RAPPORT TECHNIQUE DICO

Sous-projet 3, livrable 1: langages de détection d'attaques par signatures

Date : 10 juin 2002
Auteurs : J. Goubault-Larrecq, J.-P. Pouzol, S. Demri, L. Mé, P. Carle

Titre : Sous-projet 3, livrable 1: langages de détection d'attaques par signatures

Rapport No. / Version : 3.1/ 1

Statut : Draft

NetSecure Software
20, rue de l'Église
92200 Neuilly-sur-Seine, France
www.netsecuresoftware.fr

France Télécom R&D
38/40 rue du général Leclerc
92794 Issy-les-Moulineaux, France
www.rd.francetelecom.fr

IRISA/INSA
Campus de Beaulieu
35042 Rennes Cedex, France
www.irisa.fr

Supélec Rennes
Avenue de la Boulaie B.P. 28
35511 Cesson-Sévigné, France
www.supelec.fr

Laboratoire Spécification et Vérification
CNRS UMR 8643, ENS Cachan
61, avenue du président-Wilson
94235 Cachan Cedex, France
www.lsv.ens-cachan.fr

ONERA/DTIM
2, avenue Édouard Belin B.P. 4025
31055 Toulouse Cedex 04, France
www.onera.fr

FERIA/IRIT
118, route de Narbonne
31062 Toulouse Cedex 04, France
www.irit.fr

Résumé : Ce document présente les différentes caractéristiques des langages de détection d'attaques par signatures proposés par les participants de DICO, en effectuant une synthèse de leurs points communs et de leurs différences. En particulier, il apparaît que les constructions fondamentales de tout langage de détection d'intrusions complexes sont : 1. les filtres, 2. le **et**, 3. le **ou**, 4. la séquence, 5. le **sans**, et 6. les variables de corrélations ; constructions que nous décrivons, illustrons et dont nous montrons les formes qu'elles prennent dans les langages des participants. Nous explorons d'autre part diverses autres notions importantes et exhibées à des degrés divers par ces langages, dont la différence entre variables de corrélation et variables d'annonce, les possibilités d'extension et de modularité, les moyens mis à disposition de la personne responsable de l'écriture des signatures pour permettre une détection efficace des intrusions, et limiter les attaques par suffocation, entre autres.

Rédacteurs :

Contributeur		Organisme	Rôle	Visa
Jean GOUBAULT-LARRECQ	JGL	LSV	Responsable du livrable	
Stéphane DEMRI	SD	LSV	Contributeur	
Jean-Philippe POUZOL	JPP	IRISA	Contributeur	
Ludovic MÉ	LM	Supélec	Contributeur	
Patrice CARLE	PC	ONERA	Contributeur	

Table des matières

1 Introduction

L'approche de la détection d'attaques par signatures peut se résumer à ceci : un outil de détection d'intrusion I observe un flux d'événements E, et le compare à une liste de règles L définissant des situations anormales ou révélatrices d'une attaque en cours. Cette approche est réputée, comparée à l'approche comportementale (cf. sous-projet 2), n'engendrer que peu de faux positifs. Pour autant, cet avantage théorique ne se concrétise que si le langage de signatures utilisé permet d'exprimer une signature qui détectera les attaques visées tout en écartant de nombreuses actions parfaitement légales.

A contrario, La plupart des systèmes de détection d'intrusions actuels, comme Snort [Roe99], NetSecure Log, ou RealSecure de la société ISS, ne peuvent en sacrifient l'expressivité à la rapidité pure. Ils ne peuvent en général que détecter des événements isolés, à l'aide d'un mécanisme de pattern-matching de chaînes simple. On trouve par exemple dans l'outil NetSecure Log des règles de détection de la forme :

$$/\langle\text{mot}\rangle\ \#\langle\text{commentaire en texte libre décrivant l'attaque visée}\rangle$$

Le moteur NetSecure Log filtre les URI (Uniform Resource Identifiers) des requêtes http, y cherchant la suite de caractères ⟨mot⟩ ; toute ligne contenant le mot en question fait l'objet d'une action spécifiée à part. De façon similaire, Snort [Roe99] recherche des événements qui sont instance d'une conjonction d'expressions régulières.

Cette méthode, si elle est efficace, n'est pas assez discriminante : de nombreux faux positifs vont être générés. Pour autant, ce choix a été fait afin de permettre un travail *en ligne*. On souhaite analyser des flux d'événements réseau en temps réel, pour permettre une réaction immédiate, telle que la coupure d'une connection. Les langages d'entrée de ces outils sacrifient donc l'expressivité à la rapidité pure.

D'autres outils commerciaux, comme NetSecure Web, permettent de détecter des motifs plus discriminants dans les journaux d'événements (logs) http, mais ne fonctionnent qu'en mode *hors ligne* : le log est un fichier qui est analysé, par exemple la nuit, lorsque le trafic réseau est moins dense.

Or, de nombreuses attaques, pour lesquelles il faut réagir instantanément, sont caractérisées par des suites d'événements complexes. C'est un des buts du projet DICO, et en particulier du sous-projet 3. A cette fin, nos objectifs sont :

1. définir les caractéristiques d'un langage de description de signature permettant d'exprimer des enchaînements d'événements corrélés entre eux,
2. réaliser des prototypes d'IDS exploitant de tels langages,
3. dans la mesure du possible étudier les capacités et les limites d'un tel IDS, dans un contexte d'analyse en ligne du flux d'événements d'audit.

Les participants au sous-projet SP3 ont proposé des langages de signature : log$\mathcal{W}$eaver (LSV), Sutekh (IRISA), ADeLe (Supélec) et CRS (ONERA). L'ONERA, membre de DICO, a aussi proposé un langage de description d'attaque appelé Lambda qui présente des similitudes avec ceux présentés dans ce rapport, en particulier avec ADeLe. Lambda, dédié à la corrélation d'alertes, est décrit dans le livrable 5.1 du SP5. A titre d'exemple, le paragraphe 4.9 donne un exemple de description Lambda.

Tous ces langages permettent de décrire des signatures complexes. Ils utilisent un vocabulaire souvent très différent pour des réalités parfois plus proches qu'il n'y paraît. Il n'en reste pas moins qu'il existe une multitude de différences de style, de syntaxe et de pouvoir expressif. Écartant les deux premiers types de différences, ce document propose une synthèse des points communs et insiste sur leurs différences du point de vue de leur pouvoir d'expression.

Nous avons choisi dans ce document de comparer dans leurs versions actuelles les langages log$\mathcal{W}$eaver, Sutekh, ADeLe et CRS. En particulier, lorsque nous dirons que tel ou tel langage ne dispose pas de telle ou telle caractéristique, ceci signifiera rarement qu'on ne peut pas la rajouter. Les principes fondateurs de log$\mathcal{W}$eaver sont décrits dans [RGL01a], sections 4 et 5. Le manuel utilisateur de l'outil développé autour de ce langage (version 2.9 à la date d'écriture de ce document) est disponible en [GL01]. Les principes fondateurs de Sutekh sont décrits en [PD02], ceux de ADele dans [MM01]. La description de CRS est disponible auprès de l'auteur [Car01].

Structure du document. Le plan de ce document est le suivant. En section 2 nous examinons au travers d'un exemple les besoins présidant à la conception d'un langage de détection de séquences d'événements par signatures ; l'annexe A fournit quelques autres exemples représentatifs. Nous discutons brièvement en section 3 du format des événements en entrée des outils de détection d'intrusions, et de son influence sur le langage de signatures. Nous passons ensuite, en section 4, à la description des briques de base des langages de signatures des partenaires. Ils ont tous en commun des filtres (section 4.1), ainsi que des opérations communes de combinaisons de ces filtres (sections 4.2, 4.3). Dans le reste de la section 4, nous décrivons les constructions particulières à chacun des langages proposés, et les équivalents possibles de langage à langage. Si le format des événements en entrée a une influence sur le langage, la façon d'annoncer les attaques détectées en a une certainement plus grande, et nous examinons les implications des besoins en annonce sur les choix de conception des langages de signatures en section 5. Nous examinons finalement le point délicat de l'efficacité en section 6 : il est en effet nécessaire, pour que les outils de détection d'intrusions complexes par signatures soient réellement utilisables, qu'il soient efficaces tant en temps qu'en espace. S'il s'agit en grande partie d'une conséquence de choix d'algorithmes et de structures de données dans le moteur de détection, il n'en reste pas moins qu'il est nécessaire aussi que le langage fournisse les moyens de gagner en efficacité à la personne responsable de l'écriture des signatures. C'est d'autant plus crucial pour contrer les attaques par suffocation, dont nous parlerons en section 6.4. Nous concluons en section 7.

2 Besoins

2.1 Exemples typiques

Voici un exemple typique d'une attaque qu'il est possible de monter contre une machine qui offre des services de montage de disques à distance via le protocole NFS. L'exemple est tiré de [MM01].

En premier, l'attaquant demande à la machine cible la liste des services RPC (Remote Procedure Call) qu'elle offre par la commande

`rpcinfo -p` ⟨machine-cible⟩

et vérifie que ces services contiennent à la fois `portmapper` et `mountd`, qui permettent de dialoguer à distance avec le serveur NFS de la machine cible.

En second, l'attaquant va effectuer les deux opérations suivantes, dans un ordre arbitraire. Il va récupérer la liste des partitions disque montées sur la machine cible en tapant

`showmount -e` ⟨machine-cible⟩

et la liste des utilisateurs de la machine cible par

`finger @`⟨machine-cible⟩

Ceci étant fait, l'attaquant essaye de trouver un utilisateur parmi ces derniers dont la racine du compte est l'une des partitions listées ci-dessus, et est indiquée exportable. Ayant fixé son dévolu sur un utilisateur, disons X, il peut maintenant créer en local un compte `/home/`X, et monter le compte de X en local par

`mount -t nfs` ⟨machine-cible⟩`:/home/`X `/home/`X

L'attaquant n'a plus qu'à ajouter la ligne "+ +" au fichier `:/home/`X`/.rhosts` pour permettre à n'importe qui (et donc à lui-même en premier lieu) de se connecter via `rlogin` à la machine cible sur le compte de l'utilisateur X :

`rlogin` ⟨machine-cible⟩ `-l` X

On récapitule en annexe A une liste représentative d'attaques multi-événements qui ont servi de support à la réflexion sur les langages de signatures étudiés dans le cadre du sous-projet 3 de DICO.

2.2 Le besoin

Une des questions naturelles qui s'est posée au lancement du projet DICO était de recenser les besoins et par voie de conséquence les types d'attaques que chaque participant considérait comme typique. Nous ignorerons ici les attaques mono-événement, déjà bien traitées par les outils classiques, pour nous concentrer sur les attaques multi-événements comme l'attaque rpcinfo de la section précédente.

La description de cette attaque est très éloignée de ce qui est spécifiable à l'aide d'expressions régulières. Alors qu'un mécanisme simple d'expressions régulières permet de détecter l'utilisation de commandes telles que `showmount` ou `finger` notamment, leur seule utilisation n'est pas indicative d'une attaque : ce sont des commandes utilisées couramment par les ingénieurs systèmes par exemple. Déclencher des ripostes sécuritaires sur la simple détection de l'utilisation d'une telle commande serait donc catastrophique. Informer un officier sécurité via une alarme déclenchée à toute utilisation d'une de ces commandes ne servira qu'à la noyer sous des fausses alarmes inutiles, et peut même aider à cacher de vraies attaques, perdues au milieu d'un flux trop important de fausses alarmes. Il est ici nécessaire, en fait, de n'émettre une alarme ou de n'enclencher une riposte que lorsque tous les événements détectables sur la machine cible participant à l'attaque ont été vus.

Une signature doit permettre de décrire un ensemble de contraintes sur un enchaînement d'événements. Ces contraintes peuvent être groupées en trois catégories :

– **Contraintes de sélection** (filtrage) : elles permettent d'identifier quels sont les événements dit d'intérêt dans un log. Elles portent sur les valeurs portées par un unique événement.
– **Contraintes d'ordonnancement** (corrélation temporelle) : elles définissent un ordre (total ou partiel) que doivent respecter un ensemble d'événements qui ont été retenus comme événements d'intérêt.
– **Contraintes de corrélation logique** : elle définissent des contraintes globales sur les valeurs portées par différents événements.

3 Format d'entrée des événements

Dans chaque outil de détection d'intrusion (IDS), qu'il soit fondé comme dans ce sous-projet sur l'idée d'une détection d'événements complexes ou non, il est besoin de s'adapter à un format d'entrée du log.

Les informations analysées par un IDS sont produites par un ou plusieurs logiciels de collecte de données déployés sur le système d'information sous surveillance. Ces outils de collecte peuvent produire des renseignements de type *réseau* (ex : capture de paquets Ethernet avec la librairie `pcap` [TCP]), de type *système* (ex : capture d'appels systèmes sous Solaris avec Sun BSM [Sun00]), ou de type *applicatif* (ex : logs de serveurs web ou de pare-feux). Le medium physique (ex : inteface réseau, fichier, socket, mémoire partagée), l'encodage (ex : ASCII, Unicode, XDR, binaire) ainsi que la structure des données brutes varient selon le type des renseignements collectés et selon l'outil de collecte. Dans un but d'abstraction et donc de généricité, ces considerations doivent être transparentes du point de vue du langage de signatures. Ceci impose qu'une fois acquises, les données brutes soient pré-traitées avant d'être transmises au moteur d'analyse de l'IDS.

Ce pré-traitement remplit plusieurs missions. Tout d'abord, il effectue un travail d'analyse syntaxique et de normalisation afin d'abstraire la représentation concrète des données brutes. Il permet par ailleurs de structurer les différentes informations collectées. En effet, alors que certains formats de logs sont tres structurés (c'est le cas des enregistrements BSM ou des en-tête TCP/IP), d'autres formats se présentent sous la forme de chaines de caractères concaténées qu'il faut redécouper (on trouve dans cette catégorie la majorité des formats logs applicatifs, mais aussi le format `syslog` utilisé dans les logs systèmes Linux et les logs de routeurs Cisco).

Un tel pré-traitement permet de considérer les événements, du point de vue du langage de signature, comme des structures de données de type enregistrement, les *événements*, dont les différents constituants, les *champs*, peuvent être accédés simplement et de manière unique. Dans la pratique, deux représentations structurées sont utilisées :

– **Événements non typés** : ces événements se présentent sous la forme d'une collection de couples $(champ, valeur)$. Par exemple, un événement réseau correspondant à un paquet IP aura un champ "numéro de protocole" qui contiendra la valeur 6 dans le cas d'un paquet TCP. Cette représentation

a le mérite d'être extrêmement flexible car elle ne présuppose aucune structure particulière pour un événement. En revanche, l'utilisation de cette représentation implique que tous les champs ne sont pas forcément renseignés pour chaque événement : parler de "numéro de port" dans le cas d'un paquet ICMP n'a pas de sens, alors que cette information est pertinente dans le cas d'un paquet TCP. C'est notamment l'approche employée en log$\mathcal{W}$eaver (Dyade & LSV) et en Sutekh (IRISA).
– **Événements typés** : ces événements résultent d'une classification effectuée durant le pré-traitement des données brutes. Dans le cas de paquets réseau, on peut imaginer que le pré-traitement identifie le type de protocole dans l'en-tête IP et crée une structure de données correspondante en conséquence. Les événements typés possèdent ainsi des attributs définis a priori (de la même manière que des instances dans un langage de programmation objet à classe). Cette représentation a le mérite de produire des événements dont tous les attributs sont renseignés.
C'est l'approche employée par CRS et ADeLe. CRS pousse la notion de typage plus loin encore : les événements sont des *objets* dérivant d'une classe. Le filtrage préalable des événements selon leur classe, avant la détection d'intrusions proprement dite, permet une efficacité accrue. De plus, la notion d'héritage permet de réutiliser non seulement des champs des sur-classes, mais aussi des méthodes de filtrage élémentaire définies dans les sur-classes, ce qui évite de les réécrire.

Tous les langages des participants sont fondés sur cette notion d'événements, typés ou non. La liste des événements en entrée, le *log*, est mis sous un format universel vers lequel sont traduits, par des traducteurs spécifiques, les formats particuliers de logs (syslog Linux, BSM, en-tête TCP, etc.). Que ce format soit universel est important, au sens où ceci permet à chaque IDS de s'adapter facilement à un nouveau format d'entrée : seul un nouveau traducteur doit être écrit.

4 Langage de signatures, expressivité

Une signature synthétise un ensemble d'informations permettant au moteur d'analyse de l'outil de détection d'intrusions (IDS) de caractériser dans un log (une suite d'événements) la trace d'exécution d'un scénario d'attaque. Un certain nombre d'outils de reconnaissance de signatures multi-événements sont construits à partir d'algorithmes exploitant des systèmes de transitions ou de déduction en chaînage avant. C'est par exemple le cas de STAT [Por92], IDIOT [CDE$^+$96], ASAX [Mou97], ou P-BEST [LP99]. Les bases de connaissance utilisées par ces algorithmes se présentent sous la forme de collections de règles de la forme $Condition \rightarrow Action$. Les signatures décrites dans ce type de langages expriment comment reconnaître une sous-suite d'événements du log en spécifiant une action à effectuer lorsqu'un événement d'intérêt est rencontré. En conséquence, en décrivant *comment* reconnaître une sous-suite d'événements, les signatures deviennent très dépendantes de l'algorithme de reconnaissance sous-jacent. En outre, même en s'affranchissant des différences syntaxiques, traduire les signatures d'un langage vers un autre requiert un certain effort. Il est de même difficile de statuer de l'équivalence entre deux signatures exprimées dans des langages différents.

Finalement, un langage comme ceux mentionnés ci-dessus, qui insiste trop sur le *comment détecter* au lieu du *quoi détecter* permet trop souvent le *masquage* d'attaques. En ASAX par exemple, il n'est que trop facile, lorsque l'on souhaite détecter une suite, non nécessairement contigue, de deux événements `A` et `B`, d'oublier de relancer la règle surveillant l'apparition d'un événement `A` lorsque l'on vient de détecter `A` et que l'on cherche à surveiller l'apparition de `B`. Un tel oublie a des conséquences désastreuses : il suffira à un attaquant d'effectuer une action `A` qui ne sera jamais suivie de l'action `B` correspondante, puis d'effectuer une autre action `A` qui elle sera suivie de l'action `B` correspondante (donc l'attaque réussit), de sorte que ces deux dernières actions ne seront pas détectées. On dira que la deuxième action `A` a été *masquée*. Un langage déclaratif, en spécifiant le *quoi détecter* (`A` **puis** `B`) élimine tout risque d'erreur de spécification de l'attaque, et donc tout risque de masquage involontaire d'attaque.

Afin de pallier les difficultés exprimées ci-dessus et liées aux langages à base de règles, certains partenaires du projet ont proposé des langages déclaratifs qui s'attachent à décrire les caractéristiques et les relations entre les différents événements révélateurs d'une intrusion. Ces langages décrivent donc des signatures de manière indépendante de l'algorithme de reconnaissance qui sera utilisé durant la détection. Cette section présente une synthèse des capacités d'expression des langages/outils suivants :
– **ADeLe** (Supélec) : langage permettant de décrire l'ensemble des informations relatives à un scénario

d'attaque : pré-conditions d'exécution, étapes de réalisation, conséquences de l'exécution, traces laissées dans les fichiers de log (i.e., signature). Cette section s'intéresse exclusivement au sous-ensemble d'ADeLe permettant de décrire les signatures.
- **CRS** (ONERA) : outil de reconnaissance de chroniques. Le langage de CRS permet de décrire des enchaînements d'événements et d'ajouter des portions de code C++ permettant de vérifier des relations entre les événements.
- **logWeaver** (LSV) : model-checker de formules de logique temporelle spécialisé pour l'analyse à la volée de flots d'événements.
- **Sutekh** (IRISA) : langage logique dédié à la description de sous-suites d'événements dans une trace.

En réponse aux besoins exprimès en section 2.2, une signature dans un langage déclaratif définit tout d'abord des filtres pour sélectionner les événements d'intérêt. Ces filtres sont ensuite combinés pour former des signatures plus complexes avec des contraintes d'ordonnancement. Les contraintes de corrélation apparaissent enfin soit dans les filtres, soit dans des sections syntaxiques distinctes.

4.1 Filtres

Un filtre décrit des contraintes qui portent sur les champs d'un unique événement. Il regroupe au sein d'une seule expression les différentes contraintes de sélection ainsi que des contraintes de liaison avec des variables qui permettront d'exprimer la corrélation d'événements.

Contraintes de sélection. Dans le cas d'événements non typés, la sélection d'un événement d'intérêt s'exprime par des contraintes sur un sous-ensemble des champs d'un événement. Par exemple, la sélection d'un événement réseau correspondant à l'ouverture d'une connexion TCP sur le port 25 d'une machine est décrite en Sutekh par le filtre suivant :

```
[IP.proto = 6, TCP.SYN=true, TCP.dest.port 25]
```

et en logWeaver par :

```
.IP.proto==6, .TCP.SYN==1, .TCP.dest.port==25
```

Dans le cas d'événements typés, la sélection se fait en donnant le type de l'événement et en posant des contraintes sur ses attributs. Supposons que les événements réseau relatifs à des ouvertures de connexions TCP soient de type `TcpConnect` et possèdent un attribut `dest.port`. Dans la syntaxe de CRS, l'exemple précédent est décrit au moyen de l'expression :

```
TcpConnect->E <= #{#[Event,E]->dest.port == 25}#?
```

Cette expression signifie l'événement de type `TcpConnect` est nommé `E`, et doit vérifier que l'attribut `dest.port` est égal à 25. Dans la syntaxe d'ADeLe, la sélection du type de l'événement s'exprime de manière similaire :

```
<EVENTS>
   E: Network.Classification[0].name == "open tcp"
<EVENTS>
<CONTEXT>
   E.dest_port == 25
</CONTEXT
```

Cette expression décrit l'évenement `E` comme étant un évenement réseau (`Network`) ayant été classifié de type `open tcp`. La contrainte sur l'attribut de l'événement est décrite comme une propriéte de l'événement `E`.

Les contraintes de sélection ne se limitent pas nécessairement à des égalités entre des champs d'un événement et des constantes. D'autres critères de sélection sont envisageables, tels que la recherche d'expression régulière dans une chaine de caractères. Par exemple, la séléction d'un événement généré par l'appel d'un script CGI dans une requête HTTP se fera en comparant l'URL demandée avec une expression régulière contenant le nom du script. De manière plus générale, n'importe quel prédicat peut être utilisé pour selectionner un événement. Ces prédicats ne font toutefois pas réellement partie du langage de signatures en Sutekh, CRS ou ADeLe : on considère qu'ils sont définis dans une bibliothèque externe écrite dans un langage de programmation traditionnel. log$\mathcal{W}$eaver utilise un langage propre fournissant expressions régulières et quelques autres constructions standard [GL01] ; le but est de préserver la capacité de log$\mathcal{W}$eaver à appliquer des optimisations par analyse statique des signatures (section 6.2.2).

Nommage des éléments Afin d'être en mesure d'exprimer des contraintes de corrélation entre différents événements, il est nécessaire de nommer dans les filtres les éléments qui seront sujet à corrélation. Dans les quatre langages considérés, ce nommage est réalisé en liant tout ou partie des événements à une variable. Cette liaison associe à la variable l'information contenue dans l'événement répondant aux critères exprimés par les contraintes de sélection. L'utilisation dans les contraintes de corrélation des informations nommées dans les filtres sera étudiée en détail dans la section 4.2. Dans les langages log$\mathcal{W}$eaver et Sutekh, il est possible d'associer une variable à un champ d'un événement. L'exemple suivant cherche une ouverture de connexion TCP sur le port 25 et associe l'IP source de la connexion à la variable `X` :

```
[IP.proto = 6, TCP.SYN=true, TCP.dest.port = 25, TCP.source = X]
```

en Sutekh, et

```
.IP.proto==6, .TCP.SYN==1, .TCP.dest.port==25, .TCP.source X;
```

en log$\mathcal{W}$eaver. log$\mathcal{W}$eaver permet en fait plus généralement une forme de pattern-matching permettant de lier des variables à des sous-chaînes des champs [GL01].

Les langages ADeLe et CRS donnent la possibilité d'associer des événements à des variables. Ce nommage est utilisés pour les contraintes de sélection comme décrit précédemment (variable `E` dans les exemples). Une fois les événements nommés, il est possible d'accéder à n'importe quel attribut d'un événement et donc d'exprimer des contraintes de corrélation.

4.2 Corrélation entre les événements

L'utilisation de variables dans les filtres permet d'exprimer des contraintes globales sur les valeurs portées par les événements supposés constituer des instances d'une signature. Ces contraintes globales s'expriment de différentes manières selon les langages.

Sutekh et log$\mathcal{W}$eaver manipulent des variables logiques et utilisent un mécanisme d'unification. Ainsi, lorsqu'une variable est utilisée plusieurs fois dans différents filtre, elle exprime une contrainte d'égalité entre les champs associés à cette variable.

Les quatre langages permettent également de rajouter des contraintes globales sur les différentes variables dans les signatures. Ces constructions donnent également la possibilité de ne pas limiter les contraintes de corrélation à des contraintes d'égalité entre certains champs des évenements. Selon les langages, ces relations sont exprimées sous la forme de tests booléens (ADeLe, CRS, log$\mathcal{W}$eaver) ou sous la forme de contraintes unificatrices (Sutekh).

4.3 Opérateurs de combinaison de filtres communs

Les langages déclaratifs de description de signatures proposés par les partenaires du projet présentent une certain nombre de différences, mais tous proposent trois opérateurs de base pour combiner des filtres et ainsi former des signatures multi-événements. Ces opérateurs sont la séquence (ordre total), la conjonction (ordre partiel) et la disjonction.

4.3.1 Nécessité d'ordres total et partiel.

L'exécution d'un scénario génère une séquence totalement ordonnée d'événements que l'on suppose connue à l'avance : c'est l'hypothèse même de l'approche par scénarios/signatures. Le respect d'un ordre d'exécution de certaines étapes d'un scénario peut constituer une condition nécessaire pour l'exploitation d'une vulnérabilité. Toutefois, il est fréquent que des scénarios contiennent des étapes dites préparatoires suivies d'une dernière action qui exploite réellement la vulnérabilité. Alors qu'il est important d'exécuter l'étape finale en dernier, il est souvent possible d'intervertir les étapes intermédiaires. Ces interversions produisent des variantes de scénarios qui ont le même effet, mais une trace légèrement différente dans le log. Afin d'écrire des signatures capables de détecter des variantes de scénarios, un langage de signatures doit permettre de décrire synthétiquement les conditions nécessaires d'ordonnancement que les événements doivent respecter pour constituer une instance de la signature.

4.3.2 Séquence (ordre total).

L'expression de séquences dans les signatures permet de diminuer le nombre de fausses alertes en rejetant des événements s'il n'apparaissent pas au bon moment dans la traces.

Tous les langages considérés offrent un opérateur de séquence. On trouvera des exemples en annexe, par exemple en section A.1. L'exemple suivant (section A.3), en ADeLe, décrit la signature correspondant à l'exploitation d'un bogue dans le logiciel `ff.core` (Solaris) [CVE-1999-0442] permettant de renommer un fichier en outre-passant les permissions système. Le scénario enchaîne la création d'un lien nommé `/vol/rmt/diskette0` et l'exécution du programme `ff.core`. Dans cet exemple, l'exécution du programme `ff.core` avant la création du lien n'est pas significative d'une attaque. C'est pourquoi cette signatures combine deux événements avec l'opérateur de séquence noté "`;`"

```
<DETECT>
   <EVENTS>
      EO : System.Classification[0].name == "create link"
      E1 : System.Classificatiori[0].name == "exec"
   <EVENTS>
   <CONTEXT>
      EO.link_name == "/vol/rmt/diskette0"
      E1.prog.name == "/usr/openwin/bin/ff.core"
   </CONTEXT>
   <ENCHAIN>
      EO ; E1
   </ENCHAIN>
</DETECT>
```

En log*W*eaver, ceci s'écrit

```
.ACTION=="link", .DST=="^/vol/rmt/diskette0/$";
.ACTION=="ff.core", .SRC=="^/vol/rmt/diskette0/$";
```

Tandis qu'en Sutekh on utilisera l'opérateur de séquence `then`.

4.3.3 Conjonction (ordre partiel).

A l'opposé, imposer un ordre strict d'apparition d'événements dans la trace peut permettre à un utilisateur malveillant d'exploiter une vulnerabilité, sans se faire détecter par l'IDS, en intervertissant des étapes d'un scénario connu. L'utilisation de la conjonction de Sutekh et de CRS (parfois appelée "entrelacement" dans la littérature) permet de décrire synthétiquement l'ensemble des combinaisons possibles. L'exemple de l'attaque `rpcinfo` de la section 2.1 est un cas typique, où l'on souhaite que les commandes `rpcinfo`, `showmount` et `finger` apparaissent dans un ordre quelconque.

L'exemple d'exploitation d'un bogue dans le programme `admintool` (Solaris) décrit dans [PD01] nécessite la création de deux fichiers `pkginfo` et `pkgmap`. La signature de ces opérations dans un log d'appels système se décrit comme suit en Sutekh (l'opérateur `and` décrit la conjonction, et `#=` vérifie que le champ en membre gauche satisfait l'expression régulière en membre droit)

```
[ eventID = "AUE_CREAT" , path #= "*/pkginfo" ]
and
[ eventID = "AUE_CREAT" , path #= "*/pkgmap" ]
```

En logWeaver, on écrira

```
.eventID=="^AUE_CREAT$", .path=="/pkginfo$";
+ .eventID=="^AUE_CREAT$", .path=="/pkgmap$";
```

(Le symbole == est ici la recherche d'expression régulière ; le fait que les expressions régulières pour les champs `.path` ne commencent pas par `^` signifie que l'on doit rechercher les expressions régulières n'importe où dans le champ ; rappelons que par contre `$` impose que l'expression régulière se termine sur la fin du champ.)

Il existe une distinction subtile entre le + de logWeaver et l'opérateur `Non_Ordered` d'ADeLe d'une part, et le `and` de Sutekh et le `&` de CRS d'autre part. Dans certains cas rares, `A+B` en logWeaver ne reconnaît pas les mêmes scénarios que `A and B` en Sutekh : c'est le cas lorsque `A` et `B` sont exactement le même événement ; alors logWeaver demandera à reconnaître deux instances différentes de `A`, alors que Sutekh et CRS accepteront la présence d'un seul `A`.

4.3.4 Disjonction.

Lors de l'écriture de variantes d'un scénario d'attaque, il est parfois possible d'exécuter des actions différentes de celles du scénario original, dont les effets sont identiques, mais produisant des événement différents dans les logs. Toujours dans un objectif de synthétiser dans une seule signature les traces produites par les variantes d'un scénario, il est souhaitable qu'un langage de signatures puisse exprimer des alternatives dans les combinaisons d'événements.

Supposons, par exemple, qu'un scénario nécessite de renommer un fichier. Cette opération peut être effectuée directement au moyen de la commande `mv`, ou en enchaînant en séquence la commande de copie `cp` et la commande d'effacement `rm`. La formule logique pour logWeaver exprimant cette disjonction est la suivante (l'opérateur de disjonction est note "|")

```
  .op=="^mv$";
| (.op=="^cp$"; .op=="^rm$";)
```

L'opérateur similaire en CRS se note `||`.

logWeaver fournit un autre opérateur similaire, le *choix commis*. Quoiqu'il soit noté `||` comme l'opérateur de disjonction de CRS, il a une sémantique différente. Intuitivement,

```
   .op=="^mv$"; ... // effectuer A
|| .op=="^cp$"; ... // effectuer B
```

va chercher un événement dont le champ opération est `mv` ; s'il le trouve, logWeaver commet son choix à la première ligne, et tente ensuite de reconnaître `A`, en ignorant les scénarios qui commenceraient par une opération `cp` pour se poursuivre en `B`. De même, si l'opération n'est pas `mv` mais `cp`, logWeaver commettra son choix à la seconde ligne, en ignorant les scénarios qui pourraient commencer par un `mv` ultérieur. La raison d'être du choix commis est de permettre une efficacité accrue de la reconnaissance d'attaques (section 6.3.2), mais aussi d'améliorer la qualité des annonces d'attaques (section 5.4.2).

4.4 Non-occurrence d'événements : la construction "sans"

Le fait qu'un événement n'apparaisse pas dans un log peu constituer la manifestation d'une attaque contre le système. Dans certains cas en effet, un utilisateur n'est pas supposé effectuer une action s'il n'en a pas réalisé une autre au préalable. Par exemple, un utilisateur ne doit pas exécuter des commandes en tant que `root` s'il n'a pas auparavant exécuté la commande `su`. Cette signature s'exprime avec CRS de la manière suivante (l'opérateur de non-occurence est noté " `- [ ]` ")

```
(% Exec->ev1) - [ Exec->ev2]
   <= #{ ( (#[Exec,ev1]->uid == 0)
          && (#[Exec,ev2]->prog == "/bin/su")
         && (#[Exec,ev1]->auid == #[Exec,ev2]->auid) )#?
```

Cette chronique recherche des événements de type `Exec` (exécution de programme) et s'intéresse aux champs `uid` (*user identifier* : identité apparente d'un utilisateur), `auid` (*audit identifier* : identité réelle invariante durant toute une session d'utilisation) et `prog` (nom du programme exécuté). Une instance de la chronique est reconnue s'il existe un événement `Exec` avec un `uid root` qui n'a pas été précédé dans le log d'un `Exec` de `/bin/su` avec un `auid` identique.

Il est important de noter que l'utilisation de la non-occurence dans un log potentiellement infini nécessite l'utilisation de bornes pour la recherche de la partie "négative" de la construction. Dans cet exemple, l'événement `ev1` constitue la borne de fin de la recherche de `ev2`. Comme le scénario ne contient pas d'événement constituant une borne de début, CRS offre la possibilité d'insérer un événement fictif noté "`%`". Parmi les langages considérés, ce type d'événement fictif n'est proposé que par CRS. Dans les autres langages, il faut décider d'un événement qui déclenchera la recherche de la partie "négative". On peut par exemple choisir l'événement correspondant au login d'un utilisateur, comme cela est proposé dans la signature ADeLe suivante où la non-occurence est notée "`WITHOUT`" :

```
<DETECT>
   <EVENTS> ... </EVENTS>
   <CONTEXT> ... </CONTEXT>
   <ENCHAIN>
      { Login ; ExecAsRoot } WITHOUT ExecSu
   </ENCHAIN>
</DETECT>
```

L'utilisation de la non-occurence d'événements dans une signature permet également d'exprimer des conditions d'arrêt lors de la recherche, ce qui donne un moyen de prévenir la génération de fausses alertes. Considérons par exemple un programme connu comme vulnérable à une attaque de type *buffer overflow*. Un moyen de détecter une intrusion utilisant la faille de ce programme consiste à vérifier que durant son exécution, il ne lance jamais l'exécution de `/bin/sh`. Cette signature se décrit comme la séquence de deux événements : (a) l'exécution du programme vulnérable suivie de (b) l'exécution de `/bin/sh` avec le même numéro de processus (`pid`). Lorsque l'exécution du programme vulnérable se termine, il est nécessaire de stopper la recherche d'une exécution de `/bin/sh`. En effet, l'utilisation des numéros de processus étant cyclique, deux programmes distincts s'exécutant à des périodes différentes peuvent avoir le même numéro de processus. Une fois le programme vulnérable terminé, la recherche de l'exécution de `/bin/sh` peut aboutir et sera pourtant non significative génèrerant une fausse alerte. La signature Sutekh suivante exprime cette condition d'arrêt grace à l'opérateur de non-occurence noté "`if_not`") :

```
( [ eventID = "AUE_EXEC", path = "prog_vuln", pid = P ]
  then
  [ eventID = "AUE_EXEC", path = "/bin/sh", pid =P] )
if_not [ eventID = "AUE_EXIT", pid = P ]
```

log$\mathcal{W}$eaver permet aussi de coder les signatures négatives, mais d'une façon un peu plus détournée. La signature ci-dessus, par exemple, s'écrira :

```
.eventID=="^AUE_EXEC$", .path=="^prog_vuln$", .pid P;
{
   .eventID=="^AUE_EXIT$", .pid P; reject;
|| .eventID=="^AUE_EXEC$", .path=="^/bin/sh$", .pid P;
}
```

ce qui signifie, en première approche, que log$\mathcal{W}$eaver déclenchera une alerte s'il détecte l'exécution du programme vulnérable `prog_vuln` suivi soit de celle de `/bin/sh`, soit de la sortie du programme vulnérable. (`reject`, de même que `return`, accepte la signature courante ; alors que `return` provoque l'émission d'une alerte, `reject` l'inhibe. Les personnes qui connaissent Prolog pourront faire le rapprochement avec la construction `fail`.) log$\mathcal{W}$eaver n'affichant que les attaques les plus courtes (les *shortest runs* de [RGL01a]), si la sortie du programme vulnérable intervient avant l'exécution de `/bin/sh`, alors cette trace plus courte sera préférée, la signature sera acceptée sur `reject`, et aucune alerte ne sera annoncée, supprimant ainsi l'alerte sur l'exécution ultérieure de `/bin/sh` avec le même pid `P`. Ainsi le choix commis combiné avec la formule `reject` permet-t-il de réaliser la construction "sans" en log$\mathcal{W}$eaver.

4.5 Le partage d'événements

Le fait qu'un même filtre apparaisse plusieurs fois dans une signature peut être source d'ambiguïté sur le résultat de la détection, et plus particulièrement sur le nombre d'événements qui constituent une instance de la signature. Considérons plusieurs filtres notés A_1, A_2, B_1, B_2 et C. Dans le cas de la séquence

$$C \textbf{ puis } A_1 \textbf{ puis } C$$

aucune ambiguïté n'existe car l'opérateur de séquence impose la présence de 3 événements distincts pour constituer une instance. Il n'en est pas de même pour la signature

$$(A_1 \textbf{ puis } C \textbf{ puis } B_1) \textbf{ et } (A_2 \textbf{ puis } C \textbf{ puis } B_2)$$

qui nécessite de répondre à la question "*le filtre C correspond-il à 1 ou 2 événements dans la trace ?*" pour être non-ambigüe.

Cette ambiguïté est levée au regard de la sémantique de l'opérateur de conjonction des langages considérés. Dans le cas de CRS et de Sutekh, la signature exprime qu'au moins un événement doit être présent dans la trace pour constituer une instance. Il est en outre possible avec CRS de raffiner la signature afin d'imposer l'unicité d'un événement devant satisfaire deux filtres de la signature grace a un mécanisme de numérotation (ou indexation) des événements au sein d'une instance. Ainsi l'expression

$$(\quad A_1 \quad C:1 \quad B_1 \quad) \quad \& \quad (\quad A2 \quad C:1 \quad B_2 \quad)$$

impose le même index (et donc l'unicité) pour l'événement satisfaisant le filtre `C`. Un nommage similaire est proposé par ADeLe au moyen de la définition d'alias. On ecrira ainsi en ADeLe :

```
<ENCHAIN>
  Non_ordered{ { A1 ; XX:=C ; B1 } { A2 ; XX:=C ; B2 } }
</ENCHAIN>
```

Dans cette expression ADeLe, le même alias `XX` a été assigné au deux occurences de `C` signifiant l'unicité de l'événement de type `C` dans l'instance de la signature.

L'absence de cette construction dans les autres langages peut être comblée lorsque les événements du log contienne un champ les identifiant de manière unique dans la trace, tel qu'un timestamp ou un numéro d'ordre dans la trace. Il suffit alors d'exprimer une contrainte d'égalité sur ce champ dans les deux filtres.

Pour log$\mathcal{W}$eaver par exemple, si l'on considère que tout champ contient un champ `.line` de numéro de ligne unique, on pourra écrire l'exemple ci-dessus en forçant les deux numéros de lignes où sont reconnus les événements `C` à être la même ligne `L` :

```
  { A1;  .line L, C;  B1 }
+ { A2;  .line L, C;  B2 }
```

4.6 Boucles, comptage

Supposons que l'on souhaite détecter une attaque de spoofing. On dispose, sur un pare-feu, d'un fichier des événements réseau de la journée ou de l'heure écoulée par exemple, et l'on souhaite annoncer une attaque dès qu'une même machine destination a été la cible d'au moins 100 paquets réseau rejetés par le pare-feu. Une possibilité naïve pour la description de cette attaque sous forme de signature est de la décrire avec un opérateur de séquence, sous forme d'une mise en séquence de 100 événements E de type rejet de paquet avec la même variable destination.

Une solution plus élégante est de fournir une construction de répétition, "répéter 100 fois : reconnaître E". log$\mathcal{W}$eaver utilise une construction encore un peu plus générale, qui est inspirée par les opérateurs temporels étendus de Wolper [Wol83], et qui se matérialise sous la forme du mot clé `while`.

Cependant, si un intrus attaque 20000 fois la même machine destination, cette signature annoncera 200 attaques. (Davantage en l’absence de mécanismes du type de `synchronized`, cf. section 5.4.) On souhaite n’en annoncer qu’une. log*W*eaver permet de le faire simplement, en écrivant :

```
spoofing_attack synchronized (dest) anchored {
  while (loop: true) {
    .ACTION=="reject", .SRC $source, .DST dest, .line $line;
    || <<EOF>>; return | (_$loop>=100);
  }
}
```

ce qui exprime que l’on cherche une suite d’événements ayant tous `reject` dans le champ `.ACTION`, avec le même champ `.DST`, et ce jusqu’au moment où l’on rencontre la fin de fichier (`<<EOF>>`), auquel la signature est considérée comme reconnaissant l’attaque (`return`) à condition que le compteur de tour de boucle `_$loop` ait atteint une valeur supérieure ou égale à 100.

On constatera qu’il est ici nécessaire de disposer de variables, comme `dest`, qui vont servir à corréler les événements ayant le même champ `.DST`, mais aussi de variables dites *flexibles*, comme `$source`, qui peuvent prendre des valeurs différentes d’un événement

```
.ACTION=="reject", .SRC $source, .DST dest, .line $line;
```

à l’autre, et aussi de variables *de comptage*, comme `_$loop`. Les différents types de variables sont expliqués en Section 5.1, où nous reprendrons plus en détail l’exemple du spoofing. Les mots-clés `synchronized` et `anchored` sont expliqués en section 5.4.

Un autre exemple où les boucles sont utiles, cette fois-ci sans comptage, est donné en annexe A.2.

Il est possible de demander des répétitions d’événements en nombre arbitraire en CRS, en utilisant l’interface des chroniques avec le langage C++ : voir la section 5.1.3 pour un exemple. Il n’est pas entièrement clair que les deux solutions soient d’une puissance équivalente, mais elles suffisent toutes les deux à effectuer des comptages, ce qui est une activité courante dans l’analyse des logs.

En ADeLe, l’opérateur ˆ permet de définir de manière compacte une séquence. Ainsi :

```
<EVENTS>
  E : ...
</EVENTS>
<ENCHAIN>
  E^100
</ENCHAIN>
```

décrit une répétition de 100 événements de type `E`. Toutefois, il n’est pas possible d’exprimer des contraintes entre occurrences de l’événement répété.

On peut aussi détecter des répétitions d’événements dans CRS, grâce à l’interface avec C++. Par exemple, la chronique suivante vérifie que l’on effectue au moins 11 événements `a` avant de faire un événement `b` :

```
chronicle testvar has #{ int nba; }# is
  #{ nba = 0; }#
  a  <=#{ nba++; }#  <=#{ nba>10 }# b
```

Les signes # servent à inclure du code C++ dans la signature de la chronique `testvar`.

4.7 Modularité du langage de signature

Les langages déclaratifs présentés offrent une meilleure modularité dans l’écriture de signatures que ne le font les langages à base de règles. En effet, ces derniers imposent souvent de spécifier à l’intérieur même d’une règle le nom de la continuation de la recherche (par exemple, le nom de l’état suivant dans un automate, ou le nom de la règle suivante à activer). Pour permettre la modularité, les systèmes se doivent d’être hiérarchiques.

ADeLe, CRS et Sutekh permettent de donner des noms à des signatures afin de pouvoir les réutiliser dans d'autres signatures. Dans les exemples suivants, nous cherchons une séquence de trois événements de types `A`, `B`, `C` corrélés entre eux selon leurs champs respetifs `toto`, `foo` et `bar`. En supposant que la recherche de la séquence "B puis C" soit utilisée dans d'autre signature, nous souhaitons factoriser sa définition pour nous en resservir. La signature CRS correspondante est :

```
chronicle Sig1 is
  A->evA @Sig2->evSig2
   <= #{ #[A,evA]->toto == #[B,#[Sig2,evSig2]->evb]->foo }#?

chronicle Sig2 is
  B->evB C->evC
   <= #{ #[B,evB]->foo == #[C,evC]->bar} }#?
```

La chronique `Sig2` recherche un `B` puis un `C` corrélés selon leur champs `foo` et `bar`. La chronique `Sig1` recherche un `A`, puis attend la reconnaissance de la chronique `Sig2` ; enfin, le champ `toto` de l'événement `A` est comparé au champt `foo` de l'événement `B` qui a participé à la reconnaissance de `Sig2`.

La réutilisation de signatures en ADeLe est très proche, exepté le fait qu'il faille construire explicitement dans la section REPORT un événement synthétique correspondant au rapport de détection. On obtient ainsi en ADeLE :

```
Alert1 {
<EXPLOIT> ... </EXPLOIT>
<DETECTION>
   <EVENTS> A: ...
            B: Alert.Classification[0].name == "Alert2"  </EVENTS>

   <CONTEXT> X := A.toto
             B.foobar == X </CONTEXT>
</DETECTION>

Alert2 {
<EXPLOIT> ... </EXPLOIT>
<DETECTION>
   <EVENTS> B: ...
            C: ... </EVENTS>

   <CONTEXT> X := B.foo
             C.bar == X </CONTEXT>

   <REPORT> Alert.Classification[0].name := "Alert2"
            Alert.foobar := X </REPORT>
</DETECTION> }
```

Le langage Sutekh offre une modularité similaire à celle offerte par les fonctions dans les langages impératifs et fonctionnels ou les prédicats dans les langages logiques. Les signatures Sutekh peuvent être nommées et paramétrées. La composition de signatures s'exprime comme un appel de fonction.

```
A(X) = [ . toto=X ....... ].
B(X) = [ ... foo=X ...... ].
C(X) = [ ...... bar=X ... ].

Sig1 = A(V) then Sig2(V).

Sig2(T) = B(T) then C(T).
```

log$\mathcal{W}$eaver ne permet pas à l'heure actuelle de réutiliser de signatures précédemment définies.

4.8 Extension et ouverture

Les langages de signatures sont dédiés à la description d'enchaînements d'événements, et leur but n'est donc pas de proposer toutes les fonctionnalités d'un langage de programmation traditionnel. Toutefois, les besoins d'expression de corrélation sont parfois spécifiques à une signature ou un type de donnée et nécessitent donc de faire appel à des procedures externes. On peut avoir besoin par exemple d'effectuer des manipulations complexes sur certaines données (ex : opérations arithmétique, pattern matching sur des chaînes, masques sur des adresses IP, manipulations de dates et heures) ou de récupérer de l'information sur l'environnement (ex : interrogation du système de fichiers).

L'implémentation de l'outil CRS transforme les signatures en programmes C++. Le langage CRS offre alors la possibilité d'insérer du code C++ au sein des signatures (d'une manière similaire à ce qui est permis dans les générateurs de compilateurs tels que Lex/Yacc ou ANTLR). Les signatures ainsi décrites peuvent dans certain cas perdre de leur caractère déclaratif ; ceci permet en revanche d'effectuer toutes sortes de traitements durant la détection. L'utilisation de fonctionalités écrites dans une autre langage que C++ nécessite alors une interface pour l'édition de liens entre ce langage et C++.

Les langages ADeLe et Sutekh autorisent l'utilisation de fonctions ou prédicats "externes" dans les signatures. L'interfacage de ces fonctions avec le système de détection relève de l'implementation n'est pas considérés dans la description des langages.

L'outil log$\mathcal{W}$eaver pour sa part n'offre pas de possibilité d'extension pour des raisons liées à des soucis d'efficacité. Toutefois, le langage de log$\mathcal{W}$eaver propose de manière standard des fonctionnalités pour manipuler les types de données les plus usuels et particulierement les chaînes de caractères. La raison de ce choix sera présentée en section 6.2.2.

4.9 Lambda : un langage de corrélation d'alertes

Le langage Lambda, développé par l'ONERA, permet de synthétiser l'ensemble des informations concernant des attaques complexes contre des systèmes [CO00]. Sa vocation principale est de décrire les corrélations existantes entre les alertes émises par différentes sondes déployées sur un système afin d'identifier des scénarios d'attaques faisant intervenir plusieurs machines. En ce sens, ce langage présente de nombreuses similitudes avec le langage ADeLe, en particulier.

Une description de ce langage est proposée dans le document "Sous-projet SP5 - livrable 1" du projet DICO. Nous présentons ici un exemple de signature spécifiée avec Lambda afin d'illustrer sa syntaxe. A noter toutefois que, comme le langage ADeLe, Lambda permet de spécifier d'autres types d'informations concernant une attaque (par exemple, les pré-conditions d'éxécution) qui ne sont pas présentés ici. L'exemple étudié correspond au scénario d'attaque décrit dans la section 2.1 de ce document.

$$
\begin{array}{l}
\textbf{attack } \mathit{NFS\_abuse}(\mathit{TargetIP}) \\
\textbf{scenario } : ((E_1 ; (E_2 \,\&\, E_3)) \,\&\, E_4 \,\&\, E_5) ; E_6 \\
\textbf{where } \mathit{action}(E_1) = \texttt{rpcinfo -p } \mathit{TargetIP} \\
\quad \wedge\ \mathit{action}(E_2) = \texttt{showmount -e } \mathit{TargetIP} \\
\quad \wedge\ \mathit{action}(E_3) = \texttt{showmount -a } \mathit{TargetIP} \\
\quad \wedge\ \mathit{action}(E_4) = \texttt{finger @ } \mathit{TargetIP} \\
\quad \wedge\ \mathit{action}(E_5) = \texttt{adduser --uid } \mathit{UserId}\, U \\
\quad \wedge\ \mathit{action}(E_6) = \texttt{mount -t nfs } P \texttt{ /mnt} \\
\quad \wedge\ \mathit{actor}(E_1) = A \ \wedge\ \mathit{actor}(E_2) = A \\
\quad \wedge\ \mathit{actor}(E_3) = A \ \wedge\ \mathit{actor}(E_4) = A \\
\quad \wedge\ \mathit{actor}(E_5) = A \ \wedge\ \mathit{actor}(E_6) = A \\
\textbf{detection } : ((F_1 ; (F_2 \,\&\, F_3)) \,\&\, F_4 ; F_5) \\
\textbf{where } \mathit{action}(F_1) = \mathit{detect}(E_1) \\
\quad \wedge\ \mathit{action}(F_2) = \mathit{detect}(E_2) \\
\quad \wedge\ \mathit{action}(F_3) = \mathit{detect}(E_3) \\
\quad \wedge\ \mathit{action}(F_4) = \mathit{detect}(E_4) \\
\quad \wedge\ \mathit{action}(F_5) = \mathit{detect}(E_6)
\end{array}
$$

Dans cet exemple, la partie **scenario** décrit l'enchaînement des étapes à réaliser pour exploiter la vulnérabilité. Les connecteurs ";" et "&" correspondent respectivement à la séquence et la conjonction décrites dans la section 4.3 de ce document. La corrélation entre les événements est exprimée au moyen de l'unification de variables logiques comme avec Sutekh ou logWeaver. La partie **detection** décrit l'enchainements des événements visibles du point de vue de l'IDS. On remarque en effet que le scénario contient une action `adduser` qui est effectuée sur la machine de l'attaquant et donc invisible du point de vue de l'IDS. La partie détection décrit ainsi un *mapping* entre les actions du scénario et les événements contenus dans la trace, par exemple : $action(F_5) = detect(E_6)$.

5 Annonce des attaques (reporting)

Le langage de signatures d'un outil de détection d'intrusions doit permettre de caractériser les séquences d'événements révélateurs d'une attaque, mais également permettre de spécifier comment seront annoncées les attaques détectées. Les langages logWeaver, Sutekh, ADeLe et CRS et se différencient les uns des autres quant aux façons de spécifier les annonces.

5.1 Les différents types de variables

La présence de variables dans des signatures permet de nommer tout ou partie d'un événement durant la détection. Ce nommage permet alors, comme cela est largement illustré dans la section 4, d'exprimer des contraintes de corrélation sur les évenements constituant une instance d'une signature. L'utilisation de variables permet également de collecter de l'information durant la détection afin d'enrichir le rapport d'analyse. Un tel usage des variables conduit alors à distinguer les variables dites de corrélation des variables dites d'annonces. Par ailleurs, la combinaison de l'usage des variables et des boucles (cf. section 4.6) dans une signature fait apparaître la nécessité de distinguer les variables dites flexibles des variables dites rigides.

5.1.1 Variables de corrélation et variables d'annonce.

L'utilisation de variables dans des signatures permet de nommer tout ou des partie des événements constituant une instance d'une signature durant la détéction. Une notion introduite avec Sutekh dans [PD02] est la différence entre variables *de corrélation* et variables d'*annonces*. Considérons l'attaque `rpcinfo` présentée en section 2.1. Dans tous les langages présentés ici, l'adresse IP de la machine cible de l'attaque sera récupérée dans une variable. Cette variable, appelons-la ⟨machine-cible⟩, sert deux buts :

- d'abord, vérifier que les commandes `rpcinfo`, `showmount`, `finger`, `mount`, `rlogin` visent la même machine cible. Le fait que la variable ⟨machine-cible⟩ doive avoir la même valeur dans les cinq commandes est important lors de la phase de détection des attaques, et diminue de façon drastique le nombre de faux positifs : ⟨machine-cible⟩ est une variable de *corrélation*.
- Ensuite, la valeur de ⟨machine-cible⟩ sera utilisée dans l'annonce de l'attaque : le nom de la machine attaquée est une information vitale pour prendre des mesures defensives en réponse à cette attaque. La variable ⟨machine-cible⟩ est à ce titre une variable d'*annonce*.

Cet exemple montre une variable utilisée simultanément à des fins de corrélation et d'annonce. D'autres variables peuvent parfois être utilisées dans les annonces mais pas pour corréler des évenements. Dans le cas d'une analyse hors-ligne, rapporter les numeros des lignes contenant les evenements dans le log peut permettre d'extraire *a posteriori* les événements constituants une instance de la signature dans le but de fournir la preuve de l'attaque. Il arrive enfin qu'une variable soit utilisée pour la corrélation mais pas dans l'annonce. C'est le cas lorsque l'on utilise des dates pour calculer des timeouts : on ne souhaite pas nécessairement annoncer les dates des événements, mais il est important de les comparer durant la detection.

En Sutekh, ADeLE et CRS, la distinction entre ces deux types de variables n'est pas syntaxiquement exprimée dans les signatures. Toutes les variables sont potentiellement des variables de corrélation ou d'annonce. logWeaver pour sa part utilise un certain nombre de règles syntaxiques définissant quelles variables sont des variables d'annonces (toutes le sont, sauf celles dont le nom commence par `$$`, qui sont des variables flexibles - voir section 5.1.2 - que l'on ne souhaite pas annoncer, et celles dont le nom commence par `_$`, qui sont des variables de comptage, cf. section 5.1.3). Toutes les variables de logWeaver sont potentiellement des variables de corrélation.

5.1.2 Variables rigides et variables flexibles.

Dans le cas de signatures ne contenant pas de boucles, lorsque la detection d'une signature aboutit, l'IDS est en mesure de produire une valuation pour chacune des variables presentes dans la signatures. La présence d'une boucle dans une signature change la donne car le même nom de variable va être utiliser pour nommer des informations présentes dans différentes occurences d'un évenement spécifié dans le corps de la boucle. La distinction entre variables de correlation et variables d'annonce entre alors en jeu. Dans le cas de variables de correlation, la même valeur doit être associée a la variable dans toutes les occurences des évenement : on parle de variable rigides. Dans le cas de variables d'annonces, celles-ci doivent accumuler les différentes valeurs constatées dans les différentes occurences :on parle de variables flexibles. Dans les cas de variables utilisées simultanément à des fins de corrélation et d'annonce, la corrélation prime : on utilisera alors une variable rigide.

Sachant que Sutekh ne propose pas à l'heure actuelle de construction de boucle, et qu'ADeLE, bien que proposant cette construction, permet difficilement de correler les occurences d'evenements entre elles (cf. section 4.6), cette notion de variables rigides/flexibles n'apparait pas dans ces langages.

Cette distinction est principalement présente dans le langage logWeaver. Par exemple, la signature suivante détecte une forme de spoofing, où le celui-ci est identifié comme une suite d'événements correspondant à des paquets IP rejetés par un pare-feu dont l'adresse destination est toujours la même mais dont l'adresse source peut varier :

```
spoofing_attack synchronized (dest) anchored {
  while (true) {
    .ACTION=="reject", .SRC $_source, .DST dest, .line $line;
    || <<EOF>>; return;
  }
}
```

Dans cet exemple, la distinction entre variables flexibles et rigides s'exprime syntaxiquement par la présence d'un `$` au début de l'identificateur de la variable. Par convention, en logWeaver, les variables flexibles sont celles dont le nom commence par `$`. Le fait que l'adresse destination doive toujours être la même est représenté par le choix de stocker le contenu du champ `.DST` (destination) dans la variable rigide `dest`. L'adresse source peut elle varier, comme l'indique l'utilisation la variable flexible`$_source`. De même, le numéro de ligne de l'événement dans le log `.line` est associé à la variable flexible `$line`.

S'il est facile d'annoncer les valeurs des variables rigides, les valeurs des variables flexibles offrent davantage de variantes. logWeaver propose deux solutions. De façon standard, elles sont affichées sous forme de la liste des valeurs qu'elles ont prises lors des différents tours de boucles Ceci est vrai sauf pour les variables dont le nom commence par `$_`, qui sont vues non pas comme des listes mais des ensembles (les doublons sont éliminés). Ainsi, l'annonce de la détection de la signature donnée en exemple contiendra la valeur associée a la variable rigide `dest`, la liste des lignes où apparaissent les événements d'intérêt dans le log associée à la variable `$line`, et l'ensemble des adresses IP source contenues dans ces événements associé à la variable `$_source`.

5.1.3 Variables de comptage.

Il existe encore d'autres situations où ce n'est pas tant le fait qu'un certain type d'événements apparaisse plus de 100 fois qui soit important, mais le nombre d'apparition de ces événements tout court. Reprenons l'exemple de l'attaque par spoofing de la section précédente. Dans cette section, on visait à annoncer une attaque sitôt détectée une suite de 100 paquets IP rejetés avec la même adresse IP destination.

Il arrive que l'analyse de logs soit effectuée *hors-ligne*, et qu'on cherche plutôt à savoir *combien* de paquets IP rejetés visaient la même destination dans l'activité réseau de la journée passée. Le langage de signatures doit alors fournir le moyen d'annoncer des comptes d'événements.

En logWeaver, les variables de comptage (dont le nom commence par _$) servent ce but. Par exemple, la signature :

```
spoofing_attack synchronized (dest) anchored {
  while (loop: true) {
    .ACTION=="reject", .SRC $_source, .DST dest, .line $line;
    || <<EOF>>; return;
  }
}
```

collationne, via la ligne `.ACTION==...`, toutes les suites d'événements correspondants à des paquets IP rejetés, en accumulant l'ensemble des adresses source dans `$_source` et la liste des numéros d'événements dans `$line$`, pour chaque destination possible `dest`, jusqu'au moment où un événement "fin de fichier" (`<<EOF>>`) est détecté, ce qui provoque la réussite de la reconnaissance de la signature (`return`). À l'étiquette `loop` est implicitement associée une variable `_$loop` qui compte le nombre de fois où le corps de la boucle `while` a été reconnu. La variable de comptage `_$loop` est une variable d'annonce, et est donc utilisée pour annoncer le nombre d'événements dans l'attaque. (La variable `_$loop` sert aussi de variable de corrélation : voir aussi la section 4.6.)

Rappelons que CRS dispose pour ce faire d'une interface avec C++, qui permet d'appeler du code qui va compter le nombre d'apparitions d'un événement. Une fois une attaque reconnue, on peut encore une fois invoquer du code C++, cette fois pour afficher la valeur du compteur.

Sutekh ne proposant pas de boucle, n'a pas de notion de compteur. Si ADeLe dispose de moyens d'itération (section 4.6), il ne dispose pas de moyen d'annoncer des valeurs de compteurs.

5.2 À quel moment annoncer une attaque ?

En première approche, il semble évident que le moment où l'on doit annoncer une attaque est celui où l'outil vient de reconnaître une suite d'événements spécifiée par une signature. C'est ce que supposent par défaut logWeaver, CRS et ADeLe : dès que l'attaque est reconnue, elle est annoncée. Cependant il existe des situations où l'on souhaite annoncer une attaque avant que la signature soit entièrement reconnue. Par exemple, considérons une attaque consistant à détecter la création d'une voie d'accès détournée (*backdoor*) sur un système puis son utilisation proporement dite. Il est souhaitable de signaler la création de la backdoor le plus tôt possible pour éventuellement la condamner avant son utilisation, sans pour autant négliger de détecter son utilisation dans le cas où elle n'aurait pas été condamnée à temps.

Il est donc nécessaire que le langage de signatures permette de specifier à quel moment une alerte (éventuellement partielle) doit être émise. Sutekh propose ainsi d'annoter, au moyen du mot clé `trigger`, le texte d'une signature par des appels a des fonctions externes produisant des rapports de détection. Ainsi, la signature

```
Sig = ( (/* ... SubSignature 1 ... */ trigger Alert1 )
        then
        (/* ... SubSignature 2 ... */)
      ) trigger Alert2.
```

exprime la séquence (connecteur `then`) de deux sous-signatures et déclenchera d'une part l'alerte numéro 1 après avoir détecté la première moitié, et d'autre part l'alerte numéro 2 lorsque l'intégralité de la signature sera détectée.

Le besoin d'exprimer à quel moment produire une alerte est présent sous une autre forme lorsque la signature contient une boucle. Supposons que l'on cherche la répétition d'une action telle que celle presentée avec l'attaque par spoofing dans section 5.1.2. Cet exemple dédié à une analyse hors-ligne cherche les répétitions de paquets rejetés jusqu'à ce que la fin du fichier de log soit atteinte. Cette signature ne peut être utilisée dans le contexte d'une analyse en-ligne (pas de fin de fichier dans ce cas), il faut donc la spécifier autrement. Par exemple, l'utilisateur peut exprimer : "chercher une répétion supérieure à 100 occurences". Cette signature peut se décrire comme suit avec logWeaver :

```
spoofing_attack_early synchronized (dest) anchored {
  while (loop: _$loop<100) {
    .ACTION=="reject", .SRC $source, .DST dest, .line $line; } }
```

Cette signature exprime une boucle tant que la variable `_$loop` est inférieure 100. Toutefois, cette signature ne correspond pas réellement au besoin exprimé. En effet, si le log contient 10000 paquets rejetés, l'outil de détection produira 100 alertes correspondants chacunes à 100 paquets rejetés. Ces 100 alertes seront distinctes les unes des autres et ne seront pas révélatrices d'une seule répétition supérieure à 100. En réalité, le besoin exprimé ci-dessus ne peut pas être spécifié avec log$\mathcal{W}$eaver car le langage se base sur le fait qu'une émission d'alerte est associée avec la fin de détection de la signature. Pour répondre au besoin, il faudrait que log$\mathcal{W}$eaver possède une construction similaire à celle de Sutekh pour, par exemple, lever une alerte à chaque fois que le compteur est un multiple de 100 tout en continuant la detection de la répétition. Ainsi, il serait possible à un instant donné d'émettre une alerte signifiant la répétition de 500 occurences, puis une autre à 600, etc….

Ceci est faisable en CRS. Il suffit pour cela d'utiliser l'interface avec C++ et d'afficher l'alerte au moyen des méthodes d'entrées-sorties fournies par C++, par exemple :

```
#{ cout << "spoofing_attack: dest=" << #[SpoofingEvent,dest]->value; }#
```

5.3 Construction du rapport de detection

Nous avons vu en section 5.2 que les langages CRS, Sutekh, ADeLe et log$\mathcal{W}$eaver se distinguent par leur capacité à spécifier *quand* émettre un rapport de détection. Ils se différencient également par la *manière* de construre ce rapport :

- **ADeLE** : les travaux initiaux sur ADeLe s'inscrivant dans un contexte de corrélation d'alertes, le format retenu pour construire les rapport est IDMEF. Ainsi. la section `<REPORT> ... </REPORT>` d'une section ADeLe permet décrire comment remplir les différents champs d'un message IDMEF
- log$\mathcal{W}$eaver : le prototype construit autour de log$\mathcal{W}$eaver produit des rapports de détection dans un format propriétaire. Le rapport est constitué du nom la signature, ainsi que des valeurs associées aux variables de la signature. La manière de rapporter les valeurs des variables flexibles de log$\mathcal{W}$eaver sous forme de liste ou d'ensemble est décrite dans la section 5.1.2. L'ajout à cet outil de procédures générant des messages dans un format plus standard tel que l'IDMEF ne consitue pas en soi une difficulté.
- **CRS** : le lien fort entre CRS et C++ permettant d'insérer du code C++ dans les signature permet de générer des alertes dans n'importe quel format. La construction du message produit se fait donc en utilisant les primitives offertes par C++.
- **Sutekh** : la vocation première de Sutekh étant d'être un langage dédié à la description d'enchaînements d'événements, la construction d'un rapport de détection n'est pas prévue d'être décrite dans le langage. Sutekh fait la supposition que des fonctions externes sont fournies, et la construction `trigger` permet d'y faire appel.

5.4 Familles d'attaques

Lors de la recherche d'une signature dans un log, il est possible que plusieurs sous-suites d'événements constituent une instance de la signature. Les langages de signatures tels qu'il ont été décrits jusqu'à présent permettent de définir des contraintes sur les événements d'une instance, mais ne permettent pas d'exprimer quelles instances doivent être annoncées par le moteur d'analyse. Quelques éléments de réponse sont proposés dans les langages Sutekh, CRS et log$\mathcal{W}$eaver. Cet aspect de la spécification n'est pas abordé dans le lange ADeLe.

Le plus simple de ces mécanismes est certainement la coupure, proposée par CRS notamment, qui provoque un élagage de l'espace de recherche, et donc des attaques annoncées. Nous commençons cependant par décrire l'approche des classes d'équivalences de Sutekh, qui correspond aussi au `synchronized` de log$\mathcal{W}$eaver, pour raffiner petit à petit les moyens disponibles et revenir à la coupure en section 5.4.2.

5.4.1 Équivalences, synchronisation de variables.

L'approche des classes d'équivalence de Sutekh est la suivante. Étant donné un sous-ensemble $x_1, \ldots, x_n$ des variables présentes dans la signature, une relation d'équivalence est définie permettant de partitionner les instances de signatures en attaques vues comme équivalentes. La définition de cette relation d'équivalence permet alors à l'algorithme proposé dans [PD02] de ne signaler qu'une seule instance de la signature par valuation de ces variables : l'instance qui commence au plus tôt et termine au plus tôt. (À noter cependant que l'algorithme proposé ne prend en compte qu'un sous-ensemble de Sutekh constitué des filtres, de la séquence et de la conjonction).

Nous ne décrirons pas cet algorithme ici, ce qui serait hors de propos. Nous nous bornerons à remarquer qu'une construction de langage de signature est nécessaire pour permettre la définition de relation d'équivalences à la Sutekh.

log$\mathcal{W}$eaver propose un trait de langage similaire, identifié par le mot-clé `synchronized` ([GL01], section 5.3) : une signature S dont le nom est suivi de `synchronized` $(x_1, \ldots, x_n)$, où $x_1, \ldots, x_n$ sont des variables rigides, n'annoncera qu'une attaque parmi toutes celles qui correspondent à la signature S, qui se recouvrent et qui ont les mêmes valeurs des variables $x_1, \ldots, x_n$. L'exemple suivant, tiré de [GL01], montre une utilisation de `synchronized` visant à détecter deux tentatives de login en moins d'une heure sur une même machine :

```
bad_authentication synchronized(mach,uid) {
  .machine mach, .program "^PAM_pwdb$", .line line1, .date date1,
      .comment "authentic.*fail.*uid=([0-9]+)" { uid="\\1" };
  .machine mach, .program "^PAM_pwdb$", .line line2, .date date2,
      .comment "authentic.*fail.*uid=([0-9]+)" { uid="\\1" }
      | (date2 <= date1+3600);
}
```

Ceci se lit comme suit : la première ligne détecte un événement déclenché par le système d'authentification PAM de Linux, annonçant un login rejeté (on le reconnaît au fait que le champ `.comment` contient la chaîne `authentic` puis `fail`, etc.) ; la syntaxe `{ uid = "\\1" }` permet de récupérer la sous-chaîne du champ `.comment` qui indique l'identificateur utilisateur (uid ; le nom de login) sous lequel la tentative de login a été effectuée. La deuxième ligne est similaire ; en particulier les variables rigides `mach` et `uid` doivent avoir des valeurs identiques. La contrainte `(date2 <= date1+3600)` exprime le fait que le deuxième événement doit arriver moins de 3600 secondes après le premier.

La signature `bad_authentication` est déclarée `synchronized(mach,uid)`. Cette construction prend tout son sens au regard du fonctionnement de l'outil développé autour de log$\mathcal{W}$eaver et de la sémantique dite des *shortest runs* formalisée dans [RGL01a]. L'idée fondamentale de cette sémantique — indépendamment de toute notion de synchronisation de variables — est que lorsque plusieurs suites d'événements peuvent être reconnues comme étant instances d'une même signature, et si l'on considère les familles de ces événements qui commencent au même événement, alors log$\mathcal{W}$eaver annoncera une et exactement une attaque par famille ([RGL01a], théorème 4.8). Supposons qu'un même utilisateur U tente de se logger quatre fois de suite sur la même machine. La sémantique des shortest runs de log$\mathcal{W}$eaver va annoncer une attaque `bad_authentication` pour la paire formée du premier et du deuxième login, pour la paire du deuxième et du troisième login, pour la paire du troisième et du quatrième. L'effet de `synchronized`, maintenant, est de forcer log$\mathcal{W}$eaver à reconnaître que la deuxième attaque recouvre partiellement la première, et comme les valeurs des variables synchronisées `mach` et `uid` sont les mêmes, la seconde ne sera pas affichée. Par contre, la troisième, qui ne recouvre pas la première, sera annoncée. D'autres exemples sont donnés dans [RGL01a], section 4.2.

Le mot-clé `synchronized` de log$\mathcal{W}$eaver offre donc une fonctionnalité équivalente à celle des classes d'équivalences de Sutekh, qui ont été inventées indépendamment. Ici, la première et la deuxième attaque seront déclarées équivalentes en Sutekh.

5.4.2 Coupures

La coupure de CRS (notée "`!`") est un autre moyen pour limiter le nombre des attaques qui vont être annoncées comme reconnues par une signature. La coupure exprime que seul le premier événement du log satisfaisant un filtre peut être utilisé pour construire une instance. Ainsi, la chronique `(!A B C)` reconnaîtra des séquences d'événements satisfaisant les 3 filtres, mais ne retiendra que le premier événement `A` du log.

log$\mathcal{W}$eaver permet aussi de déclarer une coupure dans le style de CRS. Au lieu de précéder la première ligne de la signature d'un point d'exclamation, on fait suivre la déclaration de nom de la signature du mot-clé `anchored`. C'était le cas dans l'attaque de spoofing de la section 4.6, où l'on avait écrit :

```
spoofing_attack synchronized (dest) anchored {
  while (loop: true) {
    .ACTION=="reject", .SRC $source, .DST dest, .line $line;
    || <<EOF>>; return | (_$loop>=100);
  }
}
```

Ici `anchored` force le premier événement qui correspond à la ligne

```
.ACTION=="reject", .SRC $source, .DST dest, .line $line;
```

à être le premier événement de l'attaque. Plus précisément, pour chacune des valeurs prises par les variables synchronisées (ici, l'adresse destination `dest`), la signature `spoofing_attack` reconnaîtra une et une seule attaque `spoofing_attack`. En particulier, `anchored` n'a pas l'inconvénient de masquer des attaques portant sur une destination par des attaques portant sur d'autres.

Le trait de langage `anchored` peut être vu comme un raffinement supplémentaire des classes d'équivalence de Sutekh, qui réduit encore d'autant le nombre d'annonces d'attaques similaires, ou comme une coupure CRS, variable synchronisée par variable synchronisée. On notera que le choix commis `||` est en général un autre trait de langage qui permet de réduire le nombre d'annonces ; il s'agit en fait d'une autre forme de coupure.

5.4.3 Mise en garde concernant ces approches

L'utilisation de la coupure ou des classes d'équivalence de Sutekh et log$\mathcal{W}$eaver peut dans certaines situations amener l'IDS à ne pas considérer des sous-suites d'événements qui constitueraient des instances de la signature en l'absence de coupure. Le placement correct des coupures relève donc de la responsabilité du programmeur. Toutefois, dans l'approche par classes d'équivalence de Sutekh et log$\mathcal{W}$eaver, il a été montré [PD02] que lorsque le sous-ensemble des variables définissant les classes d'équivalence contient toutes les variables apparaissant plusieurs fois dans la signature, la totalité des instances sont reconnues par l'algorithme de détection. Nous reviendrons à ces constructions en section 6, où nous appellerons les coupures des élagages *rouges* (qui peuvent perdre des attaques) et l'approche par équivalence des élagages *verts* (qui ne perdent pas d'attaques essentielles).

6 Efficacité

6.1 Efficacité en temps, en espace

Un aspect crucial pour l'utilisabilité des outils de détecter d'intrusions par signatures est l'efficacité. Si la plupart des outils utilisés encore aujourd'hui ne fournissent essentiellement que des reconnaissances d'expressions régulières *mono-événement*, c'est en très grande partie parce que la détection multi-événements est perçue comme trop coûteuse.

Les outils log$\mathcal{W}$eaver, Sutekh et CRS sont le résultat d'une vision différente, selon laquelle il est possible d'effectuer une détection multi-événements efficace. Ceci nécessite de concevoir des algorithmes efficaces, et les implémentations de log$\mathcal{W}$eaver et de CRS sont notoirement optimisées de ce point de vue. log$\mathcal{W}$eaver utilise en particulier un algorithme subtil (décrit en [RGL01a], section 4.3) qui est étudié pour que l'algorithme d'ordonnancement opère en temps constant. CRS va plus loin, dans la mesure où d'une part les chroniques CRS sont compilées en C++, alors que les signatures log$\mathcal{W}$eaver sont interprétées, et où d'autre part les structures de données stockant les valeurs des variables instanciées par les signatures lors de la détection d'attaques par chroniques sont conçues pour que l'espace total de stockage consommé soit d'environ 20 à 30 octets par chronique. log$\mathcal{W}$eaver n'est pas autant optimisé en espace. Alors que l'efficacité en temps est clairement primordiale, l'efficacité en espace est elle aussi cruciale, dans la mesure où toute utilisation excessive d'espace mémoire pourra saturer l'ordinateur hôte, et de ce fait permettre des attaques de déni de service passant par l'IDS lui-même (cf. section 6.4).

Le souci d'efficacité dans les IDS demande aussi d'introduire des constructions de langage qui permettent à la personne responsable de l'écriture des signatures de limiter l'explosion combinatoire, probablement inévitable, liée à certaines signatures. Appelons ces moyens des moyens d'*élagage* des requêtes. Ces moyens d'élagage se découpent en deux catégories :

– les *élagages verts*, qui éliminent des démarrages d'attaque sans compromettre la capacité de l'outil de détection d'intrusion à fournir un rapport si une attaque a réussi. Autrement dit, même si toutes les attaques ne sont pas annoncées, au moins une sera annoncée parmi une famille d'attaques similaires, et aucune ne sera simplement oubliée par l'outil de détection d'intrusions.
  On trouve ici la notion de signatures `synchronized` de log$\mathcal{W}$eaver et la notion de classes d'équivalences d'attaques de Sutekh, déjà décrites en section 5.4. Nous décrivons les élagages verts en section 6.2.
– Les *élagages rouges*, qui peuvent provoquer la perte d'intrusions détectées. Comme dans le cas de la coupure de Prolog, ce genre d'élagage est donc dangereux, mais est souvent nécessaire pour éviter les attaques par suffocation, dont nous parlerons en section 6.4. Les deux élagages rouges standard sont :
  – les *timeouts*, cf. section 6.3.1 ;
  – la *coupure* proprement dite (coupure " !" en CRS, constructions `anchored` et choix commis en log$\mathcal{W}$eaver), cf. section 6.3.2.

La distinction élagage vert/rouge est reprise de la distinction entre cuts verts et rouges de Prolog [CM81].

6.2 Élagages verts

6.2.1 Familles d'attaques.

Une première famille d'élagages, qui sont des élagages verts, est présente en Sutekh sous forme de la notion de familles d'attaques et en log$\mathcal{W}$eaver sous forme du mot-clé `synchronized`. Nous avons déjà décrit ces mécanismes en section 5.4. Ces mécanismes syntaxiques n'ont pas que pour effet de limiter le nombre d'annonces d'attaques redondantes, mais permettent aussi d'améliorer l'efficacité du moteur de reconnaissance d'attaques.

C'est en particulier le cas en Sutekh, où le mécanisme des familles d'attaques permet directement, via l'utilisation dans le moteur de reconnaissance d'un opérateur $Restrict()$ ([PD02], section 5.2). L'utilisation de `synchronized` en log$\mathcal{W}$eaver est moins directement responsable d'une amélioration éventuelle de l'efficacité : comme il est décrit dans [GL01], section 5.3, l'implémentation des règles `synchronized` opère en fusionnant les pid ("process identifier") des threads responsables de la reconnaissance des attaques obéissant à la même signature et dont les valeurs des variables à synchroniser sont les mêmes. Aucune optimisation n'en résulte ; ceci est dû au fait que log$\mathcal{W}$eaver ne prête pas attention au fait que les pid fusionnés sont des pid de deux threads de la *même* signature.

6.2.2 Optimisations

log$\mathcal{W}$eaver fait aussi appel à une seconde famille d'élagages verts, qui ne sont pas visibles pour l'utilisateur. Il s'agit de techniques d'optimisation de programmes, dont certaines sont décrites succinctement en [RGL01a], section 4.4.

Considérons l'exemple `bad_authentication` de la section 5.4, dans laquelle le second événement est contraint de sorte que `(date2 <= date1+3600)`. L'intention ici est d'écrire un *timeout* (dont on reparlera en section 6.3.1). Or ceci n'est pas vu comme une façon de faire échouer la reconnaissance de la signature : dans la plupart des outils, la vérification d'une contrainte C se fait en testant C sur l'événement courant ; si C est fausse, on la retestera sur l'événement suivant, puis sur le suivant, etc. Or ici la contrainte `(date2 <= date1+3600)` a ceci de particulier que si elle est fausse sur l'événement courant ("il est trop tard"), alors elle restera fausse sur tous les événements ultérieurs. Si on y regarde bien, c'est parce que la valeur de `date1` est fixée (par la première ligne de la signature) et que la valeur de `date2` ne peut que croître. Le préprocesseur qui traduit les logs en le format universel d'entrée attendu par log𝓦eaver informe log𝓦eaver du fait que le champ `.date` ne peut que croître, et log𝓦eaver utilise cette information pour déduire, par des techniques d'analyse statique de code [CC92], que si la contrainte `(date2 <= date1+3600)` devient fausse, elle le restera. log𝓦eaver peut alors laisser tomber la surveillance de l'instance courante de la signature `bad_authentication`, sans danger de perdre la détection d'une attaque réelle : il a prouvé qu'il n'y en aurait pas. (Il s'agit donc bien d'un élagage vert.)

Un autre élagage vert plus subtil est dû à la sémantique des *shortest runs* de log𝓦eaver (cf. section 5.4.1), qui permet d'éliminer des démarrages de reconnaissance d'attaques lorsqu'il est garanti que si une attaque est reconnue qui prolonge le démarrage donné, alors cette attaque ne sera pas la plus courte (au sens de "shortest" décrit en [RGL01a], section 4.2). La garantie en question est découverte par analyse statique des formules log𝓦eaver, cf. [RGL01a], section 4.4.

Il est à noter que si ces optimisations et ces techniques d'analyse statique permettent d'améliorer l'efficacité du moteur de log𝓦eaver, elles sont à leur tour possibles parce que log𝓦eaver a accès à tout ce qui compose les signatures à analyser. L'analyse statique est ici d'autant plus facile et rapide à effectuer que le langage de log𝓦eaver est *déclaratif* et en particulier ne contient aucun effet de bord destructif sur les variables, et d'autre part qu'aucun appel à des fonctions de bibliothèques externes, dont la sémantique n'est pas toujours connue, n'est effectué. La possibilité d'effectuer des optimisations comme dans log𝓦eaver est donc une conséquence directe de ces choix de conception du langage de signatures, et est donc en grande partie antagoniste à un choix d'extensibilité et d'ouverture de l'outil : consulter la section 4.8.

6.3 Élagages rouges

6.3.1 Timeouts

L'idée des timeouts est de laisser tomber la surveillance d'un démarrage de séquence d'attaque lorsque ce démarrage s'étend sur une période jugée trop longue. Le délai au-delà duquel la période est jugée trop longue est laissée à l'appréciation de l'officier de sécurité. Il s'agit donc d'une solution imparfaite, mais pragmatique.

Les chroniques de CRS réalisent ceci via une notion intrinsèque de temps correspondant au nombre d'événements externes reçus par l'outil de détection d'intrusions. Le temps est donc ici juste un numéro d'ordre. Ceci évite de nombreux problèmes liés au fait que les dates incluses dans les événements ne sont pas nécessairement croissantes, ou qu'il n'y a pas nécessairement un unique champ date dans les événements.

log𝓦eaver utilise une approche plus générale des timeouts, dans laquelle un certain nombre de champs dits *monotones* sont identifiés dans les événements — par exemple le numéro d'ordre, qui est par convention le champ `.line` — et qui sont supposés ne jamais décroître. L'utilisation de contraintes comparant la valeur courante d'un champ monotone à une valeur limite simule l'utilisation de timeouts, consulter l'exemple `bad_authentication` et la discussion sur son timeout en section 6.2.2. L'avantage de cette solution est que des contraintes relativement complexes de timeouts peuvent être écrites, incluant des timeouts calculés en fonction de valeurs de champs, ou utilisant des valeurs de différentes champs de dates (date d'enregistrement, d'envoi, de réception, etc., d'événements) ; log𝓦eaver sera capable d'en déduire s'il est possible d'utiliser ces contraintes comme de véritables timeouts, autrement dit (section 6.2.2) comme des contraintes qui, si elles deviennent fausses, le restent pour l'éternité.

6.3.2 La coupure

Le prototype de ce genre de constructions est la *coupure* " !" du langage Prolog, qui permet d'éliminer tous les points de réduire le non-déterminisme en effaçant des points de backtracking. Nous avons déjà discuté de la coupure, et des différentes formes qu'elle prenait en CRS et en log𝓦eaver en section 5.4.2.

Alors que notre souci en section 5.4.2 était de diminuer le nombre d'annonces indésirables sur une même signature, l'autre intérêt évident des coupures est d'améliorer l'efficacité, tant en espace qu'en temps, des moteurs de détection d'intrusions. Il est clair en effet que la suppression de points de choix est une mesure efficace dans ce but. La pratique confirme que ces opérateurs, coupure `!` de CRS, mécanismes `anchored` et choix commis `||` de log$\mathcal{W}$eaver, sont indispensables en pratique (on consultera par exemple la section 8.6 de la FAQ de log$\mathcal{W}$eaver [GL01]).

Il n’en reste pas moins qu’il y a un certain danger à utiliser les coupures (d’où leur classification en élagages rouges), dans la mesure où il n’est que trop facile d’en abuser et, de là, d’écrire des signatures qui ignoreront des attaques que l’on cherchait à détecter. Le manuel de log$\mathcal{W}$eaver va, de fait, jusqu’à inciter les utilisateurs de l’outil à ne pas utiliser la construction `anchored` : le problème est que l’utilisation d’élagages rouges peut *masquer* des attaques. Or c’était un intérêt majeur de l’utilisation de langages déclaratifs (par opposition à ASAX par exemple [Mou97], voir section 4) de permettre d’écrire des signatures qui par construction ne permettrait pas le masquage involontaire d’attaques. Dans la section suivante, nous verrons qu’en fait les élagages rouges sont nécessaires : on peut masquer des attaques en faisant *suffoquer* l’IDS.

6.4 Attaques par suffocation

L’utilisation de log$\mathcal{W}$eaver, Sutekh ou CRS s’effectue soit en-ligne soit hors-ligne. Le mode en-ligne est bien entendu plus contraignant, car le système de détection d’intrusions doit réagir à une attaque le plus vite possible après l’avoir détectée. Or les algorithmes sous-jacents à tous les outils de détection d’intrusion présentés ici nécessitent un temps et un espace exponentiels en le nombre d’événements analysés dans le pire des cas.

Cette complexité est très rarement atteinte en pratique. Cependant, quelques signatures mènent à des comportements quadratiques en temps et en espace. Un exemple typique est celui où l’on demande de détecter une répétition de trois événements A, puis B, puis C, et le log est de la forme

$$ABABABABAB\ldots$$

possiblement avec d’autres événements entre les A et les B affichés, à l’exclusion de C. Les outils log$\mathcal{W}$eaver, Sutekh et CRS vont effectivement attendre un C qui complète la première suite AB. Après la lecture du deuxième B, ces outils seront en attente d’un C qui complète la première ou la deuxième suite AB. Après $2n$ événements, ces outils seront en train d’attendre sur n démarrages possibles d’une suite ABC. À la lecture des événements $2n + 1$ et $2n + 2$, il faudra un temps et un espace proportionnels à n pour traiter les suites d’événements en attente. L’analyse d’un tel fichier de n événements nécessite donc un temps et un espace quadratiques en n pour être traités. En pratique, au bout de quelques dizaines de milliers d’événements, les outils ci-dessus sont tellement ralentis qu’ils sont incapables de réagir en temps réel.

Les raisons pour lesquelles ce phénomène est important dans le contexte présent sont les suivantes, outre la dégradation de la réponse temps-réel, qui est comparativement d’importance négligeable :

– Il est facile pour un attaquant de submerger l’outil de détection d’intrusions de tels faux démarrages de séquences. C’est du coup un moyen commode de ralentir, voire de planter l’outil de détection d’intrusions en vue de l’empêcher de détecter une véritable attaque. Ce genre de technique est d’un usage tout à fait plausible à l’avenir, dans la mesure où les signatures des outils seront en général publiques ou facilement inférables à partir d’attaques tests. Cette technique est appelée “choking attack” dans [PD02] — en français, *attaque par suffocation*.
– La possibilité de former des attaques par suffocation est en partie liée à la conception du langage de signatures, et pas seulement à l’algorithmique utilisée dans l’outil. En fait, ce genre d’attaques est déjà intrinsèque à tout outil qui a l’ambition de détecter des corrélations entre événements, et qui ne se limite pas à du filtrage et/ou du comptage, comme dans les outils de détection d’intrusions classiques.

Les moyens les plus efficaces pour contrer les attaques par suffocation sont les élagages rouges, à commencer par l’usage de timeouts. Malheureusement il est facile pour un intrus de détecter quelles sont les valeurs des timeouts codés dans les signatures de l’IDS, et donc d’adapter ses attaques en espaçant les commandes suffisamment pour sortir des fenêtres de temps pendant lesquelles l’IDS surveillera ses actions. La coupure est aussi un moyen très efficace, quoique délicat d’utilisation. Les élagages verts sont des moyens sûrs d’amélioration de l’efficacité, au sens où il n’y a aucun moyen pour un intrus d’échapper à la détection à cause uniquement d’élagages verts.

Le problème général de l'évitement des attaques par suffocation est cependant encore largement ouvert. Quelques tentatives ont été effectuées dans le domaine des requêtes de bases de données continues [ABB+01] pour garantir des bornes strictes sur la consommation mémoire, et donc aussi le temps d'exécution de mécanismes de requêtes de bases de données sur des flux en temps réel. Traduits en langage de détection d'intrusion, ces résultats impliquent l'existence de langages de signatures totalement immunes aux attaques par suffocation. Malheureusement, les requêtes de [ABB+01] sont très loin d'être suffisamment expressives dans le cadre de la détection d'intrusions (cf. section 4).

7 Conclusion

Il est aujourd'hui nécessaire de pouvoir détecter en temps réel des attaques complexes, consistant en une séquence d'événements et non plus en des événements isolés. Les langages de signatures proposés par les participants du SP3, et qui servent de substrats aux outils log$\mathcal{W}$eaver (LSV), Sutekh (IRISA), ADeLe (Supélec) et CRS (ONERA), ont tous ce même but.

Une surprise lors de l'étude de ces langages dans le cadre du sous-projet SP3 est que, s'ils utilisent un vocabulaire souvent très différent, c'est souvent pour recouvrir des réalités parfois plus proches qu'il n'y paraît. Le simple processus d'arriver à faire se comprendre les différents participants du sous-projet SP3, en particulier, a demandé beaucoup de temps et d'efforts. Ce document est la synthèse de nombreuses discussions tendant à expliquer et à comparer les approches des différents participants.

Il est clair désormais qu'il existe un noyau de constructions fondamentales — celles autour desquelles Sutekh est architecturé en particulier —, dont tout outil de détection d'attaques complexes doit disposer :

1. les filtres,
2. le **et**,
3. le **ou**,
4. la séquence,
5. le **sans**,
6. les variables de corrélations.

Tous les langages des participants proposent ces constructions, sous une forme ou une autre, avec de légères différences de sémantiques parfois.

À ces constructions de base se rajoutent un certain nombre d'autres notions qui varient davantage d'un langage à l'autre : peut-on partager les événements ? peut-on écrire des signatures modulaires ? peut-on compter et comment ? le langage est-il (peut-il, doit-il être) extensible ? quel rôle les variables jouent-elles ? (la distinction entre variables de corrélation et variables d'annonce nous est apparue comme particulièrement importante ici ;) le langage permet-il de faire varier le moment ou la façon dont sont annoncées les attaques ? permet-il de spécifier quelles attaques parmi des familles d'attaques vues comme équivalentes seront annoncées ? quelles constructions les langages de signatures proposés fournissent-ils pour permettre au concepteur de signatures d'améliorer l'efficacité des outils ? Chaque langage apporte ses réponses dans la grande majorité des cas. Nous avons vu d'autre part que peu de ces choix de conception étaient incompatibles.

Il est maintenant envisageable, en particulier, de concevoir un moteur de détection d'intrusions unifié qui intégrerait de façon rationnelle les constructions proposées sous des formes différentes par les différents langages des participants.

Références

[ABB+01] Arvind Arasu, Brian Babcock, Sivnath Babu, Jon McAlister, and Jennifer Widom. Characterizing memory requirements for queries over continuous data streams. Technical Report 2001-49, Stanford University, 2001.

[Car01] Patrice Carle. *Détection d'intrusion par chroniques*. DTIM/MCT, 2001. Rapport Mirador, disponible auprès de l'auteur.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 1992. Disponible en `http :// www.di.ens.fr/~cousot/COUSOTpapers.shtml`.

[CDE+96] Mark Crosbie, Bryn Dole, Todd Ellis, Ivan Krsul, and Eugene Spafford. IDIOT – users guide. Technical report, Purdue University, COAST Laboratory, 1996.

[CER96] Cert(r) advisory CA-1996-16 - Vulnerability in solaris admintool, 1996. `http://www.cert.org/advisories/CA-1996-16.html`.

[CM81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[CO00] Frédéric Cuppens and Rodolphe Ortalo. LAMBDA : A language to model a database for detection of attacks. In *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection (RAID 2000)*. Springer Verlag LNCS 1907, 2000.

[GL01] Jean Goubault-Larrecq. *An Introduction to* logWeaver *(v2.8)*. GIE Dyade & LSV, September 2001. Disponible sur la page Web du projet DICO au LSV, `http://www.lsv.ens-cachan.fr/~goubault/DICO.html`.

[GS96] S. Garfinkel and Gene Spafford. *Practical Unix and Internet Security*. O'Reilly, Sebastopol, CA, USA, 2nd edition, 1996.

[HLCMM92] N. Habra, Baudoin Le Charlier, I. Mathieu, and Abdelaziz Mounji. ASAX : software architecture and rule-based language for universal audit trail analysis. In Y. Deswarte, G. Eizenberg, and J.J. Quisquater, editors, *European Symposium on Research in Computer Security, Toulouse, France*, pages 435–450. Volume 648 of Lecture Notes in Computer Science. Springer, Berlin, 1992.

[LP99] Ulf Lindqvist and Phillip A. Porras. Detecting computer and network misuse through the production-based expert system toolset (P-BEST). In *IEEE Symposium on Security and Privacy*, pages 146–161, 1999.

[MM01] Cédric Michel and Ludovic Mé. ADeLe : An attack description langauge for knowledge-based intrusion detection. In *Proceedings of the 16th International Conference on Information Security*. Kluwer, June 2001.

[Mou97] Abdelaziz Mounji. *Languages And Tools for Rule-Based Distributed Intrusion Detection*. PhD thesis, FUNDP, Namur, Belgium, 1997.

[PD01] Jean-Philippe Pouzol and Mireille Ducassé. From declarative signatures to misuse IDS. In W. Lee, L. Mé, and A. Wespi, editors, *4th International Conference on Recent Advances in Intrusion Detection (RAID'01), Davis, USA.*, pages 1–21. Volume 2212 of Lecture Notes in Computer Science. Springer, Berlin, 2001.

[PD02] Jean-Philippe Pouzol and Mireille Ducassé. Formal specification of intrusion signatures and detection rules. In *Proc. 15th Computer Security Foundations Workshop*, 2002.

[Por92] Phillip A. Porras. STAT – a state transition analysis tool for intrusion detection. Master's thesis, University of California, Santa Barbara, 1992.

[Pou00] Jean-Philippe Pouzol. Détection d'intrusions dans les systèmes informatiques. Master's thesis, INSA, Rennes, France, June 2000. Available via `http://www.irisa.fr/lande/pouzol/`.

[RGL01a] Muriel Roger and Jean Goubault-Larrecq. Log auditing through model-checking. In *Proc. 14th IEEE Computer Security Foundations Workshop*, 2001. See also [RGL01b].

[RGL01b] Muriel Roger and Jean Goubault-Larrecq. Log auditing through model-checking. Technical Report LSV-01-3, Laboratoire Spécification and Vérification, ENS de Cachan, France, April 2001.

[Roe99] Martin Roesch. Snort : Lightweight intrusion detection for networks. In *13th Systems Administration Conference (LISA'99)*, pages 229–238. USENIX Associations, 1999.

[Sun00] Sun MicroSystems. SunSHIELD basic security module guide, February 2000. Part Number 806-1789-10.

[TCP] tcpdump / libpcap. `http://www.tcpdump.org/`.

[Wol83] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2) :72–99, 1983.

A Un échantillon de quelques attaques

A.1 Vol d'un shell utilisateur

Le but de cette attaque est d'usurper l'identité d'un utilisateur sans connaissance de son mot de passe sous Unix. Les détails de l'attaque peuvent être trouvés dans [Pou00, page 41] (voir aussi [GS96, page 88]).

Avec l'identité de l'utilisateur cible, on exécute les commandes suivantes (dans le bon ordre) pour un usage ultérieur :

1. `cp /bin/sh <nom-de-fichier>`;
2. `chmod 4755 <nom-de-fichier>`.

L'exécution de `<nom-de-fichier>` permet à l'attaquant de prendre l'identité de l'utilisateur cible.

Signature dans le langage de scénario de [RGL01a].

$$\{\text{commande} = \text{"/bin/cp"},\ \text{arg1} = \text{"/bin/sh"},\ \text{arg2} = \text{X},\ \text{userID} = \text{Y}\}\wedge$$

$$\Diamond\{\text{commande} = \text{"/bin/chmod"},\ \text{arg1} = \text{"4755"},\ \text{arg2} = \text{X}, \text{userID} = \text{Y}\}$$

Par exemple, X est ici une variable rigide instanciée par le nom du fichier qui sera exécuté avec l'identité usurpée. Comme il est mentionné dans [RGL01a], le langage de cette logique classique du temps linéaire, est peu adapté à la problématique de la détection d'intrusions. Dans le langage de log$\mathcal{W}$eaver, qui réalise (avec une syntaxe pseudo-C) la deuxième logique de [RGL01a], cette attaque sera détectée par :

```
shell_stealing {
  .commande=="^/bin/cp$", .arg1=="^/bin/.*sh$", .arg2 X, .userID Y;
  .commande=="^/bin/chmod$", .arg1="^4755$", .arg2 X, .userID Y;
}
```

en supposant les champs des événements nommés encore une fois `commande`, `arg1`, `arg2` et `userID`. On notera l'utilisation d'expressions régulières, permettant par exemple de reconnaître non seulement `/bin/sh` en `arg1`, mais encore tout programme dans `/bin` se terminant par `sh`. (Le caractère `^` dénote le début du champ, `$` dénote la fin du champ ; par défaut .⟨champ⟩==⟨reg-exp⟩ cherche une sous-chaîne du champ qui soit une instance de l'expression régulière donnée.)

Signature dans le langage de scénario de [PD02].

Règles de la signature :
– $S \to \mathbf{Seq}(\text{copie}[X,Y], \text{change}[X,Y])$;
– $\text{copie}[Z,Z'] \to \mathbf{Filter}(\text{fcopie}[Z,Z'])$;
– $\text{change}[Z,Z'] \to \mathbf{Filter}(\text{fchange}[Z,Z'])$.

Filtres de la signature :
– $\text{fcopie}[Z,Z'] = \{\text{commande} = \text{"/bin/cp"},\ \text{arg1} = \text{"/bin/sh"},\ \text{arg2} = \text{Z},\ \text{userID} = \text{Z}'\}$;
– $\text{fchange}[Z,Z'] = \{\text{commande} = \text{"/bin/chmod"},\ \text{arg1} = \text{"4755"},\ \text{arg2} = \text{Z},\ \text{userID} = \text{Z}'\}$;

Les valeurs "/bin/cp", "4755", etc. peuvent être déclarées comme constantes ce que nous avons omis de faire ici.

A.2 Obtention d'un shell super-utilisateur

Le but de cette attaque est d'obtenir un shell avec les privilèges super-utilisateur. Les détails de l'attaque peuvent être trouvés dans [Pou00, Section 1.3.2]. Cette attaque exploite un bogue (corrigé depuis) du démon de courrier sendmail. Une remise à une valeur par défault des privilèges des fichiers /usr/spool/mail/<nom-utilisateur> est omise par le démon sendmail.

Les commandes suivantes sont exécutées en supposant que le super-utilisateur n'a pas de courrier dans sa boîte aux lettres :

1. cp /bin/sh /usr/spool/mail/root;
2. chmod 4755 /usr/spool/mail/root;
3. touch x;
4. mail root < x.

Les étapes 2. et 3. peuvent être interverties. De plus, le seul but de l'étape 3 est de fabriquer un fichier bidon de sorte à envoyer un mail à root en étape 4. Il est donc raisonnable de ne considérer que les étapes 1, 2 et 4 dans l'attaque.

Signature dans le langage de scénario de [PD02].

Règles de la signature :
- $S \to \mathbf{Seq}(\text{copie}[X, Y], \text{change}[X, Y], \text{mail}[X, Y, F])$;
- $\text{copie}[Z, Z'] \to \mathbf{Filter}(\text{fcopie}[Z, Z'])$;
- $\text{change}[X, Y] \to \mathbf{Filter}(\text{fchange}[X, Y])$;
- $\text{mail}[X, Y, F] \to \mathbf{Filter}(\text{fmail}[X, Y, F])$.

Filtres de la signature :
- $\text{fcopie}[Z, Z'] = \{\text{commande} = \text{"/bin/cp"},\ \text{arg1} = \text{"/bin/sh"},\ \text{arg2} = \text{"/usr/spool/mail/"} \cdot \text{Z},\ \text{userID} = \text{Z}'\}$;
- $\text{fchange}[Z, Z'] = \{\text{commande} = \text{"/bin/chmod"},\ \text{arg1} = \text{"4755"},\ \text{arg2} = \text{"/usr/spool/mail/"} \cdot \text{Z},\ \text{userID} = \text{Z}'\}$;
- $\text{fmail}[Z, Z', F] = \{\text{commande} = \text{"/bin/mail"},\ \text{arg1} = \text{Z},\ \text{input} = \text{F},\ \text{userID} = \text{Z}'\}$;

Signature log$\mathcal{W}$eaver. Une première signature possible, correspondant à celle décrite ci-dessus, est :

```
sendmail_attack synchronized(user) {
  .commande "^(.*/)cp$", .arg1 "sh$",
   .arg2 file "^/usr/spool/mail/root$",
   .userID user;
  .commande "^(.*/)chmod$", .arg1 "^4755$", .arg2 file, .userID user;
  .commande "mail", .arg1 file, .arg2 "^root$", .userID user;
}
```

Les seules différences sont qu'on estime que le programme de copie utilisé est cp, ou /bin/cp, ou n'importe quel programme dont le nom est cp et possiblement précédé d'un chemin d'accès (expression régulière ^(.*/)cp$), et similairement pour chmod ; d'autre part que le programme copié est n'importe quel fichier dont le nom se termine par sh (expression régulière sh$), et que le programme d'envoi de mail est n'importe lequel dont le nom contient mail. (Ceci détecte davantage de cas d'attaques, et est bien entendu aussi exprimable en Sutekh.)

On peut aussi demander à récupérer dans une variable flexible $line la liste des numéros de lignes des événements constituant l'attaque. On peut aussi vouloir détecter une attaque similaire, non seulement sur root, mais sur n'importe quel utilisateur ; on récupère ci-dessous le nom de la victime dans une variable victim, obtenue comme le contenu de la chaîne qui correspond à la première (d'où le \\1) sous-expression régulière de "^/usr/spool/mail/(.*)$" qui est entre parenthèses.

```
sendmail_attack synchronized(user) {
  .line $line,
   .commande "^(.*/)cp$", .arg1 "sh$",
```

```
    .arg2 file "^/usr/spool/mail/(.*)$" { victim = "\\1" },
    .userID user;
  .line $line,
    .commande "^(.*/)chmod$", .arg1 "^4755$", .arg2 file, .userID user;
  .line $line, .commande "mail", .arg1 file, .arg2 victim;
}
```

Les signatures ci-dessus ont le défaut d'annoncer une fausse alerte lorsque les opérations `cp`, `chmod 4755` et `mail` ont été effectuées, mais que le bit setuid a été remis à 0 entre le `chmod` et le `mail`, en effectuant un `chmod` avec un mode (champ `.arg1`) commençant par un autre chiffre que 4, 5, 6 ou 7. Dans ce cas, il n'y a pas d'attaque... sauf si le bit setuid est remis, et ainsi de suite.

Ceci se détecte en logWeaver en effectuant une boucle `while` ; après la reconnaissance d'un `chmod 4755`, un choix non-déterministe est effectué entre tenter de détecter un `chmod` qui remette le bit setuid à 0 et boucler (la contrainte `($$mode !~ "^[^4-7]")` impose que la variable flexible `$$mode` ne doit *pas* être une instance de l'expression régulière `^[^4-7]`, qui détecte tous les modes commençant par un chiffre entre 4 et 7 ; le double dollar en tête du nom de la variable `$$mode` est une convention logWeaver qui impose que `$$mode` n'est pas une variable d'annonce, cf. section 5.1.1).

```
sendmail_attack synchronized(user) {
  .line $line,
    .commande "^(.*/)cp$", .arg1 "sh$",
    .arg2 file "^/usr/spool/mail/(.*)$" { victim = "\\1" },
    .userID user;
 while (1) {
  .line $line,
    .commande "^(.*/)chmod$", .arg1 "^4755$", .arg2 file, .userID user;
    {
      .line $line, .commande "^(.*/)chmod$", .arg1 $$mode, .arg2 file,
        .userID user | ($$mode !~ "^[^4-7]");
    | .line $line, .commande "mail", .arg1 file, .arg2 victim; return;
    }
  }
}
```

A.3 Renommage illégal de fichier

Le but de cette attaque est de renommer un fichier sur lequel on ne possède pas de droit en écriture. Elle tire profit d'un bogue du logiciel `ff.core` et elle peut être utilisée pour obtenir un shell avec les privilèges super-user. Les détails de l'attaque peuvent être trouvés dans [Pou00, page 41] (voir aussi [PD02, Exemple 3] et le site `www.rootshell.com`). On suppose que l'attaquant dispose des droits en écriture dans le répertoire `/vol/rmt/` et que l'on dispose d'une version 2.5.1 ou 2.6 de OpenWindows.

Le scénario de l'attaque est le suivant :

1. `ln -fs /path/file /vol/rmt/diskette0;`
2. `ff.core -r /vol/rmt/diskette0/file newfile /floppy/.`

Ceci se code en logWeaver par :

```
.ACTION=="link", .DST=="^/vol/rmt/diskette0/$";
.ACTION=="ff.core", .SRC=="^/vol/rmt/diskette0/$";
```

en ADeLe par :

```
<DETECT>
   <EVENTS>
      EO : System.Classification[0].name == "create link"
      E1 : System.Classificatiori[0].name == "exec"
   <EVENTS>
   <CONTEXT>
      EO.link_name == "/vol/rmt/diskette0"
      E1.prog.name == "/usr/openwin/bin/ff.core"
   </CONTEXT>
   <ENCHAIN>
      EO ; El
   </ENCHAIN>
</DETECT>
```

Voir aussi la section 4.3.2 qui traite cet exemple pour illustrer l'opérateur d'ordre total entre plusieurs événements.

A.4 L'attaque rpcinfo

Voici l'exemple typique d'une attaque qu'il est possible de monter contre une machine qui offre des services de montage de disques à distance via le protocole NFS. L'exemple est tiré de [MM01, section 5.1] (voir aussi une variante dans la section 2.1).

L'attaque être décrite par la séquence suivante de commandes :

1. `rpcinfo -p <machine-cible>;`
2. `showmount -e <machine-cible>;`
3. `showmount -a <machine-cible>;`
4. `finger @<machine-cible>;`
5. `adduser --uid <user-id> <username>` ; création d'un nouveau compte utilisateur sur la machine de l'attaquant avec les paramètres obtenus lors des étapes précédentes ;
6. `mount -t <partition-cible> /mnt;`
7. `rlogin <machine-cible> -l <username>.`

A.5 Tentative d'intrusion

Le but de cette attaque est simplement de se connecter à une machine. Les détails de l'attaque peuvent être trouvés dans [HLCMM92, Exemple A]. La détection est effectuée lorsque plusieurs tentatives infructueuses de connection sont observées pendant une période de durée fixée.

Une variante de cette attaque consiste à détecter un nombre de tentatives de connexion infructueuses sur une même machine, par exemple 100 tentatives (voir par exemple la section 4.6 pour une attaque de ce type). log*W*eaver permet de détecter un nombre arbitraire de tentatives de connexion avec :

```
spoofing_attack synchronized (dest) anchored {
  while (loop: true) {
    .ACTION=="reject", .SRC $source, .DST dest, .line $line;
    || <<EOF>>; return | (_$loop>=100);
  }
}
```

Voir la section 4.6 pour plus de détail sur la sémantique du langage.

A.6 Bogue dans `admintool` sur Solaris

`admintool` est une interface graphique utilisateur qui permet à l'administrateur du système d'effectuer diverses tâches d'administration. Afin d'eviter de modifier des fichiers système de façon simultanée, `admintool` utilise des fichiers temporaires avec le mécanisme de verrouillage. La gestion de ces fichiers temporaires n'est pas effectuée de façon sûre ce qui peut entraîner des situations où `admintool` crée ou écrit sur des fichiers système arbitraires en présence de dépassement mémoire. Les détails de l'attaque peuvent être trouvés dans [PD01, Section 2.3] (voir aussi [CER96]).

La description de l'attaque est la suivante. `admintool` permet d'installer de nouveaux packages à l'aide de la commande `pkgadd` et tous les packages Solaris contiennent les fichiers `pkginfo` et `pkgmap`.

La première étape de l'attaque consiste à créer ces deux fichiers dans un répertoire temporaire, disons `/tmp/EXP`, et ensuite d'y mettre une chaîne très longue. La création de ces deux fichiers peut se faire dans un ordre quelconque. En log$\mathcal{W}$eaver, on écrira ainsi cette création (voir aussi la section 4.3.3)

```
  .eventID=="^AUE_CREAT$", .path=="/pkginfo$";
+ .eventID=="^AUE_CREAT$", .path=="/pkgmap$";
```

La seconde étape de l'attaque consiste à lancer `admintool` et d'installer le faux package situé dans `/tmp/EXP`. Il est difficile de détecter cette séquence sans annoncer un grand nombre de fausses alertes.

En Sutekh, on écrira :

```
[ eventID = "AUE_CREAT", path #= "*/pkginfo"]
and
[ eventID = "AUE_CREAT", path #= "*/pkgmap"]
```

En ADeLe, on écrira :

```
<DETECT>
  <EVENTS>
    E1: System.Classification[0].name == "file create"
    E2: System.Classification[0].name == "file create"
  </EVENTS>
  <CONTEXT>
    E1.file_name == "pkginfo"
    E2.file_name == "pkgmap"
  </CONTEXT>
  <ENCHAIN>
    Non_ordered{ E1 E2 }
  </ENCHAIN>
</DETECT>
```

A.7 ARP

Il s'agit d'une attaque qui consiste pour l'attaquant à répondre à une requête avant le démon ARP et de fournir des réponses erronées lors d'une demande de transformation IP → Mac par le protocole ARP. Les détails de l'attaque peuvent être trouvés dans [Car01, Section 5.2]. L'attaque peut être détectée quand deux réponses (différentes) à une même requête sont fournies dans un laps de temps court.

Par exemple en log$\mathcal{W}$eaver :

```
  .machine mach, .message "^ACK", .request req, .date date1;
  .machine mach, .program "^ACK", .request req, .date date2
      | (date2 <= date1+30);
```

qui détecte deux messages `ACK` (acknowledge) pour la même requête `req` séparés de moins de 30 secondes.

A.8 Tentative de récupération de `/etc/passwd`

Cette attaque consiste à récupérer le fichier `/etc/passwd` pour un usage ultérieur. Les détails de l'attaque peuvent être trouvés dans [Car01, Section 5.3.1]. Une version simplifiée consiste à détecter les commandes de la forme `cp /etc/passwd <nom-fichier>`. Ceci s'effectue à l'aide d'un simple filtre.