Haka : un langage orienté réseaux et sécurité

Kevin Denis, Pierre Sylvain Desse, Paul Fariello et Mehdi Talbi
kdenis@arkoon.net
psdesse@arkoon.net
pfariello@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 d'implémenter rapidement des politiques de filtrage tout en simplifiant à l'utilisateur les tâches liées à l'extraction de données, au réassemblage des flux ou bien à la gestion de la mémoire.

1 Introduction

Il existe une multitude de langages permettant d'exprimer des règles de sécurité [1,5,9]. 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. Dans les outils d'analyse réseaux, les dissecteurs protocolaires sont souvent codés manuellement en C. Leurs développements sont ardus, coûteux en temps et souvent non exempts d'erreurs.

De nombreux outils d'analyse et de sécurité réseau existent. Certains permettent de faire de la conformité réseau (le protocole respecte t-il sa spécification), de la sécurité (les données sont elles malveillantes), de la journalisation (tracer les actions effectuées) ou de la recherche d'informations dans des flux ou des fichiers de capture. Les outils couramment utilisés sont les firewalls libres ou commerciaux, les sondes IDS/IPS et les outils comme Scapy [8] ou bien wireshark [12] et son analyseur tshark.

L'objectif de Haka [2] est la création d'un Domain Specific Language (DSL) permettant l'expression de règles de sécurité et assurant l'analyse des réseaux du niveau OSI 2 (Ethernet) au pseudo-niveau 8 (applications web au dessus de http). Ce langage doit permettre à des éditeurs de solutions, des opérateurs de services (hébergeurs), des clients finaux

d'exprimer des règles de sécurité sans avoir à réaliser des développements complexes, longs et couteux. Les équipes manipulant cette technologie auront à leur disposition un outil puissant et expressif permettant des traitements fins (détection de malware, recherche de patterns, contrôle de protocole, etc.) et pourront s'abstraire des freins habituels : gestion de la mémoire, temps de développement, langage et principes de bas niveau.

L'architecture de Haka a été conçue de manière modulaire. Elle intègre dans sa version actuelle des modules de capture de paquets (Libpcap [4], NFQueue [6]), 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. Ces modules exposent une API Lua [14], permettant d'utiliser ce langage pour exprimer des fonctions de sécurité. Les raisons du choix de ce langage seront exposées dans l'article.

Le présent article est organisé de la manière suivante : l'architecture de Haka est montrée dans la section 2, puis la section 3 présente plus en détail l'API de Haka à travers un certain nombre de règles de sécurité. Ensuite, la section 4 donne des exemples de réalisations possibles avec Haka. Enfin, la conclusion présente les évolutions futures de Haka et qui portent sur la dissection protocolaire (grammaire et machine à état). Cette grammaire de dissection utilisera le même langage et une syntaxe proche que celle présenté dans cet article.

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 réassemblé conformément à la politique de sécurité exprimée. L'architecture Haka comprend également un module de log et un module d'alerte nous permettant de journaliser les événements liés à l'évaluation des règles de sécurité.

2.1 Modules de capture de packets

Haka embarque deux modules de capture de paquets : Libpcap et NFQueue. Le premier permet de rejouer des fichiers de capture Pcap ou

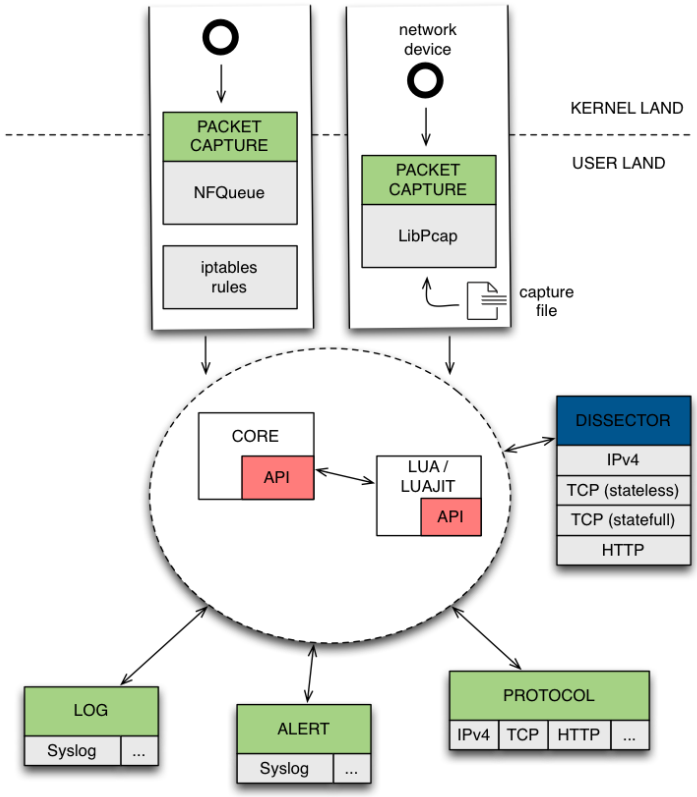

Figure 1. Architecture de Haka

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 contenu des paquets.

Haka n'est lié à aucun de ses modules de capture de paquets et l'utilisateur peut intégrer son propre module tels que PF_RING [7] par exemple.

2.2 Modules de log et d'alertes

Haka intègre 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 [13]. Le format complet est précisé dans la documentation.

De même que pour les captures de paquets, l'API Haka permet à l'utilisateur d'intégrer son propre module de journalisation pour exporter les événements dans une base de données MySQL par exemple.

2.3 Modules de dissection

La dissection des protocoles suit le schéma 2. 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..

La version 0.1 intègre quatre dissecteurs protocolaires : *ipv4*, *tcp*, *tcp-connection* et *http*. Les deux premiers ont été codés en C et les deux derniers en Lua. 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 est en cours de développement, elle sera brièvement introduite en conclusion de cet article.

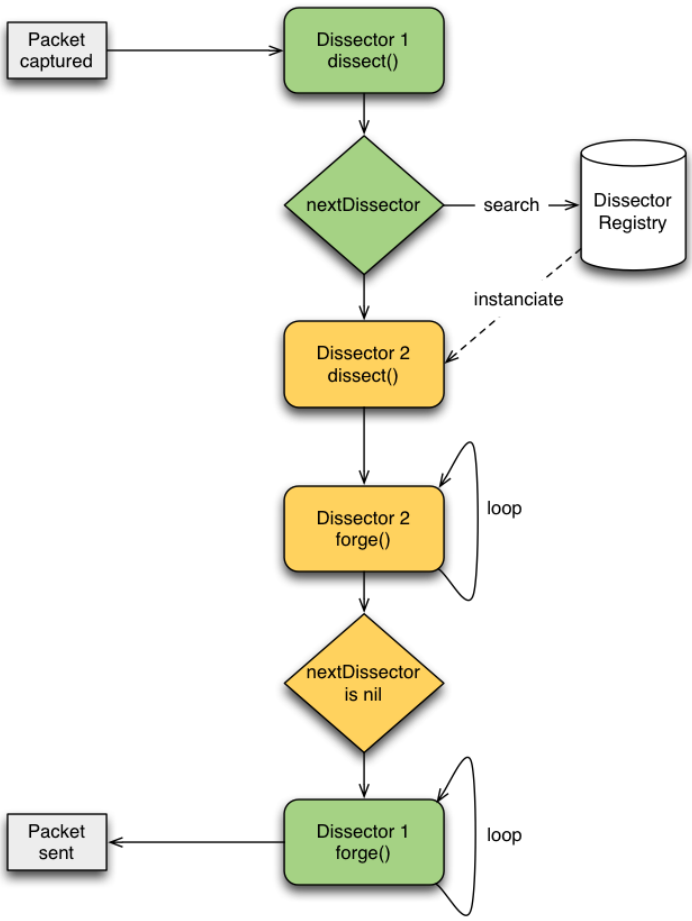

Figure 2. Dissection protocolaire

Les dissecteurs ont pour rôle de placer des *hooks*. 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* sont prédéfinis pour chaque dissecteur. Comme l'illustre la figure 3, des *hooks* "down" et "up" sont déclenchés respectivement après la dissection (*dissect()*) et avant la reconstruction (*forge()*) des paquets/flux.

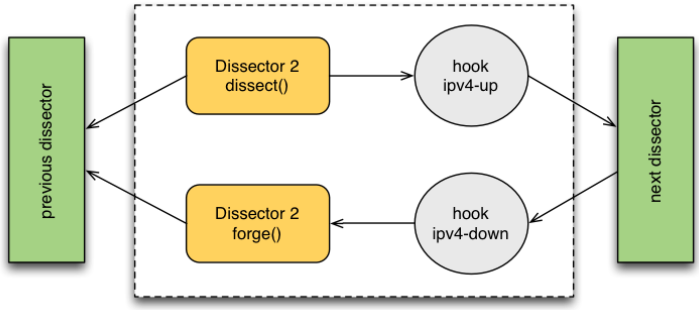

Figure 3. Hooks

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.

2.4 Langage Haka

L'objectif primaire de Haka a été de définir un langage à la fois simple, expressif et performant et permettant de définir rapidement des règles de sécurité. Nous avons choisi de prendre pour base 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) faites sur un ensemble de langages cibles (Python, GO, Ruby, Javascript etc.). À l'issue de cette phase d'évaluation, Lua a été celui qui répondait au mieux aux exigences fixées pour le langage Haka. 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 [10], Wireshark). De plus, il dispose d'une version JIT (LuaJIT [11]) nous permettant d'améliorer encore les performances.

Comme décrit par la figure 1, l'API Haka permet d'interagir avec les différents modules de l'architecture. Haka, étant un langage orienté réseau et sécurité, l'API 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. L'API sera présentée plus en détails à travers les exemples de règles de la section 3.

3 Règles de sécurité

Une règle de sécurité consiste en un point de branchement (*hooks*) et une fonction d'évaluation (*eval*). Une fonction d'évaluation permet 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.

L'API Lua de Haka est présentée via quelques exemples de règles de sécurité.

3.1 Analyse de paquets

La règle 1.1 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*. Le format des alertes inclut un certain nombre de champs inspirés du format IDMEF [13]. 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 1.1. Filtrage de paquets basique

3.2 Modification des flux de données

Haka offre la possibilité de modifier à la volée le contenu de paquets ou de flux de données. 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 1.2 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 1.2. Modification des flux de données

3.3 Création de paquets

Haka permet de lire, modifier, rejeter mais également créer de nouveaux paquets. La règle 1.3 donne exemple de création d'un paquet ip et qui consiste à renseigner les différents champs du protocole.

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

```
local npkt = haka.packet.new()
npkt = ipv4.create(npkt)
npkt.version = 4
npkt.id = 0xbeef
npkt.flags.rb = true
npkt.flags.df = false
npkt.flags.mf = true
npkt.frag_offset = 80
npkt.ttl = 33
npkt.proto = 20
npkt.src = ipv4.addr(192, 168, 0, 1)
npkt.dst = ipv4.addr("192.168.0.2")
npkt:send()
```

Listing 1.3. Injection d'un paquet ip

Cette fonctionnalité a notamment permis de créer une fonction utilitaire de clôture de connexions tcp en envoyant un paquet *RST* ("reset") aux parties communicantes.

3.4 Groupe de règles

Dans Haka, les règles sur un même *hook* sont évaluées séquentiellement jusqu'à ce que l'une d'elle décide de rejeter le paquet ou lorsque toutes les règles ont été évaluées. Un mécanisme de groupe de règles a été implémenté, ce qui permet deux choses : tout d'abord, de définir une politique par défaut. Ensuite, de sortir du groupe de règles sans évaluer les autres règles du groupe. Ceci permet par la même occasion une meilleure structuration du fichier de règles. On pourrait ainsi imaginer créer un groupe par classe d'attaque (SQLi, XSS, etc.).

Un groupe de règles consiste en un nom de groupe (*name*), un ensemble de 3 fonctions (*init*, *fini* et *continue*), et une collection de règles. Les fonctions *init* et *fini* sont exécutées respectivement avant et à la fin de l'évaluation du groupe de règles. La fonction *continue* permet de prendre la décision de poursuivre ou non l'exécution des règles ; décision prise sur la base de la valeur de retour *ret* qui lui est retournée par la dernière règle évaluée.

Le groupe de règles 1.4 représente un exemple de configuration de firewall minimaliste. Le groupe est composé de deux règles dont la spécification est quasi-identique et consistant à n'autoriser que les connexions http et ssh émanant du réseau 192.168.10.0/24. L'évaluation du groupe de règles débute par la fonction *init* où est installée une fonction de journalisation des informations de connexion. Après chaque évaluation de règle, la fonction *continue* est exécutée afin de décider si l'évaluation des autres règles du groupe se poursuit ou pas. Dans le cas où aucune des règles ne

satisfait les conditions requises, la fonction *fini* est appelée afin de décider du sort de la connexion courante. Dans notre exemple, la connexion est refusée.

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

local auth_network = ipv4.network("192.168.10.0/24");

local group = haka.rule_group{
    name = "group",
    init = function (self, pkt)
        haka.log.debug("filter",
            "entering packet filtering rules : %d --> %d",
            pkt.tcp.srcport, pkt.tcp.dstport)
    end,
    fini = function (self, pkt)
        haka.alert{
            description = "Packet dropped : drop by default",
            sources = haka.alert.address(pkt.tcp.ip.src,
                pkt.tcp.srcport),
        }
        pkt:drop()
    end,
    continue = function (self, pkt, ret)
        return not ret
    end
}

group:rule{
    hooks = { 'tcp-connection-new' },
    eval = function (self, pkt)
        local tcp = pkt.tcp
        if auth_network:contains(tcp.ip.src) and
            tcp.dstport == 80 then
            pkt.next_dissector = "http"
            return true
        end
    end
}

group:rule{
    hooks = { 'tcp-connection-new' },
    eval = function (self, pkt)
        local tcp = pkt.tcp
        if auth_network:contains(tcp.ip.src) and
            tcp.dstport == 22 then
            haka.log.warning("filter",
                "no available dissector for ssh")
            return true
        end
    end
}
```

Listing 1.4. Groupe de règles

3.5 Filtrage interactif de paquets & débogage des règles

Le mode de fonctionnement standard de Haka est de filtrer automatiquement les paquets conformément aux règles de sécurité spécifiées. Il est toutefois possible de procéder à ce filtrage de manière interactive. L'idée étant de disposer d'un prompt Haka à chaque fois qu'un nouveau paquet ip, une nouvelle connexion tcp ou bien une nouvelle requête http est reçue et ce dépendamment des "hooks" sélectionnés. En activant ce mode, l'utilisateur disposera alors d'une console Lua lui permettant de profiter de toute l'API Haka afin d'afficher, diagnostiquer, modifier ou bien tout simplement rejeter le paquet/flux courant. Du fait des délais impliqués, ce mode est est plus adapté lorsque l'on procède à la capture des paquets avec Pcap. Le mode interactif se trouve être un excellent moyen pour appréhender l'API Haka.

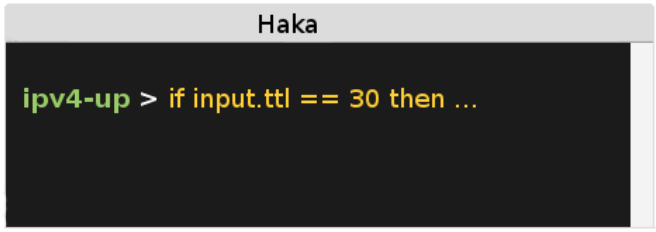

Figure 4. Filtrage interactif de paquet

Haka est doté également d'un débogueur utile pour identifier l'origine d'un éventuel problème survenant lors de l'évaluation des règles de sécurité. Lorsque ce mode est activé, un prompt invite l'utilisateur au lancement de Haka à inspecter le code incriminé. L'utilisateur aura ainsi la possibilité de placer des points d'arrêt ("breakpoints"), de suivre l'exécution pas à pas, d'afficher le contenu des diverses variables ou bien d'obtenir une trace d'exécution ("backtrace"). La table 1 recense l'ensemble des commandes disponibles pour le débogage des règles.

4 Exemples de réalisations avec Haka

Il est important de comprendre que Haka n'est pas un outil, mais un langage simplifiant le développement d'outils. A ce titre, Haka peut être utilisé par plusieurs types d'utilisateurs : chercheurs en sécurité qui

list (l)	lister le code source
print (p)	afficher le contenu des variables
local, upvalue	afficher les variables locales, les “upvalues”
break (b), removebreak (rb)	ajouter, supprimer des points d’arrêt
step (s), next (n), finish (f)	suivre pas à pas l’exécution
continue (c)	poursuivre l’exécution
backtrace (bt)	afficher une trace d’exécution
frame	sélectionner une “frame”
stack	afficher la pile Lua

Table 1. Commandes du débogueur Haka

souhaitent évaluer de nouveaux algorithmes de détection d’intrusions, administrateurs de firewalls qui développent des contre-mesures, ou bien des pentesteurs qui souhaitent analyser et modifier à la volée des flux ou effectuer des analyses après incident pour scruter des fichiers de capture.

Un exemple complet d’utilisation de Haka est fourni dans cet article. Il montre la faculté de Haka à analyser des flux applicatifs et à les modifier à la volée de manière simple. Un second paragraphe présente, sans les définir, une série de réalisations qui peuvent être facilement créées avec Haka.

4.1 Redirection de trafic HTTP

Haka va être utilisé pour répondre à la problématique suivante. Imaginons qu’un administrateur de sécurité souhaite empêcher ses utilisateurs d’accéder à internet avec des navigateurs obsolètes à l’exception faite des sites de mise à jour. La solution à ce problème est simple, il suffit de comparer la version du *User-Agent* des navigateurs avec des versions de référence et de rediriger l’utilisateur vers un site de mise à jour lorsque le navigateur utilisé est détecté comme obsolète.

Il n’existe pas à notre connaissance d’outils simples de mise en oeuvre et répondant à ce problème. Avec Haka, ce problème se résout en centaine de lignes. Nous détaillons ici les grandes lignes de la solution. Le code au complet est disponible en annexe B.

Nous avons crée un groupe deux règles que l’on a nommé *safe_update*. La première règle de ce groupe (règle 1.5) autorise la requête de l’utilisateur si celle-ci est à destination d’un site de mise à jour (e.g. *mozilla.org*). Ce test est effectué à la ligne 6 où l’on vérifie le contenu de l’entête *Host* du protocole http.

```
safe_update:rule{
    hooks = { 'http-response' },
    eval = function (self, http)
        local host = http.request.headers['Host'] or ''
        for _, dom in ipairs(update_domains) do
            if string.find(host, dom) then
                haka.log("Filter", "Requesting an update domain")
                return true
            end
        end
    end
}
```

Listing 1.5. Redirection de trafic HTTP - Première règle du groupe

La seconde règle du groupe permet de modifier à la volée les entêtes de la réponse http lorsque la requête provient d'un navigateur obsolète (vérification du contenu de l'entête *User-Agent*). Le bout de code 1.6 permet de vérifier d'abord s'il s'agit d'une ancienne version du navigateur Firefox.

```
...
local UA = http.request.headers["User-Agent"] or
    "No User-Agent header"
haka.log("Filter", "UA detected: %s", UA)
local FF_UA = (string.find(UA, "Firefox/"))

if FF_UA then -- Firefox was detected
    local version = tonumber(string.sub(UA, FF_UA+8))
    if not version or version < last_firefox_version then
         haka.alert{
             description = "Firefox is outdated, please upgrade",
             severity = 'medium'
         }
     ...
     end
end
...
```

Listing 1.6. Redirection de trafic HTTP - Seconde règle du groupe

Ensuite, les entêtes de la réponse http sont modifiées (voire crées) de telle manière à rediriger la requête vers un site de mise à jour.

```
-- redirect browser to a safe place where updates will be made
http.response.status = "307"
http.response.reason = "Moved Temporarily"
http.response.headers["Content-Length"] = "0"
http.response.headers["Location"] = firefox_web_site
http.response.headers["Server"] = "A patchy server"
http.response.headers["Connection"] = "Close"
http.response.headers["Proxy-Connection"] = "Close"
```

Listing 1.7. Redirection de trafic HTTP - Seconde règle du groupe

4.2 Autres cas d'usage

Nous présentons dans cette section des scénarios d'usage de Haka. Les exemples suivants ne sont pas complets et ne contiennent que la partie des règles les plus pertinentes pour l'exemple considéré.

Il est trivial avec Haka de développer un outil permettant de s'assurer qu'un protocole est conforme à sa spécification ou bien de mettre en place une politique de sécurité plus restrictive que celle admise par la RFC. L'exemple 1.8 présente une règle n'autorisant que les requêtes GET du protocole http.

```
haka.rule{
    hooks = { 'http-request' },
    eval = function (self, http)
        if http.request.method ~= "GET" then
            http:drop()
        end
    end
}
```

Listing 1.8. Conformité protocolaire

L'objectif de Haka a été de définir dans un premier temps un langage facilitant la définition de règles de sécurité. Il est ainsi possible de dériver à partir de Haka un système de détection d'intrusions. Un des tutoriels, disponible avec les sources de Haka [3], a été axé sur la détection des attaques par injection SQL en prenant en compte les diverses techniques d'obfuscation.

Finalement, Haka pourrait servir à des fins de forensics. Un outil dédié à l'analyse de fichiers de capture pcap (*hakapcap*) à d'ailleurs été crée pour répondre à ce type de besoin. L'utilisateur a ainsi la possibilité d'agréger les données de capture et d'effectuer par exemple des statistiques sur les données collectées.

5 Conclusion

La première version de Haka nous a permis de définir un langage dédié à la définition de règles de sécurité évoluées, et de les appliquer sur un trafic réseau. Les travaux en cours de développement sont focalisés sur la dissection protocolaire et ont abouti sur une grammaire nous permettant de 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”). Pour avoir participé au développement des deux versions des dissecteurs, nous pouvons affirmé que l’exercice est nettement simplifié avec la nouvelle grammaire. La règle 1.11 donne un aperçu de la spécification du protocole icmp. La spécification complète est disponible dans l’annexe B.

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

Listing 1.9. Dissecteur ICMP

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 tels 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 tels que PF_RING [7]. D’autres pistes sont également envisageables pour améliorer les performances de Haka tels que son intégration dans le noyau.

6 Remerciements

Le projet HAKA est en partie financé par le Fond national pour la Société Numrique (FSN). Nous tenons à remercier également nos partenaires Télécom ParisTech et Openwide avec lesquels nous collaborons sur ce projet.

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. LibPcap. http ://www.tcpdump.org.
5. ModSecurity. http ://www.modsecurity.org.
6. Netfilter.

7. PF_RING. http ://www.ntop.org/products/pf_ring.
8. Scapy. http ://www.secdev.org/projects/scapy/.
9. Snort. http ://www.snort.org.
10. Suricata. http ://suricata-ids.org.
11. The LuaJIT Project. http ://luajit.org.
12. Wireshark. http ://www.wireshark.org/.
13. Hervé Debar, David A. Curry, and Benjamin S. Feinstein. The intrusion detection message exchange format (idmef). RFC 4765, 2007.
14. Roberto Lerusalimschy, Waldemar Celes, and Luiz Enrique de Figueiredo. Lua programming language. http ://www.lua.org.

A Redirection de trafic HTTP

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

local last_firefox_version = 24.0
local firefox_web_site = 'http://www.mozilla.org'

-- Domain whitelist, all traffic to
-- these domains will be unmodified
local update_domains = {
    'mozilla.org',
    'mozilla.net',
    -- You can extend this list with other domains
}

-- Forward all traffic on port 80 to the HTTP dissector
haka.rule{
    hooks = { 'tcp-connection-new' },
    eval = function (self, pkt)
        if pkt.tcp.dstport == 80 then
            pkt.next_dissector = 'http'
        end
    end
}

-- Rule group implementing a logical 'or'
safe_update = haka.rule_group{
    name = 'safe_update',

    -- Initialization
    init = function (self, http)
        local host = http.request.headers['Host']
        if host then
            haka.log("Filter", "Domain requested: %s", host)
        end
    end,

    -- Continue is called after evaluation of each security
    -- rule the ret parameter decide whether to read next
    -- rule or skip the evaluation of the other rules in the
    -- group
    continue = function (self, http, ret)
        return not ret
    end
}

-- Traffic to all websites in the whitelist is unconditionnally
-- allowed
safe_update:rule{
    hooks = { 'http-response' },
    eval = function (self, http)
        local host = http.request.headers['Host'] or ''
        for _, dom in ipairs(update_domains) do
            if string.find(host, dom) then
```

```
                haka.log("Filter", "Update domain: go for it")
                return true
            end
        end
    end
}

-- If the User-Agent contains firefox and the version is outdated
-- then redirect the traffic to firefox_web_site
safe_update:rule{
    hooks = { 'http-response' },
    eval = function (self, http)
        local UA = http.request.headers["User-Agent"] or
            "No User-Agent header"
        haka.log("Filter", "UA detected: %s", UA)
        local FF_UA = string.find(UA, "Firefox/")

        if FF_UA then -- Firefox was detected
            local version = tonumber(string.sub(UA, FF_UA+8))
            if not version or version < last_firefox_version then
                haka.alert{
                    description = "Firefox is outdated, please
                        upgrade",
                    severity = 'medium'
                }
                -- redirect browser to a safe place where updates
                -- will be made
                http.response.status = "307"
                http.response.reason = "Moved Temporarily"
                http.response.headers["Content-Length"] = "0"
                http.response.headers["Location"] = firefox_web_site
                http.response.headers["Server"] = "A patchy server"
                http.response.headers["Connection"] = "Close"
                http.response.headers["Proxy-Connection"] = "Close"
            end
        else
            haka.log("Filter", "Unknown or missing User-Agent")
        end
    end
}
```

Listing 1.10. Redirection de trafic HTTP

B Dissecteur ICMP

```
local ipv4 = require('protocol/ipv4')

local icmp_dissector = haka.dissector.new{
    type = haka.dissector.EncapsulatedPacketDissector,
    name = 'icmp'
}

icmp_dissector.grammar = haka.grammar.record{
    haka.grammar.field('type',     haka.grammar.number(8)),
    haka.grammar.field('code',     haka.grammar.number(8)),
    haka.grammar.field('checksum', haka.grammar.number(16))
        :validate(function (self)
            self.checksum = 0
            self.checksum = ipv4.inet_checksum(self._payload)
        end),
    haka.grammar.field('payload',  haka.grammar.bytes())
}:compile()

function icmp_dissector.method:parse_payload(pkt, payload, init)
    self.ip = pkt
    icmp_dissector.grammar:parseall(payload, self, init)
end

function icmp_dissector.method:verify_checksum()
    return ipv4.inet_checksum(self._payload) == 0
end

function icmp_dissector.method:forge_payload(pkt, payload)
    if payload.modified then
        self.checksum = nil
    end

    self:validate()
end

function icmp_dissector:create(pkt, init)
    pkt.payload:insert(0, haka.vbuffer(8))
    pkt.proto = 1

    local icmp = icmp_dissector:new(pkt)
    icmp:parse(pkt, init)
    return icmp
end
```

Listing 1.11. Dissecteur ICMP