Haka : un langage orienté réseaux et sécurité

Kevin Denis, Paul Fariello, Pierre Sylvain Desse et Mehdi Talbi
kdenis@arkoon.net
pfariello@arkoon.net
psdesse@arkoon.net
mtalbi@arkoon.net

Arkoon Network Security

Résumé Nous proposons dans cet article un framework permettant de définir et d'appliquer des politiques de sécurité sur un trafic réseau. L'objectif étant d'offrir un langage à la fois simple, performant et expressif, et permettant de spécifier le protocole, sa machine à état ainsi que les règles de sécurité associées, évitant à l'utilisateur les freins habituels liés à de tels développements : extraction de données protocolaires, réassemblage de flux, gestion de la mémoire.

1 Introduction

Il existe une multitude de langages permettant d'exprimer des règles de sécurité [1,4,5]. Le plus souvent, celles-ci se limitent à évaluer la valeur de certains champs protocolaires par rapport à des expressions régulières. Appliquer certaines transformations au préalable sur les données ou bien imbriquer des règles est souvent fastidieux, voire impossible avec les langages de signatures actuels. De plus, les dissecteurs protocolaires sont souvent codés manuellement en C. Leurs développements sont ardus, coûteux en temps et souvent non exempts d'erreurs. L'objectif de Haka [2] est de définir un langage à la fois simple, expressif et performant, donnant la possibilité d'exprimer des contrôles de sécurité puis de les appliquer ensuite sur un trafic réseau. Il s'agit là de la proposition d'un DSL (Domain Specific Language) permettant de spécifier le protocole, sa machine à état ainsi que les règles de sécurité qui lui sont associée offrant ainsi à l'utilisateur la possibilité d'analyser des protocoles allant du niveau OSI 2 (Ethernet) au pseudo-niveau 8 : applications web au dessus de http (e.g. twitter, facebook).

Haka est basé sur le langage Lua. Ce choix est motivé par une série de critères (expressivité, lisibilité, popularité, etc.) et d'évaluations (mesures empreinte mémoire et vitesse d'exécution). En effet, Lua est un langage relativement simple, compact (environ 200 ko), et qui gagne tous les jours

en notoriété à en juger par les multiples outils de sécurité qui utilisent Lua (Suricata [6], Wireshark). De plus, il dispose d'une version JIT (LuaJIT) nous permettant d'améliorer encore les performances.

L'architecture de Haka a été conçue de manière modulaire. Elle intègre des modules de capture de paquets (Libpcap, NFQueue), des modules de journalisation et d'alertes (Syslog) et des modules de dissection protocolaires. Ces modules sont chargés dynamiquement et sont facilement interchangeables, permettant ainsi à un utilisateur de modifier le module de capture de paquets ou d'ajouter un module de dissection.

L'API Haka permet de s'interfacer avec les différents modules de l'architecture. Haka est un langage orienté réseau et sécurité, l'API a été donc définie de manière à permettre de manipuler aisément le contenu des paquets/flux, et de réagir activement (e.g. rejeter/modifier le paquet/flux) et/ou passivement (lever une alerte) à une potentielle anomalie détectée. La modification à la volée des données est gérée de manière transparente par Haka évitant ainsi à l'utilisateur de gérer l'ajustement des numéros de séquence, la fragmentation éventuelle des paquets ou bien le recalcul des sommes de contrôle.

Le présent article est organisé de la manière suivante : l'architecture de Haka est présentée dans la section 2. La section 3 introduit la manière de définir des règles de sécurité dans Haka. Enfin, la grammaire de dissection sera brièvement introduite en section 4.

2 Architecture de Haka

La figure 1 illustre les différents modules de l'architecture de Haka. Au coeur de celle-ci, nous retrouvons l'API Haka associé au langage Lua et nous permettant d'exprimer les contrôles de sécurité. La politique ainsi exprimée est ensuite évaluée sur le trafic collecté à partir d'une ou plusieurs interfaces réseaux ou bien à partir d'un fichier de capture au format pcap. Les paquets et flux capturés sont disséqués et le flux est réassemblé conformément à la politique de sécurité exprimée. L'architecture Haka comprend également un module de log et un module d'alerte permettant de journaliser les événements liés à l'évaluation des règles de sécurité.

Haka embarque deux modules de capture de paquets : Libpcap et NFQueue. Le premier permet de rejouer des fichiers de capture Pcap ou bien de procéder à la lecture de paquets à partir d'une ou de plusieurs interfaces réseaux. Un outil externe *hakapcap* dédié uniquement à ce mode de capture a été implémenté et peut être utilisé à des fins de forensics. Le deuxième mode de capture permet de rejeter ou bien modifier à la volée le

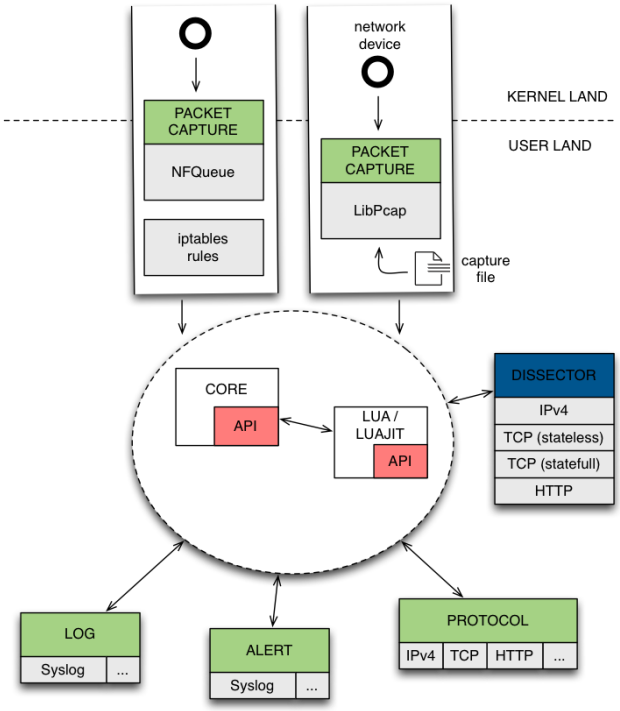

Figure 1. Architecture de Haka

contenu des paquets. Haka n'est lié à aucun de ses modules de capture de paquets et l'utilisateur peut intégrer son propre module tel que PF_RING par exemple.

Lorsqu'un paquet est intercepté par Haka, le paquet transite par une série de dissecteurs. Le rôle de la dissection est d'offrir un accès aux différents champs et états du paquet/flux. Ces derniers sont accessibles par leurs noms en lecture/écriture (e.g. ip.ttl, tcp.dstport, http.version, etc.). La dissection expose également un certain nombre de données (table des connexions, etc.) et fonctions utilitaires (vérification des sommes de contrôles, rejet du paquet, etc.) propres à chaque dissecteur. À l'issue de la phase de dissection, le paquet est reconstruit (et éventuellement fragmenté) avant d'être envoyé au dissecteur suivant jusqu'à être renvoyé sur le réseau.

Haka intègre également des modules de journalisation et d'alertes. Le module de log permet de transcrire au format Syslog des événements. Le module d'alerte permet de remonter des informations, également au format syslog, liées aux intrusions ou anomalies détectées. Le format des alertes est inspirée de IDMEF.

l'API Haka permet d'interagir avec les différents modules de l'architecture. Étant un langage orienté réseaux et sécurité, l'API de Haka a été définie de manière à permettre de manipuler aisément le contenu des

paquets/flux, et de réagir activement (e.g. rejeter/modifier le paquet/flux) et/ou passivement (lever une alerte) à une potentielle anomalie détectée.

3 Règles de sécurité

Une règle de sécurité consiste en un point de branchement (*hook*) et une fonction d'évaluation (*eval*). Un *hook* est un point de branchement où l'utilisateur peut installer des règles de sécurité. Lorsque le système atteint un point de branchement, toutes les règles qui sont enregistrées dessus sont alors évaluées dans l'ordre de leurs déclarations. Dans Haka, des *hooks* "down" et "up" sont prédéfinis pour chaque dissecteur. Ils sont déclenchés respectivement après la dissection et avant la reconstruction des paquets/flux. L'utilisateur a également la possibilité de définir ses propres *hooks*. C'est la cas par exemple des *hooks tcp-connection-new*, *http-request* et *http-response* qui nous permettent de définir respectivement des contrôles de sécurité à chaque fois qu'une nouvelle connexion est établie ou bien qu'une nouvelle requête http est émise ou renvoyée.

Une fonction d'évaluation permet quant à elle d'exprimer la logique du contrôle de sécurité. Elle prend en paramètre les données du dissecteur courant contenant entre autres les différents champs du paquet/flux (accessibles via l'opérateur '.') et sur lesquels on peut invoquer un certain nombre de méthodes utilitaires telles que la vérification du checksum par exemple. L'invocation de méthodes sur les données du protocole se fait via l'opérateur ':'. Il s'agit là des accesseurs standards sur les éléments d'une table Lua. La règle 1 illustre un exemple de règles de sécurité. Elle permet de vérifier si la somme de contrôle d'un paquet ip est correcte. Le paquet est rejeté et une alerte est levée dans le cas contraire.

La règle débute à la ligne 1 par le chargement du dissecteur *ipv4*. La définition de la règle se fait via le mot clé *haka.rule*. La vérification de la somme de contrôle s'opère au niveau de la ligne 6. La suite de la règle illustre deux possibilités de réaction à l'anomalie. D'abord, une réaction passive via l'émission d'une alerte *haka.alert*.

Dans cet exemple, nous nous contentons de renseigner uniquement une description de l'anomalie et sa source (i.e. l'adresse ip accessible via *pkt.src*). Un exemple de réaction active à l'anomalie est donnée par la ligne 11 où le paquet est rejeté via la méthode *drop*.

```
require('protocol/ipv4')

haka.rule{
    hooks = { 'ipv4-up' },
    eval = function (self, pkt)
        if not pkt:verify_checksum() then
            haka.alert{
                description = "Bad IP checksum",
                sources = { haka.alert.address(pkt.src) },
            }
            pkt:drop()
        end
    end
}
```

Listing 1. Filtrage de paquets basique

Haka offre la possibilité de créer de nouveaux paquets ou bien de modifier à la volée le contenu des paquets transitant par lui. Cela se fait par une simple affectation de la nouvelle valeur au champ concerné. À titre d'exemple, l'affectation de la ligne 8 dans la règle 2 a pour effet de modifier la valeur de l'entête "User-Agent" du protocole http [1]. Haka, gère de manière transparente les changements impliqués au niveau des couches tcp/ip telles que l'ajustement des numéros de séquences, la fragmentation éventuelle des paquets ou encore le recalcul des sommes de contrôle.

```
require('protocol/ipv4')
require('protocol/tcp')
require('protocol/http')

haka.rule{
    hooks = { 'http-request' },
    eval = function (self, http)
        http.request.headers["User-Agent"] = "Haka User-Agent"
    end
}
```

Listing 2. Modification des flux de données

4 Grammaire de dissection

L'écriture de dissecteurs est une tâche complexe et fastidieuse nécessitant de gérer les copies mémoires ou des opérations de byte-swapping. Pour cette raison, une nouvelle grammaire permettant de décrire les protocoles ainsi que leurs machines à état est proposée. Cette grammaire permet de

1. Si l'en-tête http est absent, alors il est automatiquement ajouté et affecté avec cette valeur

spécifier à la fois les protocoles de type texte et binaires ainsi que leurs machines à état. Grâce à cette nouvelle grammaire, nous avons pu recoder des versions plus complètes de l'ensemble des dissecteurs de la version 0.1 : ipv4 (avec la gestion des options), tcp (avec une meilleure gestion des états du protocole) et http (avec la gestion des "chunks"). Nous avons également rajouté trois dissecteurs supplemetaires : icmp, udp et dns.

Pour avoir participé au développement des deux versions des dissecteurs, nous pouvons affirmer que l'exercice est nettement simplifié avec la nouvelle grammaire. La règle 3 donne un aperçu de la spécification du protocole icmp. La spécification complète est disponible sur Github [3].

```
icmp.grammar = g.record{
    g.field('type', g.number(8)),
    g.field('code', g.number(8)),
    g.field('checksum', g.number(16))
}
```

Listing 3. Dissecteur ICMP

5 Conclusion

La performance du langage a été un élément clé tout au long du développement de Haka. La version actuelle intègre un certain nombre de fonctionnalités visant à améliorer les performances de Haka telles que le support du "multi-threading" ou bien l'utilisation de la version JIT de Lua (LuaJIT). Le code a été également optimisé dans le but de réduire le plus possible les copies mémoire. Dans une prochaine version, nous comptons également améliorer les performances de Haka. Des évaluations sommaires nous ont déja permis d'identifier des points de ralentissement au niveau des modules de capture de paquets. Ce point peut néanmoins être résolu en embarquant un module de capture plus performant. D'autres pistes sont également envisageables pour améliorer les performances de Haka telles que son intégration dans le noyau.

Références

1. The BRO network security monitor. http ://www.bro.org.
2. Haka. http ://haka-security.org.
3. Haka source code. https ://github.com/haka-security/haka/.
4. ModSecurity. http ://www.modsecurity.org.
5. Snort. http ://www.snort.org.
6. Suricata. http ://suricata-ids.org.