Détection des systèmes d'exploitation avec Cron-OS

Olivier Courtay, Olivier Heen, and Franck Veysset

olivier.courtay@enst-bretagne.fr, École Nationale Supérieure des Télécom.
olivier.heen@thomson.net, Thomson Recherche et Innovation
franck.veysset@francetelecom.com, France Télécom Recherche et Développement

Résumé Après un survol du domaine de la reconnaissance à distance des systèmes d'exploitation, nous rappelons le principe de la reconnaissance temporelle mis en œuvre dans l'outil RING. Nous présentons ensuite une évolution, Cron-OS, et indiquons divers exemples de fonctionnement.

Mots-clé *OSFP*, Congestion *TCP/IP*, *RTT*, *RTO*.

Introduction

Détecter à distance les systèmes d'exploitation de machines en réseau peut s'avérer utile pour :

- Qualifier un parc, y détecter des systèmes inhabituels.
- Conduire des audits de sécurité, éventuellement automatisés.
- Préparer des attaques spécifiques de certains systèmes d'exploitation.
- Alimenter des bases de données à des fins statistiques.
- Mieux connaître la source d'une attaque en cours.
- Abaisser la priorité d'une alerte IDS lorsque le système d'exploitation n'est, en fait, pas sensible à l'attaque en cours [6].
- Concevoir des leurres simulant précisément le comportement de systèmes d'exploitation spécifiques.

Cet article est essentiellement consacré à la présentation de Cron-OS. La première section situe le domaine de la détection à distance des systèmes d'exploitation[1] et en indique les principales difficultés. La deuxième section décrit les principes de l'analyse temporelle tels qu'ils sont mis en œuvre dans RING et Cron-OS. La troisième section montre des cas concrets d'utilisation des deux outils. Quelques pistes pour les évolutions de Cron-OS sont données en conclusion.

1 Détection des systèmes d'exploitation

1.1 Principes

Les systèmes d'exploitation sont reconnus au travers de leurs propriétés caractéristiques. Dans le cas de la détection à distance, les propriétés en question sont liées à des comportements réseau (cf. figure 1).

[1] Par la suite, la notation *OSFP* pour *Operating System FingerPrinting* est utilisée.

Fig. 1. Principe général de l'*OSFP*. Lorsqu'une même entité génère les stimuli et mesure les réactions on parle d'*OSFP* actif.

Les spécifications de protocoles comme ICMP, UDP ou TCP laissent une grande liberté d'implémentation. Chaque éditeur interprète, optimise et paramètre : autant d'indices qui vont ultérieurement *signer* la pile. Les caractéristiques les plus révélatrices d'une pile IP particulière se nichent préférentiellement dans des portions de code :

- Rarement exécutées comme pour la gestion d'erreurs ou la prise en charge de cas non conformes aux RFC.
- Complexes comme pour la gestion de congestion.
- Très optimisées comme pour l'assemblage de paquets fragmentés et la génération aléatoire de numéros de séquence.
- Mal programmées, sans contrôle de la taille des mémoires tampon.

Un grand nombre de techniques ont vu le jour, reprenant toutes les grandes lignes indiquées figure 2. L'état d'avancement des techniques varie de la simple publication pour l'analyse de la répartition des numéros de séquence [11], au logiciel éprouvé et maintenu dans le cas de `NMAP` [4], en passant par le démonstrateur pour `TBIT` [7], `RING` [9], `Cron-OS` [10] et le démonstrateur avancé pour `XPROBE2` [1].

1.2 Difficultés

Dans le cas particulier de la détection active, le schéma de principe laisse présager de nombreuses difficultés :

- Tous les stimuli envoyés sont-ils bien reçus et traités par l'objet de la mesure ?
- L'ordre des stimuli à-t-il une incidence sur l'objet de la mesure ? Sur les composants intermédiaires ?
- Toutes les réactions sont-elles correctement perçues et mesurées ?

Fig. 2. Arbre des techniques d'*OSFP* étiqueté par quelques outils.

– Les données de référence sont-elles exactes et en nombre suffisant ?
– Quel sont les effets du médium et des composants intermédiaires ?

Sensibilité aux conditions de test

La qualité d'une détection peut varier grandement selon qu'elle a lieu en environnement LAN (filaire ou non), ou WAN en présence de routeurs, de pare-feu, voire d'IDS actifs. La charge réseau peut également influencer les mesures. le cas de `RING`.

L'outil `NMAP`, par exemple, peut être très perturbé par la présence d'un pare-feu. En effet il effectue quelques tests sur des ports fermés, ce sont les tests `TCP Xmas (F/P/U)`, `TCP SYN`, `TCP ACK` et `UDP` avec réponse ICMP. Les résultats ne sont pas les mêmes selon que le pare-feu met en œuvre des règles de type `reject` ou bien `drop` (cf. table 1).

Perturbations induites

Certains stimuli peuvent placer la pile dans une situation d'erreur. Ce dernier point est bien connu et utilisé pour des attaques de type déni de service. Pour mémoire, voici quelques exemples historiques :

– Le *Ping of Death* et ses nombreuses variantes où la pile ne contrôle pas la taille du tampon pour les paquets ICMP `echo-request`.
– Certaines attaques comme `WinNuke` ou `Teardrop`, où la pile gère mal une conjonction, absurde au sens des RFC, du drapeau `URG` et du pointeur `OOB` dans un même paquet `SYN`.
– Certains pare-feu avec tables d'état, très perturbés par des paquets `NMAP` sortant du réseau interne.

```
Agressive OS guesses : Win2000, WinXP

OS Fingerprint :
TSeq(Class=RI%gcd=1%SI=29D09%IPID=RD...
T1(Resp=Y%DF=Y%W=402E%ACK=S++%Flags...
T2(Resp=Y%DF=N%W=0%ACK=S%Flags=AR%O...
T3(Resp=Y%DF=Y%W=402E%ACK=S++%Flags...
T4(Resp=Y%DF=N%W=0%ACK=O%Flags...
T5(DF=N%W=0%ACK=S++%Flags=AR%Ops=)
T6(DF=N%W=0%ACK=O%Flags=R%Ops=)
T7(DF=N%W=0%ACK=S++%Flags=AR%Ops=)
PU(DF=N%TOS=0%IPLEN=38%RIPTL=148%
RID=E%RIPCK=E%UCK=E%ULEN=134%DAT=E)
```

```
Aggressive OS guesses : FreeBSD2.2.1-4.1
(90%),Windows Me,Win2000 or XP (86%), ...
TCP/IP fingerprint :
TSeq(Class=RI%gcd=1%SI=6F65%IPID=I%TS=U)
T1(Resp=Y%DF=Y%W=FFFF%ACK=S++%Flags...
T2(Resp=N)
T3(Resp=Y%DF=Y%W=FFFF%ACK=S++%Flags...
T4(Resp=Y%DF=N%W=0%ACK=O%Flags...
T5(Resp=N)
T6(Resp=N)
T7(Resp=N)
PU(Resp=N)
```

Tab. 1. Effet d'un pare-feu sur la détection d'un Microsoft Windows 2000 SP1 avec NMAP (`# nmap -O --osscan-guess -n -P0 -p 139,1234 x.x.x.x`). À gauche le résultat correct sans pare-feu, à droite le résultat erroné en présence d'un pare-feuoù les ports fermés sont protégés par des règles `drop`.

Une méthode primaire et terriblement "bruyante" d'*OSFP* consisterait à envoyer successivement des tentatives de blocage spécifiques de chaque système d'exploitation. Si le détecteur est capable de vérifier l'état de disponibilité de la cible après chaque tentative, il peut obtenir une mesure du système d'exploitation.

A contrario, les méthodes qui respectent les RFC induisent moins de perturbation sur les cibles et les composants intermédiaires. Elles seront donc préférées par des auditeurs, des administrateurs ou même des systèmes de détection automatique. Lorsque la détection est utilisée comme prélude à une attaque, cette discretion peut s'avérer ennuyeuse pour les pare-feu et les IDS chargés de bloquer ou de repérer les signes avant coureurs (cf. 2.3 pour le cas de `Cron-OS`).

Apprentissage

Une fois correctement induites et mesurées, les réactions caractéristiques doivent encore être comparées à une base de référence. Plusieurs cas peuvent se produire, parmi lesquels quelques cas limites :

- Le système n'est pas dans la base, mais l'algorithme de correspondance confond avec un système proche. C'est souvent le cas avec des systèmes issus de souches BSD (très utilisées).
- Le système est dans la base mais l'algorithme de correspondance ne parvient pas à établir l'identité. C'est souvent le cas lorsque les caractéristiques sont trop peu discriminantes ou lorsque les résultats de mesure sont incomplèts.

L'obtention de valeurs de référence pose des problèmes de confiance dans les sources d'information, dans les conditions de mesure au moment de la prise de

référence (elles ne sont pas toujours connues) et de validité dans un environnement réel : certaines références correctes en LAN non filtré sont inutilisables en WAN.

*En résumé : une bonne méthode d'*OSFP *suppose une bonne base de connaissances, un jeu de tests discriminants et correctement ordonnés, un minimum de robustesse par rapport aux fluctuations du réseau et un certain respect des RFC pour minimiser les perturbations et être efficace.*

2 Principes de `RING` et de `Cron-OS`

2.1 Principe de base

Le protocole *TCP/IP* est *full duplex* et connecté, en ce sens qu'il nécessite un accord préalable entre les machines désirant communiquer (cf. figure 3). Il assure également la distribution ordonnée et sans erreur des paquets dès la première étape de la connexion. Pour ce faire, *TCP/IP* met en place des mécanismes sophistiqués de retransmission des paquets perdus, d'acquittement, de gestion de congestion, d'adaptation des tailles de fenêtres, etc.

Fig. 3. Établissement d'une connexion *TCP/IP* (*3-way handshake*) et évolution des états internes du serveur.

La gestion de congestion

Lorsqu'un paquet *TCP/IP* est perdu, le protocole réagit aussitôt comme s'il y avait congestion du réseau : il ralentit le débit, tout en provoquant la

retransmission des paquets perdus afin de maintenir l'intégrité du flux. Pour ne pas entretenir ou aggraver la congestion, les paquets doivent être retransmis avec une cadence de plus en plus lente tant que la communication n'est pas rétablie.

Le protocole *TCP/IP* doit maîtriser deux valeurs importantes :

- Le temps moyen d'aller/retour d'un paquet appelé *RTT*(*Roud Trip Time*).
- La temporisation associée à chaque segment non acquitté, appelée *RTO* (*Retransmission Time Out*). La valeur du *RTO* est dépendante de celle du *RTT*.

Pour approximer ces valeurs, les implémentations ont rapidement utilisé une moyenne lissée. Plus précisément, la RFC 793 *Transmission Control Protocol* suggère les méthodes de calcul suivantes :

```
Measure the elapsed time between sending a data octet with a particular
sequence number and receiving an acknowledgment that covers that sequence
number (segments sent do not have to match segments received).
This measured elapsed time is the Round Trip Time (RTT).  Next compute a
Smoothed Round Trip Time (SRTT) as:
                  SRTT = ( ALPHA * SRTT ) + ((1-ALPHA) * RTT)
and based on this, compute the retransmission timeout (RTO) as:
                   RTO = min[UBOUND,max[LBOUND,(BETA*SRTT)]]
where UBOUND is an upper bound on the timeout (e.g., 1 minute), LBOUND is
a lower bound on the timeout (e.g., 1 second), ALPHA is a smoothing factor
(e.g., .8 to .9), and BETA is a delay variance factor (e.g., 1.3 to 2.0).
```

P. Karn [5] a proposé quelques heuristiques maintenant adoptées dans les implémentations courantes de *TCP/IP* :

- En cas de retransmission de paquet, le *RTT* n'est pas mesuré (pour éviter toute ambiguïté sur la mesure).
- Le *RTO* est alors doublé *RTO* = `old_RTO` * 2.
- Si `old_RTO` n'est pas connu (lors du démarrage), alors *RTO* vaut 6 secondes.

N.B. : croire à la congestion dès qu'un paquet est perdu n'est pas toujours une bonne chose. Dans les réseaux sans fil la perte d'un paquet n'est pas obligatoirement synonyme de congestion. Il se peut également que l'une des deux machines, le serveur par exemple, soit persuadé qu'il y a congestion, alors qu'en fait le client ignore délibérément les messages qu'il reçoit (cf. 2.1).

Fonctionnement de `RING`

Pour tenter une reconnaissance à distance, `RING` nécessite un (seul) port *TCP/IP* ouvert. Il crée délibérément une situation que le serveur interprète comme une congestion. Ce dernier retransmet alors des paquets, conformément à son algorithme de gestion de congestion et à ses valeurs de paramètres. La machine de mesure calcule la suite des délais de retransmission, puis la confronte à une base de référence pour déduire les caractéristiques de la pile *TCP/IP* du serveur. Une fois ces caractéristiques connues, il est souvent possible d'en déduire le système d'exploitation, sa version, et dans certains cas la sous-version ou le niveau de *patch*.

Fig. 4. Schéma de principe de `RING`.

Pour la première version de `RING`, la congestion est simulée dès la première étape du *3-way handshake* (cf. figure 4), en ignorant au niveau du client tous les paquets `SYN ACK` renvoyés par le serveur. Dans ce cas particulier, à la fois la RFC 2988 et la pratique montrent qu'un même serveur réagit toujours de la même manière.

2.2 Extensions conduisant à `Cron-OS`

Dès qu'un protocole spécifie des délais entre des évènements déclenchés par l'objet de la mesure, et dès que ces délais varient d'une implémentation à l'autre, le principe utilisé dans `RING` peut être appliqué. Cette généralisation est très vaste et peut même concerner des protocoles comme SSL ou le fonctionnement d'applicatifs, ce qui peut conduire à de la reconnaissance de services. Dans le cas de *TCP/IP*, l'état `SYN RCVD` n'est pas le seul pour lequel le serveur doit renvoyer une suite de paquets avec délais.

Autres états analysables

Lors de la fermeture d'une connexion, plusieurs phénomènes peuvent se produire. La figure 5 montre le dialogue lorsque le client décide de clore une connexion. Il émet tout d'abord un paquet `FIN`. Le serveur passe alors dans l'état `CLOSE WAIT`, accuse réception du paquet en émettant un `ACK`, puis ferme sa demi-connexion en envoyant un `FIN` au client. Si ce `FIN` n'arrive pas, le serveur croit à une congestion réseau et retransmet des `FIN` avec des intervalles d'attente de plus en plus longs pour ne pas aggraver la congestion. Les temps d'attente peuvent être mesurés et, puisqu'ils sont fortement dépendants des implémentations, comparés à une base de référence pour tenter une détection du système d'exploitation.

Fig. 5. Terminaison à l'initiative du client, avec congestion simulée.

La figure 6 montre un autre cas de terminaison, plus difficile à obtenir, où c'est le serveur qui décide de terminer la connexion. Ce cas se produit par exemple après un silence prolongé du client, la durée variant selon l'application utilisant la connexion *TCP/IP* côté serveur. Avec certaines applications, il est possible de forcer cet état ; c'est le cas pour les serveurs `HTTP` auxquels il suffit d'envoyer une requête `GET␣/␣HTTP/1.0\r\n\r\n`.

Dès que l'application décide de clore la connexion, le serveur passe dans l'état `FIN WAIT 1` et émet un paquet `FIN` pour informer le client. Si celui-ci ne répond pas, le serveur conclut à la congestion, et donc retransmet. Ce comportement autorise le même type de mesures que dans les cas précédents.

Fonctionnement de `Cron-OS`

L'outil `Cron-OS` mesure les délais liés aux trois états `SYN-RCVD` (comme `RING`), `CLOSE WAIT` et `FIN WAIT 1`. Pour les deux derniers états, la mesure est délicate car elle peut dépendre du passé de la connexion et de la manière dont le client et le serveur ont ajusté leurs valeurs de *RTT* et de *RTO*. En effet, afin d'optimiser les performances de leur connexion *TCP/IP*, le client et le serveur calculent en permanence les valeurs de *RTT* et *RTO*. Sur certains systèmes d'exploitation, ces valeurs sont prises en compte pour déterminer les délais entre paquets retransmis en cas de congestion. C'est par exemple le cas pour des systèmes Linux ou SUN Solaris mais pas pour certains IBM AIX.

À lui seul, ce comportement donne donc un indice sur le système d'exploitation, mais il est également source d'erreurs de détection. En effet, lorsqu'une comparaison simple entre des temps mesurés et des temps de référence ne permet pas de conclure, deux attitudes sont possibles : conclure que le système détecté est inconnu de la base, ou bien tenter une nouvelle comparaison en multipliant

Fig. 6. Terminaison à l'initiative du serveur, avec congestion simulée (par le client).

les valeurs de référence par le RTT approximé. Si le système détecté existe dans la base de référence et s'il prend effectivement en compte les valeurs de RTT et de RTO dans ses calculs de délai, alors cette nouvelle comparaison sera conclusive ; dans le cas contraire il y a un nouvel échec. Cette seconde comparaison conduit à plus de faux-positifs lorsqu'elle est activée, et plus de faux-négatifs lorsqu'elle est désactivée.

En résumé : `RING` *mesure des temps révélateurs de l'algorithme de gestion de congestion pour l'état* `SYN RCVD`. `Cron-OS` *mesure de plus les temps liés aux états* `LAST ACK` *et* `FIN WAIT 1` *et obtient en général des mesures plus précises. Dans quelques cas toutefois, l'interprétation des valeurs mesurées est beaucoup plus complexe.*

2.3 Effet des composants intermédiaires

L'*OSFP* n'est pas une technique exacte, notamment à cause de toutes les inconnues concernant l'effet des composants intermédiaires.

Fluctuations du réseau

Le premier composant intermédiaire traversé lors d'une détection est le réseau lui-même. S'il s'agit d'un réseau local, on peut s'attendre à quelques fluctuations liées à des congestions (véritables :-), des ralentissements de machines, etc. Sur Internet des changements de routes, des congestions supplémentaires liées aux routeurs, etc. peuvent également se produire.

En pratique, avec les paquets sans *payload* utilisés par `Cron-OS`, les variations constatées sur Internet en contexte opérationnel restent de l'ordre du dixième

de seconde. À titre d'exemple, voici une mesure ponctuelle d'écart sur des liens France-France ou France-Japon :

```
France-France 20 packets transmitted, 20 received, 0\% loss, time 19017ms
rtt min/avg/max/mdev = 67.828/70.299/73.020/1.372 ms
France-Japon 50 packets transmitted, 50 received, 0\% loss, time 49002ms
rtt min/avg/max/mdev = 333.417/336.614/343.022/2.121 ms
```

Les variations constatées sur cet exemple (et sur bien d'autres) n'excèdent donc pas 1% des temps mesurés par `RING` et `Cron-OS`, de l'ordre de la dizaine de secondes.

Effet des relais `SYN`

Ces appareils, généralement intégrés à des routeurs ou à des pare-feu protègent un réseau en endossant pour le compte des serveurs tout ou partie de la phase de connexion *TCP/IP*. En présence d'un relais `SYN`, la plupart des outils d'*OSFP* vont donc mesurer les caractéristiques du relais, et pas celle du serveur (cf. figure 7).

Il est rare qu'un relais `SYN` traite également les transmissions à l'initiative du serveur, et encore moins le traffic lié à la terminaison par le serveur de sa demi-connexion *TCP/IP*. Or c'est exactement le cas que provoque `Cron-OS` en plaçant le serveur soit dans l'état `LAST ACK` soit dans l'état `FIN WAIT 1`. Dans ce cas, c'est bien les caractéristiques du serveur qui sont mesurées, malgré la présence du relais.

Fig. 7. Schéma de principe d'un relais `SYN`. Les paquets analysés par `Cron-OS` sont émis à l'initiative du serveur, ils ne sont pas obligatoirement relayés.

Effet des systèmes de détection d'intrusion

Les IDS fonctionnant par signature repèrent difficilement les paquets échangés au cours d'une détection `Cron-OS` puisque ceux-ci sont conformes aux RFC. Une règle simple a pourtant été proposée sur la liste de discussion de `SNORT` :

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any
(msg:"Possible RING Scan";flags:SA; classtype:attempted-recon;
tag: host, 10, packets, dest;)
```

Cette règle produit énormément de fausses alertes, en particulier lorsque le réseau congestionne réellement ou lorsque les machines internes ralentissent. T. Beardsley [2] a remarqué une caractéristique des paquets forgés dans la première version de `RING` : l'`ISN` et le `timestamp` ont toujours les mêmes valeurs. Précisons que cette caractéristique est entièrement liée à l'implémentation actuelle de `RING`, la supprimer ne pénaliserait pas le processus de détection. Les règles `SNORT` suivantes reconnaissent spécifiquement ces paquets :

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any
(flags:S;seq:221002;msg:"SCAN RING";classtype:attempted-recon;)
alert tcp $HOME_NET any -> $EXTERNAL_NET any
(flags:SA;ack:221003;msg:"SCAN RING response";
                          classtype:successful-recon-limited;)
```

Un mot sur les *scrubbers*

Ce terme apparu assez récemment dans la littérature désigne un élément réseau filtrant permettant de "nettoyer" à la volée une partie du trafic. M. Smart et al. [8] proposent ainsi d'effectuer des actions de nettoyage et d'anonymisation au niveau *TCP/IP*. Ils abordent la notion d'*OSFP* basé sur des analyses temporelles mais indiquent qu'une protection contre ce type de prise d'information, bien que possible à mettre en œuvre, s'avère assez complexe.

En résumé : Les composants intermédiaires ont finalement peu d'impact sur le trafic généré par `RING` *ou* `Cron-OS`*. Les IDS par signature ne sont pas suffisament discriminants pour déclencher un blocage actif. Les relais* `SYN` *fournissent une bonne protection face à* `RING` *(et beaucoup d'autres outils), mais restent peu efficaces face à* `Cron-OS`*.*

3 Utilisation de `RING` et de `Cron-OS`

Cette section est essentiellement destinée au lecteur souhaitant utiliser `RING` et `Cron-OS`, voire modifier les codes source pour des usages particuliers.

3.1 Implémentation

L'implémentation actuelle de `Cron-OS` fait appel à trois bibliothèques :

- `Libpcap` pour l'écoute du réseau et la récupération des paquets.

– `Libnet` pour la construction précise de paquets et leur transmission.
– `Libdnet` pour ses fonctions de filtrage et de contrôle des pare-feu de nombreux systèmes d'exploitation.

La fonction de capture des paquets fournie par `libpcap` est exécutée avant la fonction de filtrage fournie par `libdnet`, ce qui permet de recevoir des paquets normalement filtrés par le pare-feu. La `libdnet` n'est utilisée que pour les tests liés à l'état `SYN RCVD`. Pour les nouveaux tests de `Cron-OS` les fonctions de `libdnet` ne permettaient pas un contrôle suffisament précis. Une partie du code est donc dédiée, et moins facilement protable (actuellement, cette partie est programmée pour Linux 2.4 uniquement).

Le pseudo-code ci-après donne l'algorithme général de `Cron-OS` :

```
Cron-OS() {
  libdnet.filtrer_paquet (machine_cible, port_ouvert) ;
  libnet.envoi_paquet (SYN, machine_cible, port_ouvert) ;
  while(1) {
    temps = libpcap.reception_paquet (machine_cible, ports, flags) ;
    mesures[n++] = temps ;
  }
  Cron-OS.confronter (mesures[ ], signatures[ ]) ;
}
```

3.2 Intégration avec `NMAP`

Chaque technique d'*OSFP* fait des erreurs de détection, mais toutes les techniques ne font pas forcément les mêmes erreurs. D'où l'idée d'utiliser plusieurs techniques pour une même détection, en gérant correctement la complémentarité. Cette idée est poussée à l'extrème avec les travaux d'O. Arkin et F. Yarochkin sur `XPROBE2` [1] qui permet d'intégrer plusieurs techniques et de pondérer leurs résultats.

Pour des raisons de simplicité et de connaissance des outils, nous avons choisi d'intégrer `Cron-OS` à `NMAP`, deux outils qui offrent une grande complémentarité (en particulier, `NMAP` différencie mal FreeBSD 4.4 et Windows 2000 alors que `Cron-OS` les confond moins souvent).

Au-delà de la complémentarité, l'intégration avec `NMAP` permet de bénéficier d'un format de signature, de conventions de nommage des système d'exploitation et d'une ligne de commande déjà bien connus des utilisateurs. L'outil résultat s'appelle `nmap-Cron-OS` et sa ligne de commande à l'allure suivante :

```
# ./nmap-Cron-OS  -p 80,1234  --ring --ring_method slf
                 --ring_timeout 60  -O shal.no-ip.org
      -p 80,1234         Ports 80 et 1234 testés
      --ring             Méthodes ring et / ou Cron-OS utilisées
      --ring_method slf  Etats s(yn), l(ast-ack) et f(in-wait1) analysés
      --ring_timeout 60  Ecoute de 60 secondes pour chaque état analysé
      -O                 Prise en compte des tests habituels de NMAP
      shal.no-ip.org     Nom de la cible
```

Les signatures sont une simple extension des signatures NMAP habituelles, en ajoutant 1, 2 ou 3 lignes supplémentaires correspondant respectivement aux tests s(yn), (l)ast-ack et (f)in-wait1. Voici un exemple de signature unifiée nmap-Cron-OS :

```
Fingerprint: Windows NT
TSeq(Class=RI%gcd=1%SI=3CC4%IPID=I...
T1(Resp=Y%DF=Y%W=FFFF%ACK=S++%Flags=AS%Ops=MNWNNT)
T2(Resp=N)
T3(Resp=Y%DF=Y%W=FFFF%ACK=S++%Flags=AS%Ops=MNWNNT)
T4(Resp=Y%DF=N%W=0%ACK=O%Flags=R%Ops=)
T5(Resp=Y%DF=N%W=0%ACK=S++%Flags=AR%Ops=)
T6(Resp=Y%DF=N%W=0%ACK=O%Flags=R%Ops=)
T7(Resp=Y%DF=N%W=0%ACK=S++%Flags=AR%Ops=)
PU(Resp=Y%DF=N%TOS=0%IPLEN=38%RIPTL=148%RID=E%RIPCK=E%UCK=E%ULEN=134%DAT=E)
Ring_Syn(nbPkt=2%Time=30%p=2806316%p=6010910)
Ring_LastAck(nbPkt=3%Time=30%Connect=134571%p=2918695%p=5909092%p=12014080)
Ring_FinWait(nbPkt=0%Time=30%Connect=136213)
```

3.3 Exemples simples

Sur ce premier exemple, un système Windows 2000 est protégé par un pare-feu très fermé. Les tests T2 à T7 et PU de NMAP n'apportent donc pas d'information. Ici, c'est Cron-OS permet de conclure correctement.

```
# nmap -n -P0 -v -d -O --osscan_guess -e eth0
                                  -p 22,80,6523 win-2K.qqpart.com
Aggressive OS guesses: AIX 4.3.2.0-4.3.3.0 on an IBM RS/* (88%),
AIX v4.2 (87%), IBM AIX v3.2.5-4 (87%), AIX 4.2-4.3.3 (87%)
SInfo(V=3.00%P=i686-pc-linux-gnu%D=4/2%Time=3E8B431D%O=80%C=-1)
TSeq(Class=TR%IPID=RD%TS=0)
T1(Resp=Y%DF=N%W=4000%ACK=S++%Flags=AS%Ops=MNWNNT)
T2(Resp=N) T3(Resp=N) T4(Resp=N) T5(Resp=N) T6(Resp=N) T7(Resp=N)
PU(Resp=N)

# nmap-Cron-OS -n -P0 -v -d --ring --ring_method sf -e eth0
                                 -p 22,80,6523i win-2K.qqpart.com
Remote operating system guess: Windows 2K Cron-OS
Ring_Syn(nbPkt=2%Time=30%p=3272209%p=6562494)
Ring_FinWait(nbPkt=4%Time=30%Connect=230166%p=849410%
                                   p=1749083%p=3280573%p=6563935)
```

Le deuxième exemple met en évidence la synergie entre NMAP et Cron-OS lorsque les deux outils arrivent à une même conclusion. L'indice de croyance est renforcé : $44/46 < 46/48$.

```
# nmap-Cron-OS -n -P0 --osscan_guess -O   -e eth0
                                         -p 80 lin24.qqpart.com
Remote OS guesses: Linux Kernel 2.4 Cron-OS(44success/46tests),
Linux Kernel 2.4.0 - 2.5.20(44success/46tests)
```

```
# nmap-Cron-OS -n -P0 --osscan_guess -O --ring
                --ring_method sf -e eth0 -p 80 lin24.qqpart.com
Remote OS guesses: Linux Kernel 2.4 Cron-OS(48success/48tests),
Linux Kernel 2.4.0 - 2.5.20(46success/48tests)
```

3.4 Exemples avancés

Voici un exemple où NMAP et Cron-OS ont tous deux l'air de fonctionner. Pourtant, chacun donne des résultats très différents. NMAP utilisé seul répond "Nokia M1122 DSL Router". Pour sa part, Cron-OS utilisé seul, c'est-à-dire avec seulement les tests liés aux états LAST ACK et FIN WAIT 1, répond "Win2k".

```
#nmap -n  -P0  -p 80 -O  x.x.x.x
Port       State       Service
80/tcp     open        http
Remote operating system guess: Nokia M1122 DSL Router
OS Fingerprint:
TSeq(Class=RI%gcd=1%SI=5937%IPID=I%TS=0)
T1(Resp=Y%DF=Y%W=FAF0%ACK=S++%Flags=AS%Ops=MNWNNT)
T2(Resp=N)
T3(Resp=Y%DF=Y%W=FAF0%ACK=S++%Flags=AS%Ops=MNWNNT)
T4(Resp=N) T5(Resp=N) T6(Resp=N) T7(Resp=N) PU(Resp=N)

#nmap-Cron-OS  -n  -P0  -p 80 --ring --ring_method fl x.x.x.x
Port       State       Service
80/tcp     open        http
Remote OS guesses: Win2k Pro Base/SP1/SP2/SP3, Win2k Srv
Base/SP1/SP2/SP3, Win2k AdvSrv Base/SP1/SP2/SP3 (2success/2tests),
Windows 95 B Cron-OS(2success/2tests)...
OS Fingerprint:
Ring_FinWait(nbPkt=3%Time=30%Connect=63863%p=1454762%p=4796315%p=9609807)
Ring_LastAck(nbPkt=3%Time=30%Connect=26047%p=2971331%p=6002374%p=12020256)
```

Finalement, c'est après avoir lancé les deux outils qu'on obtient une analyse acceptable de la situation : la cible est probablement un Windows 2000 caché derrière un relais SYN Nokia. En fait, NMAP a fait une empreinte du relais SYN et Cron-OS a fait une empreinte de la cible réelle.

Conclusion

Avec des outils comme NMAP, XPROBE2, TBIT, Cron-OS, etc. le domaine de l'*OSFP* a fortement évolué depuis les premières publications sur le sujet ; gageons qu'il reservera encore des surprises dans un avenir proche.

Concernant une évolution de Cron-OS, de prochains développements pourraient offrir plus de fonctionnalités et une meilleure précision, par exemple en mesurant les réactions des systèmes à la réception de paquets non prévus dans le diagramme d'état *TCP/IP*. NMAP procède de cette manière depuis toujours, mais uniquement lorsque le serveur se trouve dans l'un des états LISTEN ou CLOSED. D'autres états comme SYN RCVD ou FIN WAIT 1 sont pourtant très prometteurs...

A propos du projet
`Cron-OS` est un projet *Open Source* hébergé sur le site `Cron-OS.tuxfamily.org`. Les commentaires ou questions sont encouragés à l'adresse `Cron-OS@tuxfamily.org`. Les auteurs désaprouvent l'usage de `RING` et de `Cron-OS` à des fins malhonnêtes ou illégales.

Remerciements
Les auteurs tiennent à remercier pour leur aide messieurs Nicolas Prigent et David Fort, ainsi que les rapporteurs de cet article, messieurs Laurent Oudot et Sylvain Gombault.

Références

1. O. Arkin, F. Yarochkin, 2002. *XProbe2 - A 'Fuzzy' Approach to Remote Active Operating System Fingerprinting.* `www.xprobe2.org/archive/papers/Xprobe2.pdf`.
2. T. Beardsley, Plan B Security, 2002. *RING Out The Old, RING In The New : OS Fingerprinting through RTOs.* `www.planb-security.net/wp/ring.html`.
3. D. Comer, J. Lin, USENIX Summer Conf. 1994. *Probing TCP Implementations.* `www.bell-labs.com/user/johnlin/probing-TCP.pdf`.
4. Fyodor, Phrack 1998. *Remote OS detection via TCP/IP Stack FingerPrinting.* `www.insecure.org/nmap/nmap-fingerprinting-article.txt`.
5. P. Karn, C. Partridge, SIGCOMM 87. *Improving Round-Trip Time Estimates in Reliable Transport Protocols.*
6. B. Morin, L. Mé, H. Debar, M. Ducassé. *M2D2 : A Formal Data Model for IDS Alert Correlation. RAID 2002 : 115-127*
7. J. Padhye, S. Floyd, SigComm 2001. *Identifying the TCP Behavior of Web Servers.* `www.icir.org/tbit/nanog-tbit.pdf`.
8. M. Smart, G. R. Malan, F. Jahanian, 9th USENIX Security Symp. *Defeating TCP/IP Stack Fingerprinting.*
9. F. Veysset, O. Courtay, O. Heen *RING : New Tool and Technique For Remote OSFP.* `www.intranode.com/site/techno/techno-articles.htm`
10. F. Veysset, O. Courtay, O. Heen. *Cron-OS Information Page.* `Cron-OS.tuxfamily.org`
11. M. Zalewski, 2001. *Strange Attractors and TCP/IP Sequence Number Analysis.* `lcamtuf.coredump.cx/newtcp`.
12. Quelques outils :
    Ettercap (`ettercap.sourceforge.net`), p0f (`www.stearns.org/p0f/README`), IP Personality (`ippersonality.sourceforge.net`), RUT (`www.thc.org/thc-rut`)