Un langage pour la configuration de DISCUS, une architecture distribuée de solutions de sécurité

Damien Riquet, Gilles Grimaud, Michaël Hauspie

ComPAS'2014 : Parallélisme/ Architecture / Système
Neuchâtel, Suisse, du 23 au 25 avril 2014

Un langage pour la configuration de DISCUS, une architecture distribuée de solutions de sécurité

Damien Riquet, Gilles Grimaud, Michaël Hauspie

Université Lille 1,
Laboratoire LIFL - Bâtiment Ircica - 50, Avenue Halley
59650 Villeneuve d'Ascq - France
prénom.nom@lifl.fr

Résumé
De nos jours, le cloud computing est très populaire dans le monde industriel et académique. La plupart des challenges de sécurité amenés par cette nouvelle architecture ne sont pas encore résolus. Les solutions actuelles, à savoir les IDS ou les pare-feux, n'ont pas été initialement conçues pour détecter des attaques qui tirent profit de la structure du cloud telles que les attaques distribuées. Dans cet article, nous proposons une nouvelle architecture, basée sur un réseau massivement distribué de solutions minimalistes de sécurité, pour adresser ces problèmes. Notre solution, DISCUS, est basée sur une architecture distribuée utilisant des sondes physiques et virtuelles, ainsi que les solutions conventionnelles (IDS et firewalls). Un problème d'une structure aussi large est la gestion de l'ensemble des composants. Cet article présente DISCUS SCRIPT, un langage dédié, qui permet de piloter facilement chacun des composants de cette structure.

Mots-clés : Sécurité, Cloud Computing, Langage, Détection d'Intrusion.

---

1. Introduction

Le cloud computing est un modèle populaire qui fournit d'importantes ressources. Une définition communément admise proposée par le NIST [17] décrit le cloud computing comme « un modèle proposant un accès distant pratique, à la demande et permanent à un ensemble de ressources partagées ». Avec la popularité croissante du cloud, sa sécurité est d'autant plus préoccupante et peut même retarder son adoption massive pour certaines entreprises exigeantes. Dès 2010, une étude [27] rapportait que 62 % des petites et moyennes entreprises ne désirent pas passer au cloud computing pour des raisons de sécurité. Des chercheurs [2], des industriels [25] ou des organisations gouvernementales [12] partagent la même inquiétude. C'est pourquoi les fournisseurs de cloud computing ont besoin de proposer une solution clé en main qui gère efficacement ces problématiques, à savoir la détection et la prévention d'intrusions.
Bien que la désignation « cloud computing » soit assez récente, la sécurité au sein de réseaux complexes a déjà été largement étudiée. Dans [7], Chen et al. listent les nouveaux verrous technologiques du cloud computing. Un de ces nouveaux challenges est la confrontation aux utilisateurs mal intentionnés, qui utilisent les ressources du cloud computing pour réaliser des cyber-attaques [5, 6]. Ils mentionnent également que le partage de ressources peut induire

des attaques par l'utilisation de canaux communs. Pour finir, ils mettent en avant la nécessité pour les fournisseurs de cloud computing d'isoler la structure réseau à plusieurs niveau de granularité (machine physique ou virtuelle, réseaux locaux et data-centres).
L'isolation des éléments du cloud permet d'éviter certaines attaques qui tirent profit de sa structure, telles que les attaques internes, qui sont menées de l'intérieur du cloud et ciblent des machines internes (dans le même data-centre, la même baie voire la même machine physique).

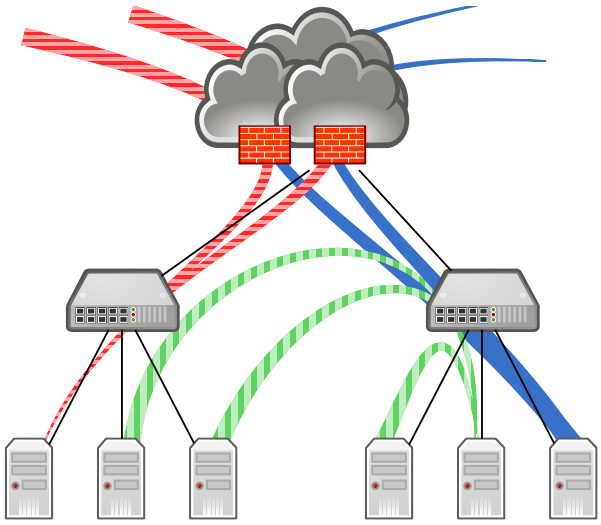

FIGURE 1 – Les différents types d'attaques du cloud computing

La Figure 1 représente une structure typique du cloud avec plusieurs points d'entrée, plusieurs niveaux d'éléments réseaux et des machines physiques. Cette structure basique peut être attaquée de l'extérieur du réseau (chemins striés horizontalement). Elle peut également être utilisée pour mener une attaque de l'intérieur vers l'extérieur du réseau (chemins pleins). Notre travail s'intéresse à la détection de ce type d'attaques, ainsi qu'aux attaques internes (chemins striés verticalement) qui sont menées depuis l'intérieur du réseau et ciblent des machines de ce réseau. À notre connaissance, aucune solution pour ces dernières attaques n'a été proposée dans la littérature.
Les solutions de sécurité actuelles utilisées dans le cloud sont les Systèmes de Détection d'Intrusions (IDS), Systèmes de Prévention d'Intrusions (IPS) et les pare-feux. Bien que les IDS et les IPS tendent à s'uniformiser, initialement le rôle d'un IPS est de prévenir une attaque (et donc l'arrêter) alors que le rôle d'un IDS est seulement de détecter une attaque en cours. Par exemple, le cloud Amazon Web Services (AWS) [1] utilise l'isolation d'instances et des pare-feux. Ce type d'installation n'est pas conçu pour détecter les attaques internes à cause de la localisation des solutions de sécurité, placées en bordure du réseau. De ce fait, ces solutions ne sont même pas conscientes du trafic interne ou de l'état global de la sécurité du réseau, car ce trafic ne les traverse pas.
Dans [21], nous montrons que les attaques distribuées peuvent facilement et efficacement passer outre ces solutions de sécurité, spécialement dans une structure avec plusieurs points d'entrée, comme le cloud computing. Le schéma de sécurisation de data-centres conventionnel ne suffit pas dans ce contexte, il s'agit plutôt d'une mutualisation des ressources dont il faut garantir l'isolation quelque soit le comportement des instances voisines. En effet, un des nouveaux

verrous soulevé par le cloud est de pouvoir protéger autant le réseau que les utilisateurs (qui peuvent changer durant une attaque). De plus, les utilisateurs peuvent être malicieux ou mal intentionnés et prendre le contrôle d'une ou plusieurs machines du cloud pour ensuite mener une attaque. C'est pourquoi il faut garantir un niveau élevé de sécurité pour adresser les différentes exigences de sécurité requises par les clients. Il s'agit ici d'assurer l'intégrité et la confidentialité des données ainsi que la disponibilité des services. De même, il faut être capable de détecter et stopper des attaques distribuées comme les dénis de services distribués (DDoS), la propagation de vers ou toutes autres attaques distribuées au sein du cloud, au sein du même cluster ou de la même machine physique.
Notre solution, DISCUS, s'adresse aux infrastructures larges potentiellement multi-sites et sujettes à des attaques (internes ou non). DISCUS est basée sur une structure massivement distribuée de solutions de sécurité utilisant des sondes hétérogènes. Ces sondes (physiques ou virtuelles) sont éparpillées sur le réseau et collaborent pour détecter les intrusions et les attaques. Les sondes physiques sont peu chères et reconfigurables (typiquement des FPGA ou des systèmes embarqués à bas coût), placées derrière une machine à surveiller ou directement intégrées dans un composant réseau (e.g. commutateur, baies, etc.). Les sondes virtuelles sont localisées sur les machines physiques et peuvent être basées sur des solutions de sécurité existantes, telles que Snort [22], Bro [19] ou Suricata[13]. Nos travaux se sont portés vers Snort, pour lequel nous avons développé un *back-end* (qui produit des fichiers exploitables par Snort depuis un fichier source écrit en DISCUS SCRIPT).
La fonctionnalité principale de notre solution est de proposer une couche logicielle commune qui permette de déployer une solution de sécurité distribuée basée sur des éléments hétérogènes. De plus, n'importe quelle solution de sécurité existante peut être intégrée à DISCUS, il suffit de développer le back-end pour cette solution. En plus de notre back-end pour Snort, nous avons l'intention de développer des back-ends pour pare-feux (comme par exemple pf ou iptables) ou pour des systèmes d'exploitation (à travers un module noyau).
Notre contribution est DISCUS SCRIPT, un langage de description de règles, pour lequel nous avons écrit un compilateur et un back-end pour pouvoir l'intégrer à DISCUS. Un des challenges induit par cette architecture est l'administration unifiée de toutes ces sondes. C'est pourquoi DISCUS se base sur un langage de description de règles de sécurité. Ces règles sont écrites une seule fois de manière globale, puis un compilateur se charge de produire pour chaque sonde une image exploitable (binaire, script, fichier de config, etc.).
Notre solution s'intéresse aux nouveaux challenges soulevés par le cloud computing. Dans ce type de structure, le fait que les clients changent constamment doit être pris en compte par la solution de sécurité mise en place. C'est notamment pourquoi nous proposons, au travers de DISCUS, une architecture reconfigurable qui est adaptée à une infrastructure complexe et qui est également capable d'identifier ou d'isoler précisément une machine parmi des milliers.
Ce papier est organisé de la manière suivante. Dans un premier temps, la Section 2 expose la problématique et positionne notre travail par rapport à littérature. Ensuite, la Section 3 décrit l'architecture de DISCUS, ses composants et son schéma de déploiement. La Section 4 présente DISCUS SCRIPT, le langage dédié utilisé pour configurer les éléments de cette structure et la Section 5 illustre un cas concret de règles décrites dans ce langage. Pour finir, la Section 6 conclut cet article.

2. Problématique et travaux connexes

Cet article se base sur trois domaines de recherche : (a) les solutions de sécurité utilisées dans les réseaux, (b) les techniques de distribution de ces systèmes et (c) les langages dédiés pour

exprimer des règles de sécurité.
Rimal et al. présentent le cloud computing dans [20] comme étant la prochaine étape technologique pour les systèmes distribués. Ils ajoutent qu'il y a une inquiétude grandissante concernant sa sécurité. Les utilisateurs stockent des informations sensibles sur ces architectures et, entre de mauvaises mains, cela peut provoquer des situations désastreuses.
Les principaux éléments de la sécurité réseau sont les pare-feux et les systèmes de détection d'intrusions (IDS). Bellovin et al. décrivent les pare-feux dans [3] comme étant des composants placés entre deux réseaux. Ces équipements filtrent le trafic interdit, défini par des règles de sécurité locales. Debar et al. soulignent dans [10] que la tâche principale des IDS est de surveiller l'utilisation de systèmes et de détecter des états non sécurisés. Les IDS détectent les tentatives et les usages malicieux réalisés par des utilisateurs légitimes qui abusent de leurs privilèges. Des IDS populaires, tels que Bro [19] ou Snort [22] sont basés sur des signatures, des règles de sécurité qui décrivent un comportement ou un trafic anormal.

Les motivations premières de la distribution des solutions de sécurité étaient de s'adapter aux réseaux modernes qui sont relativement complexes. Dans [26], Snapp et al. introduisent DIDS, un système distribué de IDS surveillant un réseau hétérogène de machines. À la fin des années 90, Bellovin et al. ont également proposé dans [4] de distribuer les pare-feux, principalement car ceux-ci se reposaient sur une topologie réseau et des points d'entrée connus à l'avance. Bien que les premières propositions étaient centralisées (plusieurs sondes sur le réseau et une seule unité de corrélation), les solutions récentes proposées dans la littérature sont distribuées (chaque composant détecte les attaques et collabore avec la globalité des composants) [29].
Ce que nous proposons au travers de DISCUS est une solution composée d'agents qui analysent le trafic local et qui soient également capables de partager les données locales de manière à ce que chacun de ces agents puissent avoir un aperçu global de l'état du réseau. Notre solution utilise un système pair à pair basé sur les données, où chacun des agents collabore et dissémine les données locales. Des travaux tels que [23, 15, 16, 28] utilisent des tables de hachage distribuées pour répartir les IDS ou détecter des attaques distribuées (comme les dénis de services distribués, propagation de vers, ou scans de ports distribués). Une des problématiques de ces systèmes est leur configuration ; certains travaux proposent d'utiliser un langage pour configurer ces solutions.

Les langages dédiés sont spécialisés pour un domaine d'application spécifique. Par exemple, un langage spécialisé pour la sécurité réseau peut décrire une exploitation de vulnérabilité, comment une attaque est réalisée, quelles sont les conséquences d'une attaque ou encore comment réagir à une telle attaque.
Les langages de description d'exploits [24, 11] sont utilisés pour décrire une attaque et les différentes phases de cette attaque. D'autre langages [18, 9], basés sur la connaissance de pré ou post conditions, réalisent des déductions pour détecter des attaques. Les langages orientés détection d'attaques [19, 22] décrivent la détection d'une attaque plutôt que son déroulement, généralement en couplant la détection avec le nombre d'occurences d'événements suspects. Pour finir, les langages réactifs [24, 8, 14] décrivent comment réagir lorsqu'une attaque est détectée.
Notre solution consiste en une description des règles de sécurité en une suite d'événements, mais également de structures et de fonctions. Les langages suivants utilisent les mêmes concepts. Un des premiers langage décrivant des règles de sécurité réseau est le langage Bro [19], qui se base sur une programmation déclarative des règles, en utilisant des variables, structures, fonctions et déclarations. Quant à STATL [11], il s'agit d'un langage à état, qui décrit une attaque comme une suite de transitions dans un automate à états.

3. DISCUS : Description de l'architecture globale

L'objectif principal de DISCUS est de détecter et stopper les attaques menées vers ou depuis une architecture réseau importante et complexe tel que le cloud computing. De plus, nous souhaitons que cette solution soit facilement (re)configurable du fait que de nouveaux exploits sont découverts chaque jour. Les personnes responsables de la sécurité de ces structures doivent être capables de charger des règles de sécurité facilement. C'est pourquoi nous proposons DISCUS SCRIPT, un langage dédié à la sécurité, qui fera l'interface entre l'administrateur et l'ensemble des composants de DISCUS.
Cette section introduit DISCUS. Dans un premier temps, nous décrivons la solution, puis nous présenterons les éléments de sécurité intégrés à celle-ci. Pour finir, nous expliciterons le schéma de déploiement.

3.1. Vue d'ensemble

Les solutions de sécurité usuelles sont les pare-feux et les systèmes de détection d'intrusions (IDS). Ces solutions considèrent généralement qu'une attaque est menée depuis l'extérieur du réseau vers l'intérieur, ce dernier étant le réseau protégé. Ce n'est plus vrai pour des réseaux de type cloud computing, où les ressources peuvent être louées à mauvais escient. Parce que ces éléments courants ne sont adaptés, nous proposons une solution composée de ces derniers, auxquels nous combinons un réseau massivement distribué de sondes de sécurité.

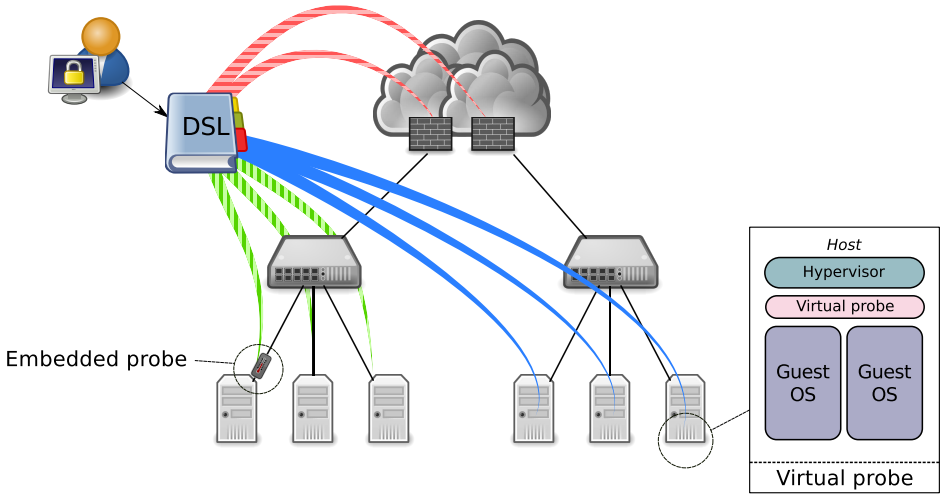

FIGURE 2 – Architecture de DISCUS

La Figure 2 schématise la structure proposée. Dans cette structure simplifiée, les pare-feux demeurent les points d'entrée de l'extérieur du réseau et filtrent le trafic non désiré. Les nouveaux éléments sont les sondes, qui peuvent être physiques ou virtuelles. Nous souhaitons avoir ces sondes au plus près des machines physiques, afin de pouvoir analyser à une granularité fine le trafic. De plus, cela nous permet d'isoler une machine si cela est nécessaire ou d'identifier clairement qui est la cible ou le commanditaire d'une attaque.

Les sondes physiques sont placées en coupure, au plus près des machines physiques, ou directement intégrées aux composants réseaux (carte réseau de la machine ou du commutateur le plus proche). Les sondes virtuelles sont intégrées aux machines physiques, notamment celles qui hébergent des machines virtuelles. Il s'agit typiquement des machines du cloud computing ; dans ce cas, il faut analyser le trafic réseau de chaque machine virtuelle, de manière à empêcher des attaques au sein d'une même machine physique.
Il y a deux types d'utilisateurs dans notre approche. Tout d'abord, il y a les personnes qui écrivent les règles en DISCUS SCRIPT lorsqu'une nouvelle faille ou vulnérabilité a été découverte. Il y a également les administateurs, qui récupèrent ces règles et souhaitent les appliquer à leur réseau. Éventuellement, ils écrivent des règles qui sont spécialisées à leur propre réseau. Pour cela, ils utilisent le compilateur mis à disposition pour générer l'ensemble des binaires, un binaire unique par élément de la structure. Une fois configurée et déployée, les sondes coopèrent pour détecter des intrusions. Les prises de décision complexes peuvent être déléguées à des éléments plus puissants, tels que les pare-feux. En plus de ces deux types d'utilisateurs, il est tout à fait envisageable d'inclure les utilisateurs finaux (avec des restrictions sur les règles) dans l'écriture des règles s'appliquant à leurs machines ou instances.

3.2. Les différents types de sondes

Dans notre solution, nous introduisons le concept de sondes massivement distribuées. Le but de ces sondes est d'être placées en coupure sur le réseau, de manière à pouvoir analyser le trafic réseau et d'être capable de détecter des attaques et également de pouvoir fournir des moyens d'isoler des machines. Parce que l'on souhaite que ce contrôle soit fin, il faut placer ces sondes au plus près des machines à surveiller. C'est pourquoi les sondes physiques sont placées sur les machines elles-mêmes, voire même intégrées sur le matériel réseau. Les sondes virtuelles sont idéales pour les machines qui hébergent plusieurs machines virtuelles. En effet, il faut être capable d'analyser le trafic de ces machines, au niveau des interfaces virtuelles. Ces dernières peuvent être basées sur des solutions existantes, telles que Snort ou Bro, qui reçoivent leurs scripts par le compilateur de DISCUS SCRIPT pour intégrer le réseau de sondes de DISCUS.
Une des caractéristiques de notre solution est de vouloir se reposer sur un réseau massivement distribué. C'est pourquoi nous permettons l'intégration de certaines solutions de sécurité existantes à DISCUS, en écrivant pour chacune un *back-end* qui se charge de traduire le langage DISCUS SCRIPT en fichiers exploitables par ces solutions (binaires, fichiers de configs, etc).

3.3. Configuration de la structure

La configuration des éléments de sécurité de notre solution est réalisée en trois étapes. La Figure 3 schématise les étapes de ce déploiement. Tout d'abord, (1) l'administrateur écrit un ensemble de règles en utilisant le langage dédié. Ensuite, (2) il compile ces règles en utilisant le compilateur de DISCUS SCRIPT. Pour finir, (3) le compilateur génère un programme binaire pour chacune des cibles de la structure. Ces binaires sont spécialisés en fonction de la topologie du réseau et des caractéristiques de chacun de ces éléments. Ils sont également produits en sélectionnant un sous ensemble des règles cohérentes avec l'élément de sécurité pris en compte. Par exemple, une règle qui filtre du trafic dans un sous réseau interne n'a pas lieu d'être déployée sur l'ensemble des sondes, seulement sur celles localisées dans ce sous réseau.

4. DISCUS SCRIPT : le langage dédié à la configuration

Dans cette section, nous introduisons le langage dédié qui va de paire avec l'architecture proposée DISCUS. Dans notre cas, nous voulons configurer facilement des sondes (physiques

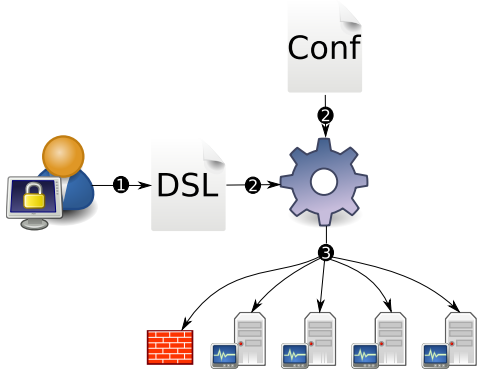

FIGURE 3 – Schéma de déploiement

ou non) et d'autres matériels de sécurité existants de notre structure. Pour cela, nous avons choisi d'élaborer un langage déclaratif typé statiquement. Nous avons choisi de proposer un nouveau langage car ceux proposés dans la litterature ne sont pas assez ou peu expressifs sur les cas d'utilisations que l'on souhaite adresser. Par exemple, il n'est pas possible ou difficile de décrire des attaques complexes (composées de plusieurs paquets) en utilisant le langage de script de Snort, qui s'intéresse généralement à un seul paquet réseau. Nous avons opté pour un langage événementiel, c'est à dire qu'une règle de sécurité exprimée dans ce langage constitue un événement réseau. Nous avons fait ce choix car il est naturel de décrire des séquences de paquets réseaux comme une séquence d'événements. Certains éléments de la syntaxe ne seront pas présentés dans ce papier (comme les structures ou les énumérations) parce que la syntaxe est très similaire au C. De même, nous n'énumérerons pas les types (dû à la longueur de l'article) ; les seuls types utilisés dans cet article sont les énumérations et les entiers *intX*, où *X* représente la taille en bits de l'entier. Plus d'informations concernant la syntaxe et les types sont présents sur la documentation en ligne [1].
Nous nous intéresserons au cours des prochaines sous sections aux éléments importants de ce langage, à savoir les tables (l'élément central qui permet le partage des données) et les règles de sécurité événementielles.

4.1. Déclaration des tables

Une solution de sécurité conserve des données contextuelles pour pouvoir analyser le système. Généralement, cela inclut les machines sur le réseau, les états de connexion, le nombre de connexions, etc. Les tables sont des types qui agrègent ces informations contextuelles. Par exemple, stocker le nombre de connexions TCP échouées peut permettre de détecter des scans de ports. L'information présente dans ces tables est partagée à l'ensemble des solutions de sécurité de notre solution. Nous expliciterons comment ces tables sont partagées dans la suite de cette section.

Listing 1 – Déclaration d'une table

```
table attack_attempts {
    int32 attacker;
    int attempt_counter;
    time last_attempt;
};
```

1. Voir le site de l'auteur : http://www.lifl.fr/~riquetd/

Dans le Listing 1 est déclarée une table, constituée du nombre de tentatives présumées d'attaques pour une adresse IPv4. Cet exemple de table peut être utilisé pour détecter des attaques ; lorsque ce nombre de tentatives dépasse un seuil fixé, le système de sécurité réagit ou lève une alerte.
Un des objectifs de DISCUS est de fournir un système distribué capable de détecter des attaques distribuées. Ce type de détection est possible au travers de nos tables qui sont partagées. Les tables sont réparties sur l'ensemble du réseau et chacune des solutions de sécurité peut accéder à ces entrées ou les modifier. L'utilisation de systèmes pair à pair est idéal pour ce cas d'utilisation, car ces systèmes gèrent l'arrivée et le départ de nouveaux éléments du système courant ainsi que la tolérance aux fautes et le passage à l'échelle. Nous nous tournerons dans nos prochains travaux vers ce type de systèmes.

4.2. Purge et suppression d'entrées dans les tables

Notre solution se repose sur une multitude de sondes de sécurité, qui ont chacune des limites en mémoire et donc en quantité d'information stockable. C'est pourquoi il est nécessaire d'effacer des entrées de nos tables. Nous avons identifié deux cas de figure : supprimer des entrées lorsque le système est proche de la saturation mémoire (on parlera ici de purge) ou bien lorsque les entrées sont obsolètes (on parle ici de suppression).

Listing 2 – Exemple de suppression d'entrées dans une table

```
remove attack_attempts
    when (now - last_transmission) > 3600 ;
```

Dans le Listing 2, nous spécifions que nous souhaitons supprimer les entrées qui ne sont pas récentes (dans ce cas, toutes les entrées réalisées il y a plus d'une heure sont supprimées). La syntaxe de l'instruction de purge est similaire, c'est pourquoi nous n'en discuterons pas ici. Le but de cette dernière est de sélectionner les entrées que l'on peut supprimer lorsque le système en fait la demande.

4.3. Règles de sécurité événementielles

Les principales déclarations de ce langage dédié sont les règles de sécurité. Nous avons choisi d'adopter une approche événementielle car les attaques réseau sont facilement assimilables à une séquence d'événements étalés sur le temps. De plus, les systèmes événementiels peuvent être facilement et efficacement implémentés sur des technologies comme le FPGA, une puce reprogrammable et rapide (circuit dédiée et parallélisable). Le Listing 3 représente la syntaxe utilisée pour définir un nouvel événement. Cette règle est déclenchée lorsqu'un événement (**on** `A`) se produit et que les conditions sont réunies. Un des événements élémentaires du langage est `packet`, qui se déclenche lorsqu'un paquet réseau doit être traité. Lorsqu'une règle est déclenchée, elle peut exécuter plusieurs instructions différentes : insérer ou modifier une entrée dans une table, déclencher un ou plusieurs événements ou exécuter une commande.

Listing 3 – Syntaxe d'une règle événementielle

```
on A(args)
    where  ... /* Conditions      */
    insert ... /* Tables */
    update ... /* Tables fields  */
    run ... /* Script or action from the library */
    raise D [in ...] ;
```

Les conditions sont optionnelles et il faut qu'il y ait au moins une action associée à chaque règle. Des exemples de ces règles de sécurité seront présents dans la Section 5.

4.4. Caractéristique du langage dédié

Dans cette sous-section, nous mettons en avant quelques caractéristiques de DISCUS SCRIPT. Tout d'abord, nous avons choisi d'avoir un compilateur minimaliste, c'est à dire que nous souhaitons que le moins de choses soient fixées dans le code du compilateur. C'est pourquoi la majorité des éléments est décrit en utilisant le langage (par exemple, la structure d'un paquet IP ou TCP est décrite dans le langage). Également, le compilateur est capable de détecter les règles incohérentes ou en conflit ; dans ce cas, il alerte l'utilisateur de manière à lui indiquer d'où provient l'erreur.
D'autre part, DISCUS SCRIPT a aussi quelques propriétés. Il s'agit d'un langage typé statiquement, car nous souhaitons éviter les erreurs de programmation ou de sémantique à l'exécution. De même, le compilateur détecte les cycles, de cette manière nous pouvons assurer que la phase de détection se terminera, car on arrivera ultimement à un événement terminal (qui ne déclenche pas d'autres événements). Pour finir, le compilateur détecte et supprime les règles incohérentes ou inutilisées : par exemple des événements orphelins ou des séquences d'événements qui mènent à un événement terminal qui ne fait rien. Le binaire produit sera alors correct par construction grâce à ces propriétés.

5. Exemples de règles de sécurité

Dans cette section, nous décrivons plusieurs règles de sécurité implémentées avec notre langage. Au travers de ces exemples, nous cherchons à détecter une attaque SYN flood, attaque relativement simple mais efficace si rien n'est mis en place pour la stopper. Une connexion TCP s'établit en trois temps ; cette attaque se satisfait des deux premières phases. Le SYN flood consiste à établir un grand nombre de connexions TCP sans procéder à la phase finale de l'établissement de la connexion. Cela provoque des ralentissements réseaux et de la saturation de la mémoire. Une méthode basique de détection de cette attaque consiste à surveiller le nombre de connexions semi-ouvertes entre deux machines. Lorsque ce nombre dépasse un seuil fixé, la solution de sécurité en déduit qu'une attaque est en cours et peut alors réagir : alerter l'administrateur, fermer toutes les connexions, mettre sur une liste noire l'initiateur des connexions, etc.
Nous considérons pour la suite de l'exemple que la table `attack_attempts` du Listing 1 est définie. De même, nous considérons qu'un événement `tcp_packet` est déclenché à chaque paquet TCP.
Tout d'abord, il faut créer une table qui stocke les états des connexions TCP. Cet état est défini dans une énumération appelée `tcp_state`.

```
enum tcp_state {...};
table tcp_table {
    int32 src, dst;
    int16 p_src, p_dst;
    enum tcp_state state;
    time last_trans;
};
```

Quand une nouvelle connexion TCP est initiée, on crée une entrée dans cette table. Seuls les arguments utiles seront indiqués dans l'ensemble de cet exemple.

```
on tcp_packet(..., int32 src, int32 dst, int16 p_src, int16 p_dst, int9 flags, ...)
    where flags == SYN
    insert into tcp_table {
        src = src;
        dst = dst;
```

```
        p_src = p_src;
        p_dst = p_dst;
        tcp_state = TCP_HANDSHAKE_SYN;
        last_transmission = now;
    };
```

Lorsqu'une entrée de cette table est créée et que l'hôte distant continue l'établissement de la connexion (SYN-ACK), la détection de l'attaque débute. Comme cette attaque consiste à ne pas finaliser une connexion TCP, il suffit de verifier l'état de la connexion après un court instant (ici, nous considérons 250 ms), afin de vérifier si oui ou non son état a évolué. Si l'établissement de la connexion est toujours à l'état intermédiaire, il s'agit probablement d'une tentative d'attaque.

```
on tcp_packet(...)
    where flags == SYN | ACK
    and exists t in tcp_table
        with t.src == src,
             t.dst == dst,
             t.p_src == p_src,
             t.p_dst == p_dst,
             t.state == TCP_HANDSHAKE_SYN
    raise syn_flood_attempt(src, dst, p_src, p_dst)
        in 250 ms;
```

Si la connexion TCP n'est toujours pas établie après ce délai, nous créons une entrée dans la table `attack_attempt`, qui stocke le nombre de tentatives d'attaques. Le code suivant crée une nouvelle entrée dans cette table.

```
on syn_flood_attempt(int32 src, int32 dst, int16 p_src, int16 p_dst)
    where exists t in tcp_table
        with t.src == src,
             t.dst == dst,
             t.p_src == p_src,
             t.p_dst == p_dst,
             t.state == TCP_HANDSHAKE_SYN
    and not exists a in attack_attempt
        with a.attacker == dst
    insert into attack_attempt {
        attacker = dst;
        attempt_counter = 1;
    };
```

Dans le cas où une entrée existe déjà pour une machine suspecte, il suffit de mettre à jour le nombre de tentatives d'attaques.

```
on syn_flood_attempt(int32 src, int32 dst, int16 p_src, int16 p_dst)
    where exists t in tcp_table
        with t.src == src,
             t.dst == dst,
             t.p_src == p_src,
             t.p_dst == p_dst,
             t.state == TCP_HANDSHAKE_SYN
    and exists a in attack_attempt
        with a.attacker == dst
    update a.attempt_counter += 1
    update a.last_attempt = now
```

```
    raise syn_flood_check(src)
```

Lorsque ce nombre atteint un seuil fixé (matérialisé ici par `THRESHOLD`), la solution de sécurité a détecté l'attaque et peut choisir de réagir. Dans notre cas, nous mettons l'hôte en question en quarantaine pour une heure.

```
on syn_flood_check(int32 src)
    where exists a in attack_attempts
        with a.src == src,
             a.attempt_counter >= THRESHOLD
    run blacklist(src, 3600);
```

6. Conclusion

Le cloud computing est une architecture puissante. Elle peut fournir plusieurs couches de services suivant les besoins des utilisateurs. Actuellement, seules des inquiétudes quant à sa sécurité retardent son adoption massive.
Dans cet article, nous proposons DISCUS, une nouvelle architecture qui peut résoudre des problèmes amenés par le cloud computing. Notre proposition repose sur une structure massivement distribuée, basée sur des solutions conventionnelles mais également des sondes disséminées sur le réseau. Ces sondes collaborent afin de détecter les intrusions sur le cloud computing. Une des difficultés de cette architecture est la configuration de ces nombreuses sondes. Nous avons opté pour un langage dédié, DISCUS SCRIPT, qui décrit les règles de sécurité de manière événementielle.
Actuellement, le compilateur est implémenté, ainsi que le back-end Snort. Nos travaux actuels se tournent vers la distribution et la communication des sondes, ainsi que l'expérimentation de ces sondes dans un environnement réel.

Bibliographie

1. Amazon. – *Amazon Web Services : Overview of Security Processes*. – Rapport technique, 2011.
2. Armbrust (M.), Fox (A.), Griffith (R.), Joseph (A. D.), Katz (R. H.), Konwinski (A.), Lee (G.), Patterson (D. A.), Rabkin (A.) et Zaharia (M.). – *Above the Clouds : A Berkeley View of Cloud Computing*. – Rapport technique, 2009.
3. Bellovin (S.) et Cheswick (W.). – Network firewalls. *Communications Magazine, IEEE*, vol. 32, n9, sept. 1994, pp. 50 –57.
4. Bellovin (S. M.). – Distributed firewalls. *Journal of Login*, vol. 24, n5, 1999, pp. 37–39.
5. Bitweasil. – Cryptohaze cloud cracking. *Defcon 20*, 2012.
6. Bryan (D.) et Anderson (M.). – Cloud computing : A weapon of mass destruction ? *Defcon 18*, 2010.
7. Chen (Y.), Paxson (V.) et Katz (R. H.). – *What's New About Cloud Computing Security ?* – Rapport technique nUCB/EECS-2010-5, EECS Department, University of California, Berkeley, Jan 2010.
8. Cuppens (F.), Gombault (S.) et Sans (T.). – Selecting appropriate counter-measures in an intrusion detection framework. – In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*. IEEE, 2004.
9. Cuppens (F.) et Ortalo (R.). – Lambda : A language to model a database for detection of attacks. – In *Recent advances in intrusion detection*. Springer, 2000.

10. Debar (H.), Dacier (M.) et Wespi (A.). – Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 1999.
11. Eckmann (S.), Vigna (G.) et Kemmerer (R.). – Statl : An attack language for state-based intrusion detection. *Journal of Computer Security*, vol. 10, n1/2, 2002, pp. 71–104.
12. European Network and Information Security Agency. – *Cloud Computing Risk Assessment*. – Rapport technique, 2009.
13. http ://suricata ids.org/. – Open source ids suricata, 2014.
14. Kanoun (W.), Dubus (S.), Papillon (S.), Cuppens-Boulahia (N.) et Cuppens (F.). – Towards dynamic risk management : Success likelihood of ongoing attacks. *Bell Labs Technical Journal*, 2012.
15. Li (Z.), Chen (Y.) et Beach (A.). – Towards scalable and robust distributed intrusion alert fusion with good load balancing. – In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense, LSAD '06*, LSAD '06, pp. 115–122. ACM, 2006.
16. Marchetti (M.), Messori (M.) et Colajanni (M.). – Peer-to-peer architecture for collaborative intrusion and malware detection on a large scale. *In : Information Security*, pp. 475–490. – Springer Berlin Heidelberg, 2009.
17. Mell (P.) et Grance (T.). – *The NIST Definition of Cloud Computing*. – Rapport technique, juillet 2009.
18. Michel (C.) et Mé (L.). – Adele : an attack description language for knowledge-based intrusion detection. – In *Proceedings of the 16th International Conference on Information Security (IFIP/SEC 2001)*, 2001.
19. Paxson (V.). – Bro : a system for detecting network intruders in real-time. *Computer Networks*, 1999.
20. Rimal (B.), Choi (E.) et Lumb (I.). – A taxonomy and survey of cloud computing systems. – In *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on*, pp. 44 –51, aug. 2009.
21. Riquet (D.), Grimaud (G.) et Hauspie (M.). – Large-scale coordinated attacks : Impact on the cloud security. *The Second International Workshop on Mobile Commerce, Cloud Computing, Network and Communication Security 2012*, juillet 2012, p. 558.
22. Roesch (M.) et al. – Snort-lightweight intrusion detection for networks. – In *Proceedings of the 13th USENIX conference on System administration*, pp. 229–238. Seattle, Washington, 1999.
23. Saad (R.), Nait-Abdesselam (F.) et Serhrouchni (A.). – A collaborative peer-to-peer architecture to defend against ddos attacks. – In *Local Computer Networks, 2008. LCN 2008. 33rd IEEE Conference on*, pp. 427–434, 2008.
24. Salgueiro (P.), Diaz (D.), Brito (I.) et Abreu (S.). – Using constraints for intrusion detection : the NeMODe system. *Practical Aspects of Declarative Languages*, 2011.
25. Shankland (S.). – Hp's hurd dings cloud computing, ibm, 2009.
26. Snapp (S. R.), , Brentano (J.) et Dias (G. V.). – Dids (distributed intrusion detection system) - motivation, architecture, and an early prototype. *in proceedings of the 14th National Computer Security Conference*, 1991, pp. 167–176.
27. Spiceworks. – New study sees rise in cloud services adoption among small and medium businesses in first half of 2010, 2010.
28. Yegneswaran (V.), Barford (P.) et Jha (S.). – Global intrusion detection in the domino overlay system. – In *In Proceedings of Network and Distributed System Security Symposium (NDSS*, 2004.
29. Zhou (C. V.), Leckie (C.) et Karunasekera (S.). – A survey of coordinated attacks and collaborative intrusion detection. *Computers and Security*, vol. 29, n1, 2010, pp. 124 – 140.