_5 reste le noeud courant. On poursuivra sans mal le raisonnement pour le reste des figures.

Les liens faibles dans le graphe non nécessaires pour maintenir la connectivité sont éliminés par un processus de faible priorité.

8.4.4 Retrait d'événements du graphe de corrélation

Tous les événements doivent un jour ou l'autre disparaître du graphe. Le temps pendant lequel ils restent dans le graphe dépend de leur score d'anomalie, mais aussi des autres événements du groupe dont ils font partie. Le temps de vie d'un événement individuel est proportionnel à son score d'anomalie. Celui-ci est ajouté au temps auquel l'événement est né (le paquet est arrivé) pour obtenir le temps auquel il faudra le détruire (*scheduled timeout*). Cependant, les événements d'un groupe héritent du *timeout* le plus éloigné dans le temps parmi les *timeout*s des paquets du même groupe. Ceci, afin de conserver les événements qui font partie d'un groupe «actif».

8.4.5 Alertes

Le score d'anomalie d'un groupe d'événements est la somme des scores d'anomalie des événements du groupe. Si le score d'anomalie d'un groupe dépasse un certain seuil, une alerte de port scan est déclenchée.

8.5 Implémentation

Il va sans dire que les concepts développés ci-dessus ont donné lieu à une implémentation logicielle. Seul le senseur d'anomalie est à présent fonctionnel. Il s'appelle *Spade*, pour Statistical Packet Anomaly Detection Engine. Il s'agit d'un préprocesseur pour Snort. Ce qui a permis à ses auteurs d'exploiter les facilités offertes par Snort. A l'heure actuelle, Spade ne regarde que les segments TCP syn, qui

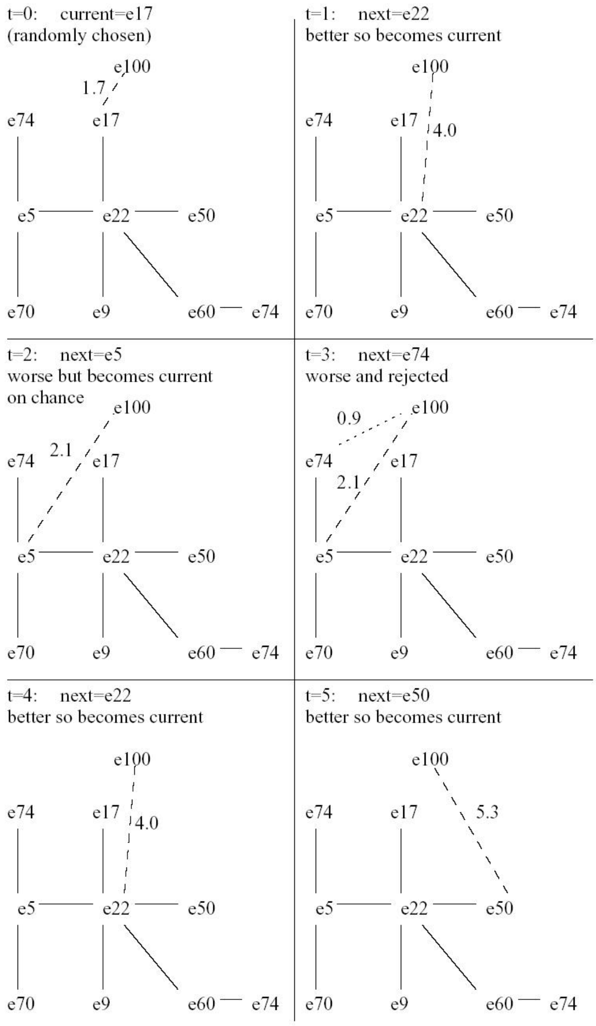


FIG. 6 – *Ajout d'événements dans le graphe de corrélation*

constituent sans aucun doute les plus intéressants. De par sa conception, il pourrait sans mal gérer d'autres types de paquets.

8.5.1 Caractéristiques de Spade

Spade a un certain nombre de caractéristiques qui peuvent être activées et configurées par l'intermédiaire des fichiers de configuration de Snort. Il offre à l'utilisateur quatre alternatives d'évaluation des probabilités des paquets. Les voici:

1. Utilisation du raison Bayésien de la figure (5).
2. Utilisation de la probabilité conjointe *P(SIP, Sport, DIP, Dport)*
3. Utilisation de la probabilité conjointe *P(SIP, DIP, Dport)*
4. Utilisation de la probabilité conjointe *P(DIP, Dport)*

L'utilisateur peut aussi demander à Spade de ne monitorer que les paquets de certains réseaux. Ceci permet à Spade de focaliser son évaluation sur un trafic plus spécifique, réduisant par là-même le bruit lié à du trafic «étranger» (qui a typiquement un beaucoup plus grand domaine d'adresses et de ports possibles).

Puisque Spade calcule toute une série d'information concernant le réseau sur lequel tourne Snort, il est intéressant que cette information soit conservée. Chaque fois qu'il est démarré, il reprend toutes les informations concernant le réseau dans un fichier spécifié. Périodiquement pendant son fonctionnement, il y sauve l'état du réseau (il sauve également automatiquement sur réception d'un signal ou quand Snort est arrêté).

Le seuil de signalisation d'anomalie (*anomalous event reporting treshold*) est un paramètre important passé au préprocesseur Spade. Malheureusement, sa valeur idéale est très dépendante des caractéristiques du réseau surveillé (et du temps). Un seuil de valeur trop élevée peut entraîner que des événements pourtant intéressants ne soient pas signalés. Dans le cas d'un seuil de trop faible valeur, l'utilisateur peut se retrouver submergé d'alertes inintéressantes. Les concepteurs de Spade ont cependant mis en oeuvre plusieurs techniques qui lui permettent d'ajuster dynamiquement la valeur du seuil au trafic observé.

Remarquons enfin qu'il est possible de demander à Spade d'afficher les entropies et les probabilités conditionnelles calculées. C'est ce qui permet notamment de constater les relations représentées par le réseau Bayésien de la figure (5).

8.5.2 Gestion des probabilités calculées

Selon le mode d'évaluation des probabilités choisi par l'utilisateur, Spade doit conserver un certain nombre de probabilités conjointes concernant les caractéristiques (SIP, DIP, Sport, Dport) des paquets. Notons que les probabilités conditionnelles requises pour les calculs du réseau Bayésien peuvent être dérivées de probabilités non conditionnelles. Pour toute caractéristique A dont on désire obtenir la probabilité *P(A)*, il y aura une table reprenant, pour chaque valeur de A, sa fréquence d'occurrence. Pour déterminer la probabilité d'une valeur particulière de A, sa fréquence d'occurrence est divisée par le nombre total d'événements enregistrés dans la table de A. Si on désire connaître la probabilité de l'occurrence conjointe d'une valeur de la caractéristique A et d'une valeur d'une autre caractéristique B, on a besoin d'une table à deux dimensions. A chaque entrée de la table, on enregistre une probabilité conjointe d'occurrence pour les deux valeurs concernées.

Bien sûr, on peut se demander comment représenter efficacement ces tables. La nature du trafic du réseau influence cela. Certaines valeurs de caractéristiques sont bien plus courantes que d'autres (le Dport 80 est beaucoup plus fréquent que le 5037, par exemple). Les concepteurs de *Spade* ont opté pour une structure de données de la forme d'un arbre binaire balancé. Cette structure permet un accès très rapide aux cas classiques (serveurs principaux et ports les plus populaires). Elle est par ailleurs en mesure de gérer une croissance très importante du nombre d'entrées (un *flood scan* conçu délibérément pour remplir la structure de données avec toutes les combinaisons possibles ne doit pas mettre à terre Spade).

8.5.3 Corrélateur

Comme nous l'avons déjà signalé, le corrélateur est toujours au stade du développement. Les grandes bases de son fonctionnement sont déjà en place.

Le corrélateur constitue un processus séparé qui peut tourner sur une machine distante. La communication entre le détecteur d'anomalies et le corrélateur se fait via des sockets. Deux raisons à cette séparation en deux processus. La première évite à Snort de passer trop de temps à traiter chaque paquet, ce qui pourrait l'amener à jeter des paquets sans les examiner. Par ailleurs, cela donne plus de liberté au corrélateur, qui peut effectuer beaucoup plus de calculs concernant les événements anormaux que lui a passés le senseur d'anomalie.

L'approche utilisée pour son implémentation est une approche *multi-thread*. Les différents threads sont: un thread pour recevoir les signaux concernant des événements anormaux dans un file d'attente, des threads pour ajouter des événements de la file d'attente au graphe de corrélation et signaler la présence de scans, un thread pour «nettoyer» le graphe (enlever les liens de poids faible), un thread pour enlever des événements du graphe, un thread pour répondre aux questions concernant l'état du corrélateur et un thread de réception de commandes pour ajuster des paramètres opérationnels dynamiquement.

8.6 Conclusion

Spice devrait également se montrer utile pour la détection de la propagation de vers (*worms*) et de l'usage de dénis de service. Tout comme les port scans, ces applications impliquent un grand nombre de connexions ou de paquets avec des structures similaires qui seront typiquement assez anormaux par rapport au trafic normal. La seule différence par rapport à la détection de port scans consistera à changer certaines heuristiques. Spice devrait donc être capable de détecter ces usages douteux du réseau sans d'abord devoir procéder au «reverse engineering» de l'outil servant à mettre en place le vers particulier ou le déni de service.

8.7 Avantages et Inconvénients

Avantages:

1. Spice ne logge pas tous les paquets, seuls les paquets anormaux sont enregistrés.
2. Structures de données très efficaces et pensées pour résister aux *flood scan*.
3. *Détection de scans lents*. Rappelons qu'un paquet ne se voit affecter un score d'anomalie que sur base des caractéristiques SIP, DIP, Sport et Dport et non sur base du protocole, des flags,... Un paquet d'un scan lent sera passé au corrélateur s'il est assez rare (combinaison de DIP, SIP, Dport et Sport peu courante). C'est au niveau du corrélateur qu'un certain nombre de mécanismes importants sont mis en oeuvre pour la détection du scan lent. Un paquet anormal issu d'un scan lent sera conservé longtemps, et une fois qu'un autre du même scan sera reçu, le corrélateur n'aura pas de mal à faire le rapprochement (grâce à des heuristiques appropriées). Comme il fut dit, l'heure programmée de destruction du premier paquet va être reportée à celle du nouveau paquet arrivé. Les heuristiques utilisées vont faire apparaître des liens très forts entre les paquets du scan lent (même IP source, même intervalle de temps entre les paquets, ports destinations rares, même IP destination,...) . Un sous-graphe de poids fort ne tardera pas à se constituer.
   Le préprocesseur de portscans de Snort, lui, se contente de regarder s'il reçoit *maxPorts* paquets en *maxTime* secondes. Et il n'est pas question de lui mettre un *maxTime* de plusieurs heures,

parce que ses structures de données exploseront. Ici, Spice est beaucoup plus sélectif concernant les paquets. Il ne retient que les paquets anormaux aussi longtemps qu'il faut.

4. *Scans coordonnés.* Spice devrait se montrer performant quand il s'agit de détecter un scan issu de plusieurs machines qui scannent des services différents à des moments différents. Les paquets du scans seront assez vraisemblablement classés comme anormaux, parce qu'ils sont issus de ports peu communs, d'adresses IP très rares dans le trafic normal, et visent des ports tout aussi peu courants. Donc admettons que les différents paquets du scan coordonnés soient passés au corrélateur. C'est à celui-ci que revient la responsabilité de les regrouper au sein d'un même sous-graphe. Ici, ce sont clairement les heuristiques qui devront être puissantes, de manière à mettre en évidence les points communs entre les paquets du scan coordonné (même réseau de destination voire même même machine de destination, même protocole, même flags,. . .). L'avenir nous dira si Spice parvient effectivement à détecter le scan coordonné le plus pernicieux et le mieux mené. Acceptons le fait que Spice dispose d'un potentiel certain de détection grâce à son architecture remarquable.
5. Séparation du senseur d'anomalie et du corrélateur, ce qui donne une beaucoup plus grande liberté au corrélateur. Pas de risques de ralentir Snort.

Inconvénients:

Travail de grande ampleur, mais sans aucun doute une approche remarquable pour la détection de scans.

Références

[1] Fyodor, **Détection d'OS par prise d'empreinte de pile TCP/IP**
http://membres.tripod.fr/goer_le_renard/empreinteTCPIP.html

[2] Postech Laboratory for Unix Security, **ICMP Usage in Scanning**
http://www.plus.or.kr/seminar/icmp.ppt

[3] Lawrence Teo, **Network Probes Explained: Understanding Port Scans and Ping Sweeps**
http://www.linuxjournal.com/article.php?sid=4234

[4] Rik Farrow, **System Fingerprinting with NMAP**
http://www.networkmagazine.com/article/NMG20001102S0005

[5] Ofir Arkin, **Network Scanning Techniques**
http://www.sys-security.com. November 99

[6] Fyodor et Ouah, **The Art of Port Scanning**
http://www.insecure.org. Octobre 2000

[7] Marco de Vivo, Eddy Carrasco, Germinal Isern et Gabriela O. de Vivo, **A Review of Port Scanning Techniques**.
http://www.cert.org/advisories/CA-1997-27.html

[8] **NMAP network security scanner**, man

[9] dethy@synnergy.net, **Examination des méthodes de scan port - Analyse des Techniques d'Audit**
http://www.synnergy.net/downloads/papers/portscan-fr.html

[10] Onathon Giffin, **Portscans**
http://www.cs.wisc.edu/~giffin/portscans.ppt. April 2001

[11] Kathryn Kerr, **Malicious Scanning**
http://rr.sans.org/audit/mal_scan.php. March 2001

[12] SnakeByte, **Scanning ideas to defeat Intrusion Detection Systems**
http://www.kryptocrew.de/snakebyte/e/Scanning.txt

[13] Nicolas Justin, **Etude sur les Port Scans et Elaboration d'un Outils de Détection**
http://nicolas.justin.free.fr/. Février 2000

[14] Elie aka Buerztein, **Le scanning par témoin**
http://www.bursztein.net/secu/temoin.html. Mai 2001

[15] S. Sanfilippo (Antirez), **New TCP scan Method**
Bugtraq mailing list archives. Dec 1998

[16] Stuart Staniford, James A. Hoagland, Loseph M. McAlerney, **Practical Atomated Detection of Stealthy Portscans**
http://www.silicondefense.com/