Extending Orchids for Intrusion Detection in 802.11 Wireless Networks

Romdhane Ben Younes
Dépt. d'informatique, UQAM
ben_younes.romdhane
@courrier.uqam.ca

Guy Tremblay
Dépt. d'informatique, UQAM
C.P. 8888, Succ. Centre-Ville
Montréal, QC, Canada
H3C 3P8
tremblay.guy@uqam.ca

Guy Bégin
Dépt. d'informatique, UQAM
begin.guy@uqam.ca

ABSTRACT

In recent years, wireless equipment and services have become close to ubiquitous. Unfortunately, these services are more often than not ill configured security-wise, with minimal protection against potentially damaging attacks. To alleviate this problem, intrusion detection techniques may be used to help identify suspicious behaviour and counter intrusion attempts.

In this paper, we describe an extension to the Orchids intrusion detection tool, aimed at detecting intrusions in wireless networks. First, an event analysis module specialized for 802.1 wireless network events has been developed and integrated into Orchids. Next, a number of known attacks (e.g., deauthentication flooding, rogue access points and ChopChop) were modelized and described using declarative signatures. Then, within a simplified but realistic environment, the attacks were reenacted and successfully detected. To our knowledge, our team is the first to detect the ChopChop attack.

Categories and Subject Descriptors

D.2.4 **[Software Engineering]**: Software/Program Verification; D.4.6 **[Operating Systems]**: Security and Protection; C.2 **[Computer-Communication Networks]**

General Terms

Security, Verification

Keywords

Intrusion detection, wireless networks, model-checking

1. INTRODUCTION

Malgré de nombreuses lacunes sur le plan de la sécurité, les équipements sans-fil deviennent omniprésents : au travail, au café, à la maison, etc. Malheureusement, pour des raisons de convivialité, de simplicité ou par simple ignorance, ces équipements sont souvent configurés sans aucun service de sécurité, sinon un service minimal extrêmement vulnérable. Avec de telles configurations de base, plusieurs attaques sont facilement réalisables avec des moyens financiers négligeables et des connaissances techniques élémentaires.

Diverses approches peuvent être utilisées soit pour prévenir les attaques, soit pour réduire leur impact lorsqu'elles se produisent. Par exemple, les outils de *détection d'intrusions* peuvent aider les administrateurs systèmes à détecter certains comportements suspects et à prévenir ainsi les tentatives d'intrusions.

Dans cet article, nous proposons un détecteur d'intrusions dans les réseaux locaux sans-fil 802.11 qui étend l'outil Orchids [13, 11], un outil de détection d'intrusions fondé sur la vérification de modèles. Orchids est particulier en ce que le « modèle » à analyser est décrit directement par une trace d'exécution, obtenue par exemple d'un fichier de journalisation (analyse en différé) ou, dans notre cas, du flux réseau (analyse en *ligne*).

Dans un premier temps, nous présentons les principales vulnérabilités des réseaux locaux sans-fil 802.11, et donnons un aperçu d'attaques qui permettent d'exploiter ces faiblesses. Ensuite, nous présentons des méthodes utilisées pour la détection des attaques et présentons des solutions de détection d'intrusions spécifiques aux réseaux sans-fil 802.11. La section qui suit est consacrée à l'outil Orchids. Nous y présentons les principes de fonctionnement de cet outil, son architecture, ainsi que son langage de spécification des attaques. Par la suite, nous présentons la mise en œuvre du module sans-fil 802.11 pour Orchids, qui analyse et décortique (en ligne) les trames 802.11, ainsi que la spécification de signatures pour les attaques considérées — notamment, *ChopChop*. Finalement, nous présentons quelques expériences effectuées sur notre plateforme de test ainsi que les résultats obtenus.

2. VULNÉRABILITÉS ET DÉTECTION D'INTRUSIONS DANS LES RÉSEAUX SANS-FIL 802.11

« Your 802.11 Wireless Network has no Clothes », titre d'un article de Arbaugh, Shankar et Wan [1], exprime bien l'extrême vulnérabilité des réseaux 802.11. Cette vulnérabilité touche autant le niveau radioélectrique (couche phy-

NOTERE08 2008 Lyon, France

sique), où il est toujours possible d'effectuer une attaque de déni de service à l'aide d'un brouilleur de fréquences et d'une antenne faits maison, que les niveaux supérieurs, ciblés par des attaques plus complexes. Dans ce qui suit, nous nous limiterons aux vulnérabilités de la couche MAC 802.11 (*Medium Access Control*), ainsi qu'aux attaques actives.

2.1 Diverses formes d'attaques

Attaques contre l'authentification

Dans un réseau sans-fil 802.11, une station est identifiée à l'aide de son adresse MAC. Pour autoriser une station à s'associer au réseau, il faut « authentifier la station », c'est-à-dire vérifier son identité. Lorsque le système est configuré pour une authentification ouverte *(Open System authentication)*, ce qui est généralement la configuration par défaut, il n'est pas possible de véritablement « authentifier une station ». Par exemple, l'attaque d'usurpation d'adresse MAC *(MAC spoofing)* est facilement réalisable, puisqu'un intrus peut simplement contrefaire des trames 802.11 avec les adresses MAC de son choix.

Plusieurs chercheurs ont tenté de détecter l'attaque *MAC spoofing* [4, 6, 17] mais aucun ne réussit à le faire de façon efficace — les taux de faux positifs et faux négatifs demeurent élevés. Or, il est important de pouvoir détecter cette attaque car elle est généralement la première brique sur laquelle se construisent de nombreuses autres attaques [2, 4, 9, 14].

Attaques contre la disponibilité (dénis de service)

Véritable talon d'achille des technologies sans-fil, les attaques de déni de service sont parmi les plus simples à mettre en œuvre. Ainsi, une des attaques les plus courantes procède par inondation de trames de type désauthentification (*deauthentication flooding*), réalisable du fait que le protocole n'authentifie pas les trames de gestion, ce qui facilite leur usurpation (*spoofing*).

Le standard 802.11 permet la désauthentification d'une station, soit de sa propre initiative lorsque l'utilisateur n'a plus besoin du réseau, soit par le point d'accès lorsqu'il lui faut, par exemple, effectuer une mise à niveau. Dans les deux cas, l'envoi d'une simple notification suffit. Un intrus peut donc se faire passer soit pour le point d'accès, soit pour la station, et envoyer une trame de désauthentification contrefaite avec les adresses MAC source et destination respectives. Suite à la réception de cette trame, aucune communication n'est possible avant une ré-authentification de la station.

Une autre attaque semblable est celle par désassociation (*disassociate flooding*), qui repose sur un même principe et une même cause, parce que les trames de désassociation ne sont pas authentifiées. La seule différence est qu'en se dissociant, la station ne perd pas son authentification et donc il lui suffit de se ré-associer pour rétablir la communication.

Dans la même catégorie, on retrouve l'attaque par inondation RTS *(Request To Send)* fondée sur l'exploitation du mécanisme de réservation. L'intrus peut bloquer le canal de communication en envoyant des trames de réservation (RTS) qui comportent un champ durée ajusté sur un grand intervalle. On se trouve alors dans une situation où le canal reste monopolisé par l'intrus qui ne cesse de faire des réservations successives.

Une autre attaque vise le système de gestion de l'énergie. Pour économiser l'énergie, une station peut entrer en mode repos, état dans lequel elle ne peut ni envoyer ni recevoir de données. Dans ce cas, le point d'accès stocke tout le trafic qui est destiné à la station en mode repos dans un tampon, jusqu'à ce qu'elle se réactive pour récupérer les trames en attente. Si un intrus se fait passer pour la station pendant que celle-ci est en mode repos et récupére les trames en attente, celles-ci ne seront alors plus disponibles pour la station.

Attaques contre la confidentialité

Le standard 802.11 définit trois mécanismes de chiffrement : WEP *(Wired Equivalent Privacy)*, TKIP *(Temporal Key Integrity Protocol)*, et CCMP *(Counter Mode with Cipher Block Chaining Message Authentication Code Protocol)*.

Le WEP est le plus simple à paramétrer, mais il est aussi le plus vulnérable. En effet, bien qu'« il n'existe pas à l'heure actuelle de technique connue permettant de casser les protocoles TKIP et CCMP » [4], et à cause de la nécessité d'un déploiement laborieux pour ces deux protocoles, la plupart des équipements sans-fil sont activés soit sans aucun mécanisme de chiffrement, soit avec le WEP.

La figure 1 illustre le fonctionnement du WEP du côté de l'émetteur (mode chiffrement). La première étape consiste à appliquer une fonction CRC-32 sur le texte en clair pour générer un ICV *(Integrity Check Value)*, utilisé plus tard par le récepteur pour vérifier l'intégrité du message reçu. D'un autre côté, on produit un flux de chiffrement en appliquant la fonction RC4 à une clé WEP composée d'un vecteur d'initialisation *(IV — Initialization Vector)* et d'une clé pré-partagée *(PSK — Pre-Shared Key)*. Ensuite, le message chiffré est obtenu en effectuant un OU-exclusif (XOR) entre le flux de chiffrement et l'ensemble composé par le texte en clair suivi de l'ICV. Finalement, on compose la trame 802.11 en préfixant le message chiffré avec le vecteur d'initialisation et l'en-tête MAC, et en le suffixant avec le FCS *(Frame Check Sequence)*. Soulignons que le vecteur d'initialisation est transmis en clair pour permettre au récepteur de générer le même flux qui a servi au chiffrement.

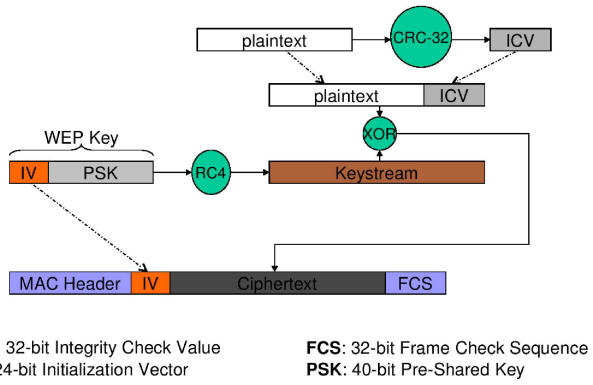

Figure 1: Fonctionnement du WEP en mode chiffrement (figure inspirée de [8]).

À la réception de la trame 802.11, le récepteur la découpe, extrait le vecteur d'initialisation et le concatène avec la clé pré-partagée pour former la clé WEP, applique la fonction RC4 sur cette clé pour produire le flux de déchiffrement, effectue un OU-exclusif entre le flux de déchiffrement et le

message chiffré pour obtenir le texte en clair et l'ICV associé, et applique finalement la fonction CRC-32 sur le texte en clair et compare le résultat avec l'ICV pour vérifier que le message n'a pas été modifié depuis sa transmission.

Les faiblesses du WEP son bien connues [16, 3] et plusieurs attaques[1] les ont exploitées : *dwepcrack*, *wepAttack*, *wep-tools*, *wep-crack*, *AirSnort*, *ChopChop*, *ARP Reply*, etc. Nous présentons l'attaque ChopChop, qui sera ensuite décrite (et détectée) à l'aide d'une signature Orchids (section 4.3.3).

ChopChop est le nom de l'outil publié par KoreK (un pseudonyme) en septembre 2004 sur un forum de discussion (`NetStumbler.org`). Cette attaque exploite une vulnérabilité liée à l'utilisation de la fonction CRC-32. La figure 2 illustre le déroulement de l'attaque ChopChop. On commence par supposer que l'octet de poids faible du texte en clair est un zéro (les huit bits à zéro). On supprime ensuite l'octet chiffré correspondant à cet octet du message chiffré, obtenant ainsi un nouveau message chiffré malformé car l'ICV est celui qui a été obtenu par le calcul de la fonction CRC-32 sur le message initial entier (message M) au lieu du message réduit d'un octet (message M-1). Pour corriger cette malformation, KoreK utilise une fonction mathématique permettant d'enlever l'effet du zéro sur l'ICV. Le message obtenu suite à l'application de cette fonction est valide seulement si la supposition initiale est bonne (soit, si l'octet de poids faible du texte en clair est effectivement un zéro).

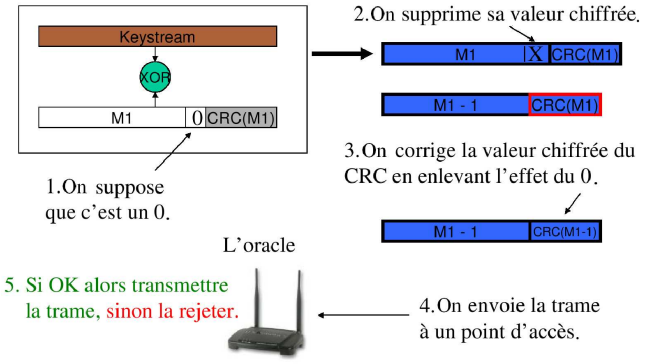

Figure 2: Déroulement de l'attaque ChopChop.

Pour vérifier si on a bien deviné, on envoie le nouveau message chiffré au point d'accès. Si le message est bon, il sera retransmis par le point d'accès, sinon il sera rejeté. Si le message est rejeté, on refait les mêmes opérations en incrémentant de 1 la valeur supposée pour le dernier octet, et ainsi jusqu'à ce qu'on trouve la bonne valeur. Quand on réussit à deviner le dernier octet, on l'enlève du message et on reprend la même démarche avec l'octet précédent jusqu'à ce qu'on découvre tout le message en clair, un octet à la fois.

2.2 Détection d'intrusions dans les réseaux sans-fil

Les détecteurs d'intrusions se fondent sur l'analyse des événements survenant durant le fonctionnement du système. Un événement est généralement matérialisé sous forme d'une ligne dans un fichier de journalisation ou sous forme d'une trame (ou fragment d'une trame) transmise sur le réseau ou sous une autre forme logicielle.

Les méthodes de détection d'intrusions peuvent être classées en deux grandes catégories : celles qui s'intéressent aux actions *interdites* dans le système surveillé — approches par détection d'abus *(misuse detection)* ou approches par scénarios — et celles qui s'intéressent aux actions *autorisées* — approches par détection d'anomalies *(anomaly detection)* ou approches comportementales.

Avec une approche par détection d'abus, on dispose d'une banque de signatures d'attaques (actions illégales) servant à identifier des attaques. On doit alors chercher les occurrences de ces signatures dans un flux d'événements ; si on trouve une concordance, on lance une alerte. L'inconvénient de cette approche est de ne pouvoir détecter que les attaques connues.

L'approche par détection d'anomalies consiste à définir le comportement normal (actions légales). Ainsi, toute combinaison d'événements qui violent ce comportement attendu est considérée comme suspecte. Le principal avantage de cette approche est de permettre la détection d'attaques inconnues. Par contre, elle est sujette à de nombreux faux positifs si le modèle comportemental n'est pas complet ou s'il est trop restrictif.

Dans le cas des réseaux locaux sans-fil 802.11, les particularités suivantes sont importantes pour déterminer quel type de détection d'intrusions est le plus approprié :

- Le débit est moins élevé que dans les réseaux filaires, donc il y a relativement moins de données à analyser.
- L'environnement favorise la mobilité des systèmes, ce qui implique le besoin de *corrélation* — par exemple, on doit corréler des événements observés sur différents points d'accès.

Quelques outils permettant d'effectuer la détection d'intrusions dans les réseaux sans-fil sont actuellement disponibles, notamment Kismet [7] et Snort-Wireless (décrits plus en détails à la section 6). Une particularité de ces divers outils est qu'il s'agit généralement *d'extensions* d'outils de détection d'intrusions existants.

Dans la prochaine section, nous présentons de façon plus détaillées l'outil Orchids, que nous avons étendu pour effectuer la détection d'intrusions dans un réseau sans-fil 802.11.

3. ORCHIDS

Orchids [11] est un détecteur d'intrusions par reconnaissance de scénarios qui peut analyser et corréler plusieurs événements séparés dans le temps et qui, lorsque requis, peut appliquer des contre-mesures. Le processus d'analyse est basé sur un algorithme d'analyse de journaux d'événements (*log files*) [13], auquel on a apporté quelques optimisations et ajouté de nouveaux opérateurs — *cut*, *without*, etc. L'outil Orchids est distribué sous la licence de logiciel libre CeCILL[2] (version 2).

Bien que l'outil Orchids soit fondé sur une approche de vérification de modèles — mieux connue sous le nom de « *model-checking* » [5, 15] — il diffère d'autres outils de ce genre par la façon dont le comportement du système est décrit. Ainsi, plutôt que d'analyser un modèle abstrait —

[1] `www.wi-foo.com`

[2] `http://www.cecill.info`.

décrit par exemple par un automate ou un ensemble de processus décrits dans une algèbre — Orchids analyse plutôt, en ligne ou en différé, une représentation du comportement effectif du système, à savoir une séquence d'événements générés au cours de l'exécution du système, par exemple, les fichiers de journalisation, les flux réseaux, etc. Les propriétés à vérifier sont alors les signatures d'attaques qu'on veut détecter. Ces propriétés sont exprimées par des automates, représentant un sous-ensemble de LTL [12].

3.1 Architecture générale d'Orchids

La plateforme Orchids est composée de quatre composantes principales (voir figure 3) :

- un ensemble de règles qui décrivent les comportements suspects et inhabituels qu'on veut détecter ;
- un compilateur de règles qui traduit les définitions des règles en une représentation interne d'automates ;
- une machine virtuelle qui simule le comportement des automates — machine qui forme le *noyau* d'Orchids ;
- un ensemble de modules qui décodent le flux d'informations provenant de diverses sources.

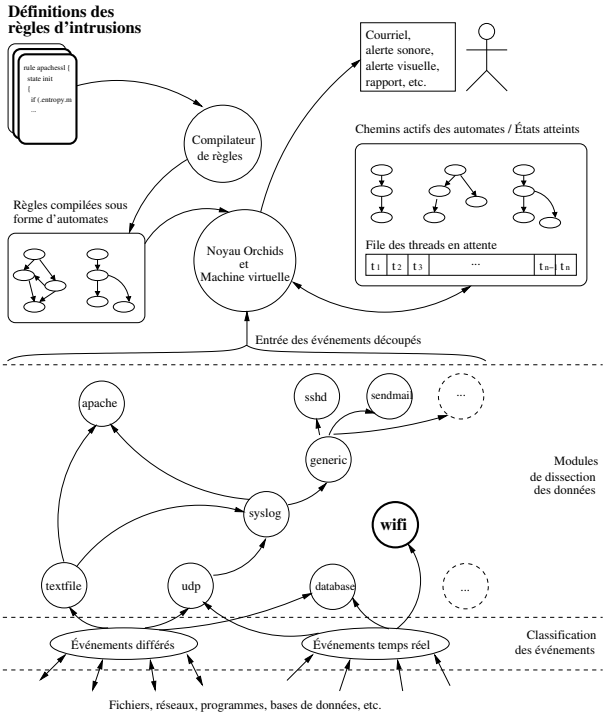

Figure 3: Architecture d'Orchids (inspirée de [10]).

La syntaxe du langage de spécification des règles est décrite dans la documentation de l'outil[3]. Nous présentons un exemple plus bas. Le compilateur traduit chaque règle en un automate pour constituer ainsi la base des signatures d'attaques à détecter. À partir de cette base et des événements injectés dans le noyau d'Orchids (la machine virtuelle), ce dernier va simuler les automates et identifier, à travers les divers chemins actifs et états atteints, les attaques qui surviennent.

[3] `http://www.lsv.ens-cachan.fr/orchids/`

Les événements injectés proviennent des modules de dissection de données. Ainsi, chaque type de journal (ou de flux) est associé à un module de dissection spécifique, module servant à découper le flux d'informations en divers champs. Ces champs peuvent être ensuite référencés dans les définitions des règles. Dans la section 4, nous présentons le module **wifi**, que nous avons développé et qui traite les événements survenant sur un réseau local sans-fil 802.11. Mais auparavant, nous présentons un exemple de scénario d'attaque exprimé avec la syntaxe d'Orchids.

3.2 Spécification des scénarios d'attaque

En Orchids, les règles de spécification des signatures sont vues à un niveau abstrait comme des automates finis. La figure 4 présente un automate pour la signature d'une attaque de capture de mot de passe sur un réseau *(password sniff)*. Chaque règle doit avoir au moins un état initial init, et un ou plusieurs états finaux. À partir de l'état init, et selon les événements reçus (p. ex., lus dans le journal), on peut passer à travers les états suivants un à un jusqu'à ce qu'on atteigne un état final.

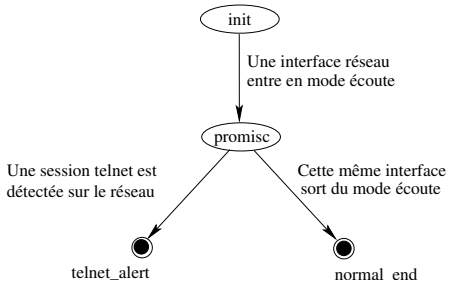

Figure 4: Automate pour l'attaque password_sniff.

Pour notre exemple, l'événement qui déclenche le passage à l'état promisc est la détection d'une interface réseau qui entre en mode écoute (*promiscuous mode*). À partir de cet état, si on détecte que l'interface réseau a quitté le mode écoute, on passe à l'état normal_end et aucune alerte n'est déclenchée. Par contre, si on détecte le début d'une session Telnet on passe à l'état telnet_alert pour aviser l'administrateur système de cette situation.

La signature Orchids de l'attaque en question est présentée à la figure 5. Le mot clé `rule` permet d'indiquer le nom de la règle. Chaque règle est composée de plusieurs blocs, chacun débutant par le mot clé `state` suivi du nom de l'état.

À un état peuvent être associées des actions et des transitions. Une action « $var = .module.champ » récupère la valeur du champ champ (générée par le module de dissection module) et la met dans la variable $var.

Une autre forme d'action fréquemment rencontrée, « fonction(arguments) », correspond à un appel de fonction — Orchids propose une multitude de fonctions prédéfinies.

Lorsqu'une suite d'événements conduit à un état terminal (sans transition sortante), on dit que la règle a été observée. C'est donc qu'une séquence d'événements décrite par la règle a été identifiée dans les événements injectés dans le noyau Orchids.

```
rule password_sniff {

  state init {
    if ( .promisc.action == "enter" )
      goto promisc;
  }

  state promisc {
    $interface = .promisc.interface;
    if ( .promisc.action == "left" &&
         .promisc.interface == $interface )
      goto normal_end;
    if ( .xinetd.action == "START" &&
         .xinetd.service == "telnet" )
      goto telnet_alert;
  }

  state telnet_alert {
    print("warning, telnet while someone is sniffing");
  }

  state normal_end {
  }

} // end rule
```

Figure 5: Signature de l'attaque password_sniff.

4. MODULE WIFI POUR ORCHIDS ET SIGNATURES D'ATTAQUES

Dans ce qui suit, nous présentons l'architecture de notre plateforme de détection d'intrusions, dont le module de dissection des trames 802.11, ainsi qu'un certain nombre de signatures d'attaques que nous avons spécifiées et testées.

4.1 Architecture de la plateforme de détection d'intrusions

Les détecteurs d'intrusions sont généralement composés d'un ensemble de sondes (pour la collecte des événements) et d'un ou plusieurs analyseurs. Pour une meilleure sécurité, on place généralement les sondes sur les réseaux qu'on veut surveiller et les analyseurs dans une zone protégée.

En attachant un ensemble de sondes à un analyseur, on doit veiller à ce que le nombre d'événements reçus par ce dernier ne dépasse pas sa capacité de traitement, sinon des événements peuvent être rejetés et on pourrait donc passer à côté d'une attaque sans la remarquer.

Un autre choix crucial est celui de l'endroit où placer les sondes, soit n'importe où dans la zone couverte par le point d'accès, soit dans le point d'accès lui-même. Bien que situer la sonde dans le point d'accès semble le plus approprié, puisque c'est le point central de la zone qu'il couvre, nous avons choisi de la placer dans la proximité immédiate du point d'accès, pour les raisons suivantes. Tout d'abord, le détecteur d'intrusions n'a pas pour tâche de protéger seulement le point d'accès mais aussi les clients du réseau. Il faudrait donc une interface, avec une meilleure sensibilité, qui puisse couvrir une zone plus large que celle couverte par le point d'accès. Ainsi, un intrus potentiel serait visible pour le détecteur d'intrusions même s'il agit en dehors de la zone couverte par le point d'accès. Ensuite, la charge d'analyse est assez considérable. Il faudrait donc concevoir des points d'accès plus performants avec des processeurs plus rapides et une plus grande mémoire. Ceci implique non seulement des coûts supplémentaires, mais aussi une gestion plus lourde pour les équipements.

4.2 Capture et découpage des trames 802.11

En se basant sur l'architecture retenue, on pouvait soit exploiter les journaux du pilote de la carte sans-fil, soit travailler directement sur les trames 802.11 en format brut. Notre choix s'est fixé sur la deuxième possibilité. En effet, les formats des journaux des pilotes ne sont pas standardisés et dépendent donc de la carte et du pilote utilisés. De plus, pour que la détection d'une intrusion se fasse plus rapidement (en ligne plutôt qu'en différé), il est préférable d'analyser directement les trames au format brut.

Champ	Type	Description
wifi.type	T_STR	*frame type*
wifi.subtype	T_STR	*frame subtype*
wifi.time	T_TIMEVAL	*reception time*
wifi.da	T_STR	*destination address*
wifi.sa	T_STR	*source address*
wifi.bssid	T_STR	*basic service set identifier*
wifi.ta	T_STR	*transmitter address*
wifi.ra	T_STR	*receiver address*
wifi.dir	T_STR	*direction of the frame*
wifi.seqnum	T_INT	*sequence number*
wifi.iswep	T_INT	*1 if it is an encrypted frame, 0 otherwise*
wifi.bodylen	T_INT	*body length*
wifi.rssi	T_INT	*received signal strength identification*

Tableau 1: Champs retenus pour chaque trame identifiée par le module wifi.

Le tableau 1 indique les divers champs retenus pour une trame wifi : nom du champ, type et bref commentaire décrivant sa signification.

La capture des trames est effectuée à l'aide de la bibliothèque `Pcap`. Lorsqu'une trame est captée, un enregistrement avec les champs appropriés est créé puis injecté dans le noyau d'Orchids, i.e., transmis à la machine virtuelle.

4.3 Spécification de signatures d'attaques

Pour spécifier de façon correcte et précise une signature d'attaque, il faut non seulement *bien connaître l'attaque* — afin de ne traiter que l'essentiel des informations (minimum requis pour identifier l'attaque rapidement) tout en s'assurant de détecter toutes les variantes —, mais aussi *bien connaître le système* surveillé — afin d'éviter les *fausses alertes*.

Bien que le nombre et la variété d'attaques détectées par un outil de détection d'intrusions soit un facteur important de la puissance de cet outil, nous avons choisi ici de ne décrire que quelques signatures d'attaques fondamentalement différentes. L'objectif est de montrer les principales fonctionnalités offertes par Orchids et par notre extension, ainsi que certaines de leurs limites. Plus précisément, dans ce qui suit, nous présentons les attaques *deauthentication flooding*, *rogue access point* et *ChopChop*.

4.3.1 *Identification de l'attaque* deauthentication flooding

Le fonctionnement de cette attaque a été illustré à la section 2.1. Ici, nous présentons les propriétés essentielles de l'attaque *deauthentication flooding* ainsi que notre démarche pour l'identifier à l'aide d'Orchids.

Une attaque de désauthentification par inondation génère un nombre anormalement élevé de trames de désauthentification impliquant un ou plusieurs clients. En effet, lorsqu'un client est désauthentifié, il va tenter de se réauthentifier. Pour l'empêcher de reprendre sa connexion, l'intrus peut envoyer à répétition (à une fréquence donnée) des trames de désauthentification. Ainsi, il est possible d'excéder le temps d'arrêt *(timeout)* d'un protocole de couche supérieure (p. ex., TCP), ce qui causerait la perte de toutes les sessions du client. Il est aussi possible d'effectuer cette attaque avec une fréquence moins élevée, ce qui n'empêche pas le client de travailler mais qui réduit le débit de communication.

Le choix de la fréquence à partir de laquelle on peut juger que le comportement est suspect dépend de l'architecture, des applications, du matériel et des clients. Nous supposons donc que ce choix est laissé à l'administrateur du réseau.

Pour paramétrer cette propriété dans la signature d'attaque, nous avons défini deux constantes, `DELAY` et `NB_FRAMES` : `DELAY` indique l'intervalle maximal durant lequel on va chercher une instance de l'attaque, alors que `NB_FRAMES` indique le nombre minimal de trames de désauthentification qu'il faut observer durant l'intervalle `DELAY` avant de signaler une alerte. Donc, si on observe au moins `NB_FRAMES` trames de désauthentification dans un intervalle de temps de `DELAY` secondes, on signale une alerte. La figure 6 présente un premier automate correspondant à la signature de l'attaque *deauthentication flooding*, automate que sera amélioré par la suite.

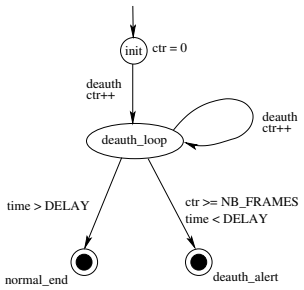

Figure 6: Automate correspondant à la signature de l'attaque *deauthentication flooding*.

Pour mieux visualiser le problème avec cette description, on peut simplifier l'automate de la figure 6 en ignorant le paramètre `DELAY` et en fixant `NB_FRAMES` à 2, ce qui nous ramène à l'automate de la figure 7 (avec d=deauth, L=deauth_loop et A=deauth_alert). Cet automate modélise donc le fait de signaler une alerte après l'observation de deux trames de type désauthentification.

La figure 8 illustre le résultat de la simulation de l'automate simplifié de la figure 7 sur la séquence d'événements d_1, d_2, d_3, et d_4 correspondant à l'observation de quatre trames de type désauthentification provenant de la même source et envoyées vers la même destination. On peut remarquer que trois instances de règles aboutissent à un état d'alerte. La première alerte signale le chemin $[d_1 d_2]$, la deuxième signale le chemin $[d_2 d_3]$ et la troisième signale le chemin $[d_3 d_4]$. Et si une cinquième trame de type désauthentification (d_5) surgit, une nouvelle alerte serait lancée en signalant le chemin $[d_4 d_5]$, et ainsi de suite.

Ce comportement n'est pas le comportement idéal souhaité, puisque pour une même attaque, il est préférable

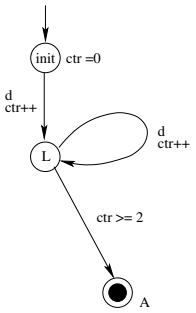

Figure 7: Automate simplifié de la signature pour *deauthentication flooding*.

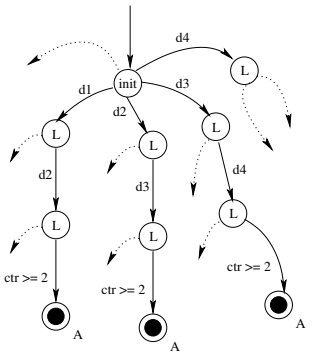

Figure 8: Simulation de l'automate de la figure 7.

de réduire autant que possible le nombre d'alertes. Ainsi, au lieu de trois alertes signalant successivement $[d_1 d_2]$, $[d_2 d_3]$ et $[d_3 d_4]$, il serait préférable de n'avoir que deux alertes signalant successivement $[d_1 d_2]$ et $[d_3 d_4]$, donc ne signaler une alerte qu'à tous les `NB_FRAMES` trames.

La création d'une nouvelle instance de règle n'est nécessaire que lorsqu'il s'agit d'une nouvelle attaque (un nouvel attaquant ou une nouvelle victime) ou lorsqu'il n'existe aucune instance de règle active qui traite une attaque en cours. Ceci peut être réalisé, en Orchids, à l'aide du mot clé `synchronize` appliqué, dans notre cas, à l'adresse source `$addr_src` et l'adresse destination `$addr_dst` — `synchronize` est un mécanisme de verrouillage, qui assure qu'une seule instance de règle utilisera les variables indiquées, empêchant ainsi la création de nouvelles instances de règles.

Ensuite, il est aussi préférable d'observer un temps d'attente après chaque alerte, de façon à retarder la libération de l'instance de règle pour un même couple victime/attaquant. En d'autres termes, l'état d'alerte A ne doit plus être un état terminal. Pour cela, on rajoute une transition partant de l'état A vers un nouvel état terminal, transition qui ne doit être franchie qu'après l'écoulement d'un certain délai d'attente représenté par un paramètre `WAITIME` (en secondes, voir figure 9). Ainsi, si après ce délai l'attaque est encore active, alors on signale une nouvelle alerte puis on attend de nouveau durant `WAITIME` secondes.

Cela dit, le fait de maintenir l'instance de règle active durant un certain temps n'est pas sans conséquences. En effet, dans la machine virtuelle Orchids, un *thread* est associé à chaque chemin partiellement exploré. Or, les *threads* des états précédant l'état d'alerte restent actifs tant qu'on n'a

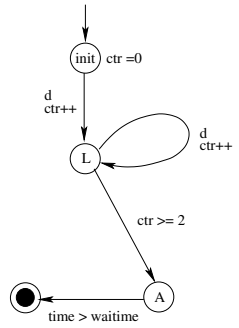

Figure 9: Automate modifié pour *deauthentication flooding* avec délai d'attente après une alerte.

pas atteint un état terminal. Ceci veut dire que, tant qu'on ne libère pas l'instance de règle, plusieurs *threads* vont continuer à chercher un chemin vers un état terminal et vont potentiellement atteindre l'état d'alerte.

Ce dernier problème peut être évité en utilisant la fonction `cut`, qui met en œuvre le concept de « coupure explicite » de Prolog, et qui permet donc d'appliquer un choix définitif *(commited choice)* en arrêtant la recherche de nouveaux chemins à partir d'un état donné. Dans notre cas, un appel à `cut("deauth_loop")` à l'entrée de l'état `deauth_alert` permet ainsi d'arrêter tous les *threads* en attente dans le sous-arbre ayant pour racine la première instance d'état `deauth_loop`. Le code complet de la signature d'attaque *deauthentication flooding* est présenté à la figure 10.

4.3.2 *Identification de l'attaque* rogue access point

On distingue deux sortes de points d'accès voyous (*rogue access points*) : ceux qui ne sont pas autorisés à fonctionner dans une zone géographique déterminée et ceux qui sont autorisés mais qui sont mal configurés (par rapport à une politique de sécurité définie). De tels points d'accès représentent un risque dans un environnement où les besoins de sécurité sont élevés, et leur présence doit être signalée par le détecteur d'intrusions. Ici, on ne s'intéresse qu'au premier type de points d'accès voyous (non autorisés).

Pour détecter les points d'accès non autorisés, la signature d'attaque doit spécifier la liste (des adresses MAC) des points d'accès autorisés. Il suffit alors d'observer une trame 802.11 impliquant un point d'accès qui n'est pas dans cette liste pour signaler une alerte. De plus, comme on l'a fait avec l'attaque *deauthentication flooding*, il faut aussi éviter de multiplier les alertes pour une attaque répétitive. Par exemple, on ne veut pas signaler une nouvelle alerte — du moins durant un temps déterminé — pour un point d'accès voyou pour lequel on vient tout juste de signaler une alerte.

La signature Orchids pour cette attaque est relativement simple (voir figure 11), et ce grâce au mot clé `synchronize`. En effet, à l'aide de cette construction, il n'est pas nécessaire de gérer explicitement une liste des points d'accès voyous, comme c'est le cas avec la plupart des autres outils : la gestion de cette liste, ainsi que tout le travail de vérification des adresses MAC autorisées, est effectuée au coeur du noyau d'Orchids, de façon implicite.

Par contre, signalons que le langage des signatures d'Orchids n'inclut pas un type tableau ou collection. Les diverses adresses MAC doivent donc être testées de façon explicite, à l'aide d'une conjonction complexe.

```
#define NB_FRAMES   5
#define DELAY       10
#define WAITIME     15

rule deauth_flood
  synchronize($addr_src, $addr_dst)
{

  state init {
    $addr_src = .wifi.sa ;
    $addr_dst = .wifi.da ;

    $counter = 0 ;
    $start_time = .wifi.time ;
    $end_time = .wifi.time + DELAY ;

    if ( .wifi.subtype == "deauth" )
      goto deauth_loop;
  }

  state deauth_loop {
    $counter = $counter + 1;

    if (    .wifi.subtype == "deauth"
         && $addr_src == .wifi.sa
         && $addr_dst == .wifi.da
         && .wifi.time < $end_time
         && $counter < NB_FRAMES )
      goto deauth_loop;

    if (    .wifi.subtype == "deauth"
         && $addr_src == .wifi.sa
         && $addr_dst == .wifi.da
         && .wifi.time < $end_time
         && $counter >= NB_FRAMES )
      goto deauth_alert;

    if (    .wifi.subtype == "deauth"
         && $addr_src == .wifi.sa
         && $addr_dst == .wifi.da
         && .wifi.time >= $end_time )
      goto normal_end;
  }

  state deauth_alert {
    cut("deauth_loop");
    $end_time = .wifi.time + WAITIME ;
    $msg =   "deauthentication flood src:"
           + .wifi.sa + " dst:" + .wifi.da ;
    print ( $msg );

    if ( .wifi.time >= $end_time )
      goto normal_end;
  }

  state normal_end {
  }

}
```

Figure 10: Code complet de la signature d'attaque *deauthentication flooding*.

4.3.3 *Identification de l'attaque* ChopChop

L'attaque *ChopChop* a été décrite à la section 2.1. On y a vu que pour découvrir une trame chiffrée de N octets, un intrus doit effectuer N cycles de devinettes. Durant chaque cycle, de 1 à 256 trames de même taille et transportant exactement les mêmes données, sauf pour l'ICV, sont envoyées vers un point d'accès. En passant d'un cycle à un autre, la taille des données est réduite d'un octet (voir figure 12).

On vient de décrire l'essentiel de l'attaque, mais d'autres indices peuvent venir préciser cette description, comme par exemple le fait que les trames soient envoyées vers des adresses de destination *multicast* (débutant par `ff`) succes-

```
#define TIMEOUT 3600

rule rogue_ap synchronize( $bssid ) {

  state init {
    $bssid = .wifi.bssid ;
    if (    .wifi.bssid != "00:1b:11:61:af:74"
         && .wifi.bssid != "00:15:e9:b3:f4:7e" )
      goto alert;
  }

  state alert {
    $msg = " Rogue AP detected (BSSID " + $bssid + ")" ;
    $end_time = .wifi.time + TIMEOUT ;
    print ( $msg );
    if ( .wifi.time >= $end_time )
      goto the_end;
  }

  state the_end {
  }

}
```

Figure 11: Signature d'attaque *rogue access point*.

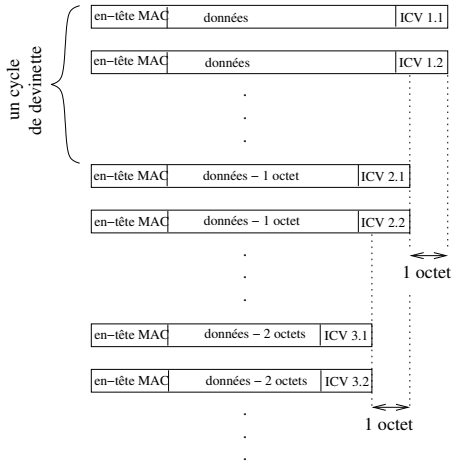

Figure 12: Cycles de devinette d'une attaque *ChopChop*.

sivement différentes au niveau d'un seul octet (exemple : `ff:c1:24:e3` et `ff:c1:24:e9`).

La figure 13 présente un automate correspondant à cette description de l'attaque *ChopChop*. On commence par initialiser un compteur `ctr` à 0 (dans l'état `init`). Ensuite, dès qu'on reçoit une trame chiffrée destinée à une adresse *multicast*, on passe à l'état `first_loop` en sauvegardant la taille de la trame dans la variable bodylen (sur la figure, la taille est N). Le passage à l'état `first_loop` marque le début d'un éventuel cycle de devinette. On s'attend donc à recevoir de 1 à 256 trames de même taille : bodylen' == bodylen.

La transition vers l'état `second_loop` marque la fin d'un cycle de devinette et le début du cycle suivant puisque la taille des données est réduite d'un octet : bodylen' == bodylen-1. Dans l'état `second_loop`, on réinitialise la valeur de bodylen avec celle de bodylen' et on incrémente le compteur `ctr` — variable qui compte le nombre de cycles observés. Si on atteint un certain nombre de cycles à partir duquel on juge que le comportement observé est probablement une attaque *ChopChop* (dans l'exemple on a choisi de le fixer à 3), on signale une alerte, sinon on revient à l'état `first_loop` et on cherche un nouveau cycle, et ainsi de suite.

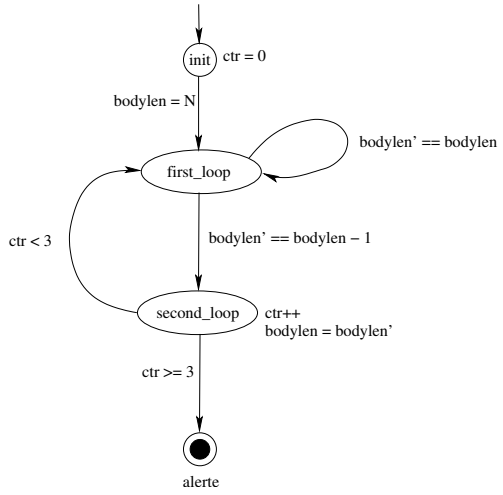

Figure 13: Automate correspondant à la signature d'attaque *ChopChop*.

Si on simule directement cet automate, l'espace d'états — donc l'espace mémoire requis — explose rapidement, et ce même pour un nombre relativement limité de cycles courts. Par contre, en utilisant la construction `synchronize` sur l'attribut `bodylen`, on peut réduire de façon considérable l'espace mémoire requis. En effet, il n'est pas nécessaire de créer une nouvelle instance de règle durant un même cycle de devinette.

Malheureusement, cette amélioration, bien qu'intéressante, n'est pas suffisante dans un scénario où l'attaquant ne réussit à deviner chaque octet qu'après un grand nombre de tentatives.

Pour atténuer davantage le problème de l'explosion du nombre d'états, on pourrait être tenté d'effectuer un choix définitif *(commited choice)* après chaque cycle en faisant appel à `cut("init")` à l'entrée de l'état second_loop. Bien qu'on réussisse ainsi à réduire le nombre d'états, cette opération est dangereuse, comme l'illustre la figure 14 : à partir du premier `cut("init")` aucune nouvelle instance de règle ne peut être créée, ce qui peut masquer des attaques.

Avec cet exemple, nous en arrivons donc aux limites d'Orchids. Une idée intéressante serait d'étendre le langage de signatures en définissant un nouveau type d'état « volatile ». Ainsi, dans le cas où une transition boucle sur un état, on pourrait allouer un espace mémoire pour un seul état qu'on écrase avec chaque nouvelle occurence, ce qui réduirait de façon significative l'espace d'états d'une signature d'attaque.

4.3.4 Autres attaques

En plus des attaques *deauthentication flooding*, *rogue access point* et *ChopChop*, nous avons pu aussi détecter les attaques *ARP Replay* et *MAC spoofing*, mais avec quelques cas de faux positifs : à notre connaissance, aucun outil ne réussit à détecter ces attaques sans cas de faux positifs (ou même des cas de faux négatifs), dû à la nature même de ces attaques.

Notre expérience nous porte aussi à croire que nous pour-

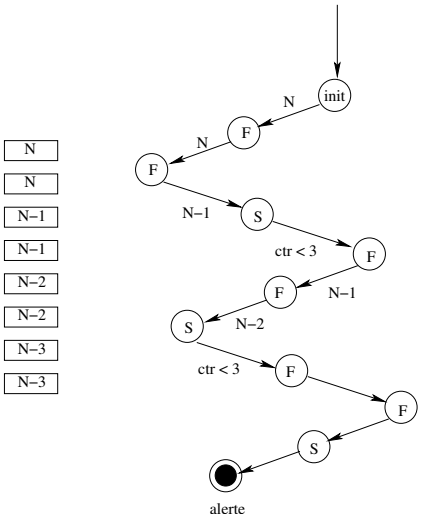

Figure 14: Simulation de l'automate de la figure 13 modifié à l'aide de la fonction cut.

rions facilement détecter plusieurs attaques semblables à l'attaque *deauthentication flooding*, telles que *authentication flooding*, *association flooding*, *disassociation flooding*, etc., ainsi que toutes les attaques qui révèlent une empreinte particulière, telles que *Wellenreiter*, *Airjack*, *NULL SSID Probe Response Dos*, *Long SSID*, etc.

5. EXPÉRIMENTATIONS

Dans cette section, nous présentons les résultats d'expériences effectuées sur notre plateforme de tests avec les signatures d'attaques décrites à la section précédente. Pour deux de ces attaques, nous comparons aussi nos résultats avec ceux obtenus avec l'outil Snort-Wireless (que nous appelerons simplement Snort). Nos critères de comparaisons seront la consommation mémoire, le temps processeur, le temps de réaction et l'exactitude des alertes.

La plateforme de tests consiste en un réseau local sans-fil 802.11b, en mode infrastructure, connecté à Internet à travers un réseau câblé. Les détails de la configuration matérielle et technique des équipements impliqués dans nos tests sont présentés au tableau 2.

Point d'accès	Routeur sans-fil Linksys (modèle BEFW11S4) mettant en œuvre le standard IEEE 802.11b et utilisant le chiffrement WEP.
Client	Ordinateur sous Windows 2000 équipé d'une carte D-Link modèle DWL-G122, standard 802.11g.
Détecteur d'intrusions	Ordinateur sous Linux (Fedora Core 6, kernel-2.6.20) équipé d'une carte D-Link modèle DWL-G520 (*chipset* Atheros AR-5212 802.11abg), standard 802.11g.
Intrus	Ordinateur sous Linux (Fedora Core 3, kernel-2.6.12) équipé d'une carte Netgear modèle WG511T (*chipset* Atheros AR-5212 802.11abg), standard 802.11g.

Tableau 2: Plateforme de tests.

Dans tous les tests, nous avons supposé que l'intrus n'a aucune connaissance préalable de la configuration du réseau et ne dispose d'aucune information privilégiée.

Pour réaliser les attaques *deauthentication flooding* et *ChopChop*, nous avons choisi d'utiliser l'outil `aireplay-ng` — qui fait partie de la collection d'outils Aircrack-ng[4] — que nous avons paramétré minimalement.

5.1 Tests de l'attaque *deauthentication flooding*

Dans ce premier scénario d'attaque, on a lancé sur le poste du client la commande `ping -t` vers un poste situé derrière le point d'accès (un poste connecté à Internet). Cette commande permet d'envoyer, sans interruption, des trames ICMP *request* vers le poste spécifié en paramètre. Si tout se passe bien, on doit recevoir une trame ICMP *echo* en réponse à chaque requête envoyée. Si un problème survient, le message « `Délai d'attente de la demande dépassé` » est affiché sur le poste du client.

Sur le poste de l'intrus, l'attaque est effectuée à l'aide de la commande `aireplay-ng -0` en lui passant en paramètres les adresses MAC du point d'accès et de la station client et en précisant le nombre de trames désauthentification à envoyer. L'outil envoie alors la moitié de ces trames dans le sens « point d'accès vers client » et l'autre moitié dans le sens « client vers point d'accès ». Dans ce qui suit, on utilisera la notation simplifiée suivante pour décrire ces attaques : source $\xrightarrow{nbt}$ destination (où *nbt* correspond au nombre de trames de type désauthentification).

Lorsque la station du client légitime reçoit la première trame de type désauthentification, sa connexion est perdue et le message « `Délai d'attente de la demande dépassé` » est affiché. La station tente alors de se réauthentifier auprès du point d'accès mais le flot de trames de type désauthentification reçu empêche le rétablissement de la connexion. C'est seulement après la réception de la dernière trame que la station réussit à se réauthentifier.

Pour détecter cette attaque, on a configuré Orchids et Snort pour charger uniquement la signature d'attaque *deauthentication flooding* propre à chaque outil. Dans la signature Orchids, on doit spécifier les valeurs suivantes : la fréquence des trames de désauthentification et la durée d'inertie *par rapport à une attaque* (pour éviter de signaler une même attaque plusieurs fois). Dans la signature Snort, on doit spécifier la fréquence des trames et la durée d'inertie *par rapport à une victime*. En fait, Snort ajoute les victimes (adresse destination) dans une liste ; s'il détecte une attaque contre une victime déjà présente dans cette liste, il ne signale aucune alerte. Une victime est retirée de la liste quand elle n'est plus la cible d'aucune attaque durant une durée de temps déterminée — qu'on a appelé « durée d'inertie par rapport à une victime ».

On suppose dans ce qui suit que les deux signatures sont initialisées avec les valeurs suivantes (choisies sans biais particulier envers notre outil) pour la fréquence et la durée d'inertie : 20 trames par minute et 30 secondes.

Le tableau 3 présente les résultats de cette expérience. On dispose de deux postes clients (`cl1` et `cl2`) et d'un point d'accès (`ap`). Orchids et Snort sont lancés en même temps sur le même système. La première colonne du tableau indique l'ordre chronologique dans lequel les attaques sont exécutées. La deuxième colonne indique l'attaque en cours (source, destination et nombre de trames). Les troisième et quatrième colonnes affichent l'alerte signalée par Orchids et

[4] `http://www.aircrack-ng.org/`

Snort après l'observation de l'attaque en cours (l'information affichée inclut principalement les adresses source et destination des équipements impliqués dans l'attaque).

	Attaque	Orchids	Snort
1	ap $\xrightarrow{11}$ cl1		
2	cl1 $\xrightarrow{11}$ ap		
3	ap $\xrightarrow{11}$ cl2		
4	cl2 $\xrightarrow{11}$ ap		alerte cl2 $\rightarrow$ ap
5	ap $\xrightarrow{11}$ cl1	alerte ap $\rightarrow$ cl1	alerte ap $\rightarrow$ cl1
6	cl1 $\xrightarrow{11}$ ap	alerte cl1 $\rightarrow$ ap	pas d'alerte car ap est déjà dans la liste des victimes et la durée d'inertie n'est pas terminée
7	ap $\xrightarrow{11}$ cl2	alerte ap $\rightarrow$ cl2	alerte ap $\rightarrow$ cl2
8	cl2 $\xrightarrow{11}$ ap	alerte cl2 $\rightarrow$ ap	pas d'alerte car ap est déjà dans la liste des victimes et la durée d'inertie n'est pas terminée

Tableau 3: Comparaison du comportement d'Orchids et de Snort.

Les attaques correspondant aux lignes 1, 2 et 3 du tableau 3 ne provoquent aucune alerte car on n'atteint pas le nombre minimal de trames (20 trames dans ce cas) pour une cible donnée. Après la quatrième attaque, Snort a comptabilisé exactement 22 trames de désauthentification pour la cible ap, 11 pour cl1 et 11 pour cl2. Il affiche donc une alerte au moment où il reçoit la $20^{ième}$ trame ciblant ap et la source indiquée est celle qui a été observée en dernier. L'information est donc incomplète puisqu'elle omet l'attaque provoquée par cl1 sur ap. En effet, en consultant cette alerte, on pourrait penser que cl2 avait envoyé au moins 20 trames vers ap alors que ce n'est pas le cas.

Après la cinquième attaque, Orchids a comptabilisé 22 trames pour l'attaque ap $\rightarrow$ cl1 et 11 trames pour chacune des attaques cl1 $\rightarrow$ ap, ap $\rightarrow$ cl2 et cl2 $\rightarrow$ ap. Il affiche donc une alerte au moment où il reçoit la 20^{ieme} trame de l'attaque ap $\rightarrow$ cl1 et il affiche les bonnes adresses source et destination. Snort, de son côté, affiche une alerte pour la cible cl1 puisqu'il a comptabilisé 22 trames pour cette cible — dans ce cas, l'information affichée est correcte.

Après la sixième attaque, Orchids a comptabilisé 22 trames pour l'attaque cl1 $\rightarrow$ ap et il affiche l'alerte correspondante. Quant à Snort, il n'affiche aucune alerte car la cible ap a été ajoutée à la liste des victimes après la quatrième attaque et la durée d'inertie par rapport à cette victime n'est pas encore terminée.

Après la fin des attaques, on constate qu'Orchids a affiché en tout quatre alertes et Snort trois. Orchids a été plus complet dans les informations données par ses alertes. Snort a été le premier à signaler une alerte : la principale raison n'est pas dans la rapidité de son système d'analyse des trames mais plutôt dans son approche qui considère uniquement la cible de l'attaque — ce qui n'est pas nécessairement un avantage.

Nous avons aussi prélevé des mesures sur la consommation mémoire et le temps processeur pour chaque outil à l'aide de la commande `top`. Orchids a utilisé 136 Ko d'espace mémoire et 37 millisecondes de temps processeur durant cette attaque contre 8 Ko et 10 millisecondes pour Snort (la durée des mesures s'étale sur 20 secondes et elles sont effectuées à chaque seconde). Signalons de plus que Snort utilise au départ presque 4 Mo de données statiques *(Data Resident Set size)* alors qu'Orchids n'en utilise que 0,5 Mo.

5.2 Tests de l'attaque *rogue access point*

Pour réaliser cette attaque, il suffit d'avoir un point d'accès ou une interface 802.11 qui prend en charge le mode maître (mode *access point*). Une fois mis en marche, selon sa configuration, le point d'accès peut s'annoncer en diffusant des trames de type *beacon* ou attendre qu'une station client requière ses services. Dans les deux cas, il suffit d'observer une trame 802.11 impliquant un point d'accès inconnu (son adresse MAC) pour signaler une alerte.

Le détecteur d'intrusions maintient ainsi une liste dynamique des points d'accès inconnus. Les adresses sont ajoutées à cette liste au fur et à mesure que les points d'accès voyous sont détectés. Aucune nouvelle alerte n'est signalée pour une adresse déjà dans la liste. Chaque adresse de la liste est conservée durant une période de temps prédéterminée, après laquelle elle est retirée.

Comme pour Snort, la signature Orchids est paramétrée avec la liste des points d'accès autorisés ainsi que la durée de préservation d'une adresse inconnue dans la liste des points d'accès voyous.

Nous avons lancé les deux outils à un endroit avec cinq points d'accès voisins, dont trois étaient considérés comme points d'accès voyous, l'un d'entre eux configuré en mode passif. Le comportement des deux détecteurs d'intrusions fut semblable. Ainsi, les deux points d'accès voyous actifs ont été rapidement identifiés par les détecteurs d'intrusions et deux alertes ont été signalées. Puis, quelques instants plus tard, le troisième point d'accès (passif) a été identifié après qu'il eut fourni une réponse à une requête d'un client.

En termes de consommation mémoire et de temps processeur, Orchids et Snort sont équivalents (4 Ko et 3 millisecondes). Par contre, un avantage d'Orchids est que sa signature est nettement plus simple — la signature pour Orchids ne requiert que 18 lignes de code, alors que celle de Snort-Wireless en requiert près de 250.

5.3 Tests de l'attaque *ChopChop*

À notre connaissance, ni Snort ni aucun autre outil ne détecte l'attaque *ChopChop*, donc nous n'illustrerons que le comportement de notre outil.

L'outil `aireplay-ng` permet de réaliser l'attaque *ChopChop* en spécifiant l'adresse MAC de la station à attaquer. Le système se met alors en mode écoute et affiche la première trame chiffrée, provenant de la station ayant l'adresse MAC sous attaque. On choisit ensuite si on veut découvrir le contenu de cette trame ou si on veut continuer à chercher une autre trame. Lorsqu'on a fixé son choix sur la trame qu'on veut découvrir, *ChopChop* commence à effectuer les cycles de devinette.

Dans un premier temps, pour ce scénario d'attaque, nous avons lancé Orchids avec la signature d'attaque correspondant à la première signature décrite à la section 4.3.3, donc sans la synchronisation sur la taille des trames. Tel que prévu, les mesures du nombre d'instances de règles et d'états ainsi que d'alertes donnent des nombres élevés (voir la pre-

mière colonne du tableau 4) puisqu'une nouvelle instance de règle est créée pour chaque nouvelle trame reçue.

Dans le deuxième cas de figure, on a lancé Orchids avec une signature d'attaque correspondant à la signature décrite à l'aide des mots-clé cut et synchronize. La deuxième colonne du tableau 4 présente les mesures pour ce cas. Comme on peut le constater, une seule instance de règle est active durant l'attaque, ceci à cause du `cut("init")` effectué à l'entrée de l'état second_loop qui empêche la création de toute nouvelle instance de règle tant que la règle est active. Après chaque trois cycles de devinette[5], une nouvelle alerte est signalée dans le journal du détecteur d'intrusions.

	Sans cut et sans synchronize	Avec cut et synchronize
Nombre d'instances de règles	258	1
Nombre d'instances d'états	95582	9628
Nombre d'alertes	21864	31

Tableau 4: Résultats de l'attaque *ChopChop* pour une durée de 20 secondes.

Avec ces différentes expériences, nous avons donc pu confirmer les conséquences des choix de spécification de signatures décrits à la section 4.3.3, choix qui influencent de façon significative les performances du système de détection d'intrusions.

En ce qui concerne l'attaque *ChopChop*, on constate que la consommation mémoire ainsi que le nombre d'alertes sont relativement élevés par rapport à ce qu'on obtient avec l'attaque *deauthentication flooding*. Ceci confirme le besoin d'améliorer l'expressivité du langage de signatures d'attaques de façon à pouvoir identifier correctement les attaques basées sur des événements répétitifs (en boucle) semblables à l'attaque *ChopChop*.

6. TRAVAUX CONNEXES

La plupart des outils proposés dans la littérature pour détecter les intrusions dans les réseaux sans-fil sont présentés comme des *extensions* d'outils existants de détection d'intrusions. Toutefois, en pratique, plusieurs de ces extensions n'ont pas été intégrées dans les outils correspondants.

Par exemple, Neumerkel et Groβ [9] ont proposé une extension de l'outil *Bro Intrusion Detection System*[6]. Toutefois, bien que cette extension ait été proposée en septembre 2006, elle n'avait toujours pas été incluse dans cet outil, en date de juillet 2007. De plus, comme l'indiquent les auteurs eux-mêmes, la stratégie de détection adoptée est relativement rudimentaire, et aucune information sur les performances (mémoire, temps requis, nombre d'alertes) n'est fournie.

De même, il n'a pas été possible de tester l'outil proposé par Butti [4]. Cela dit, dans la description du fonctionnement de son outil, Butti explique que les trames 802.11 sont conservées dans un tableau, alloué de façon statique, et que des règles d'aggrégation et de corrélation entre trames sont ensuite appliquées sur ce tableau à l'aide de l'outil SEC *(Simple Event Correlator)*[7]. Ce choix de mise en œuvre a assurément un impact majeur sur le temps d'exécution et l'espace mémoire — d'ailleurs, Butti recommande la « limitation du nombre de règles faisant appel aux traitements dans le tableau de paquets » [4]. De plus, la syntaxe du langage de règles d'aggrégation et de corrélation fait que les spécification sont difficiles à lire. En outre, Butti indique que les règles sont maintenues selon une hiérarchie de dépendances à plusieurs niveaux, ce qui rend difficile leur maintenance.

Kismet est à la fois un renifleur, un outil d'attaque contre le protocole WEP et un détecteur d'intrusions spécifique pour les réseaux sans-fil 802.11. L'outil est codé en C/C++ et est distribué sous licence GPL *(GNU General Public License)*. La version décrite ici est Kismet-2006-04-R1 [7], laquelle définit une dizaine de signatures d'attaques codées en dur dans l'outil. En effet, Kismet ne propose pas de langage de signatures ; il faut donc à chaque fois modifier le code source et recompiler l'outil pour inclure une nouvelle description d'attaque. Kismet semble toujours actif (décembre 2007) et permet de détecter des attaques qui, pour la plupart, sont décrites dans le catalogue WVE *(Wireless Vulnerabilities and Exploits)*[8].

WIDZ[9] n'est plus actif depuis 2003. L'outil était codé en C et sa dernière version (v1.5) permettait de détecter uniquement trois attaques : *Association Request Frame DoS*, *NULL SSID Probe Request* et *Rogue Access Points*. Comme pour Kismet, les règles et attributs sont définis de façon statique ; en fait, même pour modifier un attribut d'une signature d'attaque — par exemple, le seuil permettant de déclencher une alerte —, il était nécessaire de modifier le code du programme et de le recompiler.

Snort, une référence en matière de détection d'intrusions, dispose aussi d'une extension spécifique aux réseaux sans-fil 802.11 (Snort-Wireless[10]). Cette extension, codée en langage C, définit une dizaine de signatures d'attaques spécifiques aux réseaux sans-fil. Snort-Wireless utilise la syntaxe de Snort pour la description de signatures d'attaques, à la différence que des adresses MAC sont utilisées pour les sources et destinations, plutôt que des adresses IP et des ports TCP. Les règles de Snort-Wireless ont donc l'allure suivante :

```
<action> wifi <mac> <direction> <mac>
         (<rule options>)
```

Snort-Wireless offre les mêmes actions de base que Snort : `alert`, `log`, `pass`, `activate` et `dynamic`. Par contre, de nouvelles options de règles (`<rule options>`), spécifiques au protocole `wifi`, sont définies en plus des options standards : `wep`, `ssid`, `pwr_mgmt`, `frame_control`, etc. Par contre, Snort-Wireless ne semble plus actif depuis juillet 2005, et il n'a pas non plus été intégré à son projet source (Snort).

[5] Ce choix a été fixé au chapitre précédent lors de l'identification de l'attaque *ChopChop*.

[6] `www.bro-ids.org`.

[7] `http://www.estpak.ee/~risto/sec/`.

[8] `http://www.wve.org/` : ce catalogue, semblable à CVE *(Common Vulnerabilities and Exposures)* et OSVDB *(Open Source Vulnerability DataBase)*, est spécialisé dans les vulnérabilitées des protocoles sans-fil

[9] `http://freshmeat.net/projects/widz/`

[10] `www.snort-wireless.org`.

7. CONCLUSION ET TRAVAUX FUTURS

Dans le cadre du travail décrit dans cet article, nous avons tout d'abord développé et intégré, dans l'outil de détection d'intrusions Orchids [13, 11], un module de capture et de découpage des trames 802.11. Par la suite, nous avons décrit, à l'aide de signatures, un certain nombre d'attaques, dont *ChopChop*, que nous sommes les premiers, à notre connaissance, à détecter. Ces attaques ont ensuite été mises en œuvre, puis détectées avec succès dans un environnement simple mais réaliste.

Goubault-Larrecq et Roger, les concepteurs d'Orchids, suggèrent que « la détection d'intrusions complexes est essentiellement une activité de *model-checking* de formules temporelles (décrivant des classes de scénarios d'attaques) sur des flux d'événements »[11]. Le succès que nous avons obtenu avec Orchids dans le contexte des réseaux sans-fil viennent renforcer cette observation. Cela dit, nous croyons avoir montré certaines limites d'Orchids pour détecter efficacement les attaques dans un environnement sans-fil, notamment l'attaque *ChopChop*. Entre autres, il pourrait être intéressant d'étendre le langage de signatures d'Orchids pour permettre une identification plus efficace d'une série d'événements répétitifs, en définissant par exemple un nouveau type d'état « volatile ».

Les détecteurs d'intrusions tentent, pour la plupart, d'appliquer des approches universelles sans nécessairement prendre en compte les spécificités des systèmes surveillés. Face aux nouveaux systèmes (réseaux mobiles, ubiquitaires), qui posent de nombreux défis aux détecteurs d'intrusions, nous croyons qu'il pourrait être utile de reconsidérer cette absence de spécialisation, et donc de repenser le problème de la détection d'intrusions dans un cadre plus spécifique (moins large). Dans le cas des réseaux sans-fil, à notre connaissance, il n'y a pas eu d'effort d'analyse et d'étude en ce sens. Nous croyons qu'en enrichissant les outils de détection fondés sur la vérification de modèles en fonction des caractéristiques et fonctionalités spécifiques des systèmes surveillés, on pourra construire des systèmes de détection à la fois efficaces et flexibles.

Remerciements

Cette recherche a été rendue possible grâce au support financier provenant d'une subvention du CRSNG. Nous désirons aussi remercier Julien Olivain, pour l'aide qu'il nous a apportée dans la compréhension du fonctionnement de l'outil Orchids.

8. REFERENCES

[1] W. Arbaugh, N. Shankar, and J. Wang. Your 802.11 Network has no Clothes. In *Proc. of the First IEEE Int. Conf. on Wireless LANs and Home Networks*, pages 131–144, Dec. 2001.

[2] J. Bellardo and S. Savage. 802.11 denial-of-service attacks: Real vulnerabilities and practical solutions. In *Proc. of the Twelfth USENIX Security Symposium*, pages 15–28, Washington, DC, USA, Aug. 2003.

[3] A. Bittau, M. Handley, and J. Lackey. The final nail in WEP's coffin. In *Proc. of the 2006 IEEE Symp. on Security and Privacy (S&P'06)*, pages 386–400, Washington, DC, USA, 2006. IEEE Computer Society.

[4] L. Butti. Détection d'Intrusion dans les Réseaux 802.11. In *Symp. sur la Sécurité des Technologies de l'Info. et des Comm. 2006*, pages 411–433. École Supérieure et d'Application des Transmissions, 2006.

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach. *Tenth ACM Symp. on Principles of Programming Languages*, pages 244–263, Jan. 1983.

[6] F. Guo and T.-C. Chiueh. Sequence number-based MAC address spoof detection. In *Proc. of the 8th Int. Symp. on Recent Advances in Intrusion Detection (RAID'05)*, pages 309–329, Seattle, WA, Sept. 2005.

[7] M. Kershaw. *Kismet 2005-08-R1*, Aug. 2005. Projet Kismet : `http://www.kismetwireless.net`.

[8] LAN/MAN Standards Committee of the IEEE Computer Society. *Information technology – Telecommunications and Information Exchange between Systems – Local and Metropolitan Area Networks – Specific Requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. IEEE Standards Board, ANSI/IEEE Std 802.11 (R2003) edition, 1999. Reaffirmed 12 June 2003.

[9] R. Neumerkel and S. Groβ. A Sophisticated Solution for Revealing Attacks on Wireless LAN. In *Proc. of the 3rd Int. Conf. on Trust, Privacy and Security in Digital Business (TrustBus '06)*, volume 4083, pages 223–232, Krakow, Poland, Sept. 2006. Springer-Verlag.

[10] J. Olivain. Plate-forme de détection d'intrusions : Analyse et corrélation temporelle d'événements en temps réel. Technical report, Laboratoire Spécification et Vérification, ENS Cachan, France, Nov.–Dec. 2003.

[11] J. Olivain and J. Goubault-Larrecq. The Orchids intrusion detection tool. In *Proc. of the 17th Int. Conf. on Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 286–290, Edinburgh, Scotland, UK, July 2005. Springer.

[12] A. Pnueli. The temporal logic of programs. In *18th IEEE Symp. on Foundations of Comp. Sci.*, pages 46–57. IEEE Comp. Soc. Press, 1977.

[13] M. Roger and J. Goubault-Larrecq. Log auditing through model checking. In *Proc. of the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 220–236, Cape Breton, NS, June 2001. IEEE Comp. Soc. Press.

[14] M. Salmanian, S. Leonard, J. H. Lefebvre, and S. Knight. Intrusion detection in 802.11 wireless local area networks. Technical Memorandum DRDC Ottawa TM 2004-120, Defence R&D Canada, Ottawa, ON, Sept. 2004.

[15] P. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie, and A. Petit. *Vérification de logiciels : techniques et outils du model-checking*. Vuibert, 1999.

[16] E. Tews, R.-P. Weinmann, and A. Pyshkin. Breaking 104 bit WEP in less than 60 seconds. Cryptology ePrint Archive, Report 2007/120, Apr. 2007.

[17] J. Wright. Detecting Wireless LAN MAC Address Spoofing, Jan. 2003. `http://home.jwu.edu/jwright/papers/wlan-mac-spoof.pdf`.

[11] `http://www.liafa.jussieu.fr/web9/manifsem/description_fr.php?idcongres=602`