Un petit galop d'essai avec le système Haka (filtrage, sécurité, paquets, protocoles, etc)

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 20 juillet 2014

https://www.bortzmeyer.org/haka.html

Le 10 juillet dernier, lors des RMLL <https://2014.rmll.info/> à Montpellier, j'ai eu le plaisir de participer à l'« Atelier HAKA : un langage "*open source*" [sic] de sécurité réseau ». Haka <http://haka-security.org/> est un langage permettant d'analyser les paquets, de réagir à certaines caractéristiques des paquets et de générer des réponses diverses. Cela permet entre autres de faire des pare-feux très souples car complètement programmables.

Haka <http://haka-security.org/> est fondé sur Lua et il faut donc un peu réviser ses connaissances Lua avant de pratiquer l'atelier. (J'ai déjà écrit ici <https://www.bortzmeyer.org/lua-sur-machine-gener> html> et là <https://www.bortzmeyer.org/evolution-de-lua.html> sur Lua et réalisé un tout petit programme avec <https://www.bortzmeyer.org/unicode-to-sql.html>.) Haka étend Lua en ajoutant au langage de base des extensions permettant de tripoter les paquets réseau facilement. Je ne vais pas vous faire un cours sur Haka (il y en a un en ligne <http://doc.haka-security.org/haka/release/v0.1/doc/user/userindex.html> et les transparents de l'atelier sont disponibles en ligne <https://hakasecurity.files.wordpress.com/2014/07/haka-rmll-2014-slides.pdf>), juste documenter mon expérience.

L'atelier se faisait dans une machine virtuelle Debian dans VirtualBox. La machine virtuelle utilisée pouvait être téléchargée en <https://hakasecurity.files.wordpress.com/2014/07/haka-live-iso.zip>. (Je n'ai pas encore essayé d'installer Haka moi-même.) Une machine plus récente est, depuis, en <http://www.haka-security.org/resources.html>. Une fois la machine démarrée, il faut faire un `setxkbmap fr` dans un terminal pour passer en AZERTY, le clavier par défaut étant QWERTY. Si vous éditez les sources Lua avec Emacs, notez qu'il n'y a pas de mode Lua pour Emacs dans la machine virtuelle. Bon, on s'en passe, sinon, on le télécharge avec `apt-get`. Les pcap d'exemple sont livrés avec la machine virtuelle, dans `/opt/haka/share/haka/sample/hellopacket`.

Premier exercice, s'entraîner aux bases de Haka et apprendre à afficher certains paquets. Allons-y en Lua :

```
-- On charge le dissecteur IPv4 (pas encore de dissecteur IPv6,
-- malheureusement ; un volontaire pour l'écrire ?)
local ip = require('protocol/ipv4')

-- Haka fonctionne en écrivant des *règles* qui réagissent à des
-- *évènements*, ici, l'évènement est le passage d'un paquet IPv4
haka.rule{
    hook = ip.events.receive_packet,
    -- Suit une série d'actions, qui reçoivent un paramètre qui dépend
    -- du dissecteur. Le dissecteur IPv4 passe à l'évènement
    -- receive_packet un paquet.
    eval = function (pkt)
        haka.log("Hello", "packet from %s to %s", pkt.src, pkt.dst)
    end
}
```

Et comment on a trouvé que les champs de `pkt` qui contenaient les adresses IP source et destination se nommaient `src` et `dst` ? On a lu la doc `<http://doc.haka-security.org/haka/release/v0.1/modules/protocol/ipv4/doc/lua.html#dissector>` (également disponible dans la machine virtuelle en `/lib/live/mount/medium/haka/manual/modules/protocol/ipv4/doc/ipv4.html#di`
Il existe aussi un mode interactif de Haka, permettant d'explorer les paquets, pas montré ici.

Si on lance ce script Haka sur un pcap existant, il affiche :

```
% hakapcap hellopacket.lua hellopcaket.pcap

info  core: load module 'packet/pcap.ho', Pcap Module
info  core: load module 'alert/file.ho', File alert
info  core: setting packet mode to pass-through

info  core: loading rule file 'hellopacket.lua'
info  core: initializing thread 0
info  dissector: register new dissector 'raw'
info  pcap:      opening file 'hellopacket.pcap'
info  dissector: register new dissector 'ipv4'
info  core:      1 rule(s) on event 'ipv4:receive_packet'
info  core:      1 rule(s) registered

info  core:      starting single threaded processing

info  Hello:     packet from 192.168.10.1 to 192.168.10.99
info  Hello:     packet from 192.168.10.99 to 192.168.10.1
info  Hello:     packet from 192.168.10.1 to 192.168.10.99
info  Hello:     packet from 192.168.10.1 to 192.168.10.99
info  Hello:     packet from 192.168.10.99 to 192.168.10.1
info  Hello:     packet from 192.168.10.1 to 192.168.10.99
info  Hello:     packet from 192.168.10.99 to 192.168.10.1
info  Hello:     packet from 192.168.10.1 to 192.168.10.99
info  core:      unload module 'Pcap Module'
info  core:      unload module 'File alert'
```

Ça a bien marché, chaque paquet du pcap a été affiché.

Deuxième exercice, filtrage des paquets qui ne nous plaisent pas, en l'occurrence, ceux qui viennent du méchant réseau `192.168.10.0/27` :

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

local bad_network = ip.network("192.168.10.0/27")

haka.rule{
    hook = ip.events.receive_packet,
    eval = function (pkt)
      -- On teste si le paquet appartient au méchant réseau
      if bad_network:contains(pkt.src) then
        -- Si oui, on le jette et on journalise
        haka.log("Dropped", "packet from %s to %s", pkt.src, pkt.dst)
pkt:drop()
      -- Si non (pas de 'else') le paquet suit son cours
      end
   end
}
```

Une fois lancé le script sur un pcap, on obtient :

```
...
info  Dropped:   packet from 192.168.10.1 to 192.168.10.99
info  Dropped:   packet from 192.168.10.1 to 192.168.10.99
info  Dropped:   packet from 192.168.10.1 to 192.168.10.99
info  Dropped:   packet from 192.168.10.1 to 192.168.10.99
info  Dropped:   packet from 192.168.10.10 to 192.168.10.99
info  Dropped:   packet from 192.168.10.10 to 192.168.10.99
info  Dropped:   packet from 192.168.10.10 to 192.168.10.99
```

Si vous regardez le pcap avec tcpdump, vous verrez qu'il y a d'autres paquets, en provenance de 192.168.10.99 qui n'ont pas été jetés, car pas envoyés depuis le méchant réseau :

```
14:49:27.076486 IP 192.168.10.1 > 192.168.10.99: ICMP echo request, id 26102, seq 1, length 64
14:49:27.076536 IP 192.168.10.99 > 192.168.10.1: ICMP echo reply, id 26102, seq 1, length 64
14:49:28.075844 IP 192.168.10.1 > 192.168.10.99: ICMP echo request, id 26102, seq 2, length 64
14:49:28.075900 IP 192.168.10.99 > 192.168.10.1: ICMP echo reply, id 26102, seq 2, length 64
14:49:31.966286 IP 192.168.10.1.37542 > 192.168.10.99.80: Flags [S], seq 3827050607, win 14600, options [mss 146
14:49:31.966356 IP 192.168.10.99.80 > 192.168.10.1.37542: Flags [R.], seq 0, ack 3827050608, win 0, length 0
14:49:36.014035 IP 192.168.10.1.47617 > 192.168.10.99.31337: Flags [S], seq 1811320530, win 14600, options [mss
14:49:36.014080 IP 192.168.10.99.31337 > 192.168.10.1.47617: Flags [R.], seq 0, ack 1811320531, win 0, length 0
14:49:46.837316 ARP, Request who-has 192.168.10.99 tell 192.168.10.10, length 46
14:49:46.837370 ARP, Reply 192.168.10.99 is-at 52:54:00:1a:34:60, length 28
14:49:46.837888 IP 192.168.10.10.35321 > 192.168.10.99.8000: Flags [S], seq 895344097, win 14600, options [mss 1
14:49:46.837939 IP 192.168.10.99.8000 > 192.168.10.10.35321: Flags [R.], seq 0, ack 895344098, win 0, length 0
14:49:50.668580 IP 192.168.10.10 > 192.168.10.99: ICMP echo request, id 26107, seq 1, length 64
14:49:50.668674 IP 192.168.10.99 > 192.168.10.10: ICMP echo reply, id 26107, seq 1, length 64
14:49:51.670446 IP 192.168.10.10 > 192.168.10.99: ICMP echo request, id 26107, seq 2, length 64
14:49:51.670492 IP 192.168.10.99 > 192.168.10.10: ICMP echo reply, id 26107, seq 2, length 64
14:49:51.841297 ARP, Request who-has 192.168.10.10 tell 192.168.10.99, length 28
14:49:51.841915 ARP, Reply 192.168.10.10 is-at 52:54:00:30:b0:bd, length 46
```

Bon, jusqu'à présent, on n'a rien fait d'extraordinaire, Netfilter en aurait fait autant. Mais le prochain exercice est plus intéressant. On va faire du filtrage TCP. La documentation du dissecteur TCP nous apprend que le champ qui indique le port de destination est `dstport` :

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

-- On charge un nouveau dissecteur
local tcp = require ('protocol/tcp_connection')
```

```
haka.rule{
    -- Et on utilise un nouvel évènement, qui signale un nouveau flot TCP
    hook = tcp.events.new_connection,
    eval = function (flow, pkt)
      -- Si le port n'est ni 22, ni 80...
      if flow.dstport ~= 22 and flow.dstport ~= 80 then
        haka.log("Dropped", "flow from %s to %s:%s", pkt.ip.src, pkt.ip.dst, flow.dstport)
flow:drop()
      else
        haka.log("Accepted", "flow from %s to %s:%s", pkt.ip.src, pkt.ip.dst, flow.dstport)
      end
   end
}
```

Et on le teste sur un pcap :

```
info  Accepted:  flow from 192.168.10.1 to 192.168.10.99:80
info  Dropped:   flow from 192.168.10.1 to 192.168.10.99:31337
info  Dropped:   flow from 192.168.10.10 to 192.168.10.99:8000
```

Il y avait trois flots (trois connexions TCP) dans le pcap, une seule a été acceptée.

Maintenant, fini de jouer avec des pcap. Cela peut être intéressant (analyse de pcap compliqués en ayant toute la puissance d'un langage de Turing) mais je vous avais promis un pare-feu. On va donc filtrer des paquets et des flots vivants, en temps réel. On écrit d'abord le fichier de configuration du démon Haka :

```
[general]
# Select the haka configuration file to use
configuration = "tcpfilter.lua"

# Optionally select the number of thread to use.
#thread = 4

# Pass-through mode
# If yes, haka will only inspect packet
# If no, it means that haka can also modify and create packet
pass-through = no

[packet]
# Select the capture model, nfqueue or pcap
module = "packet/nfqueue"

# Select the interfaces to listen to
interfaces = "lo"
#interfaces = "eth0"

# Select packet dumping for nfqueue
#dump = yes
#dump_input = "/tmp/input.pcap"
#dump_output = "/tmp/output.pcap"

[log]
# Select the log module
module = "log/syslog"

[alert]
# Select the alert module
module = "alert/syslog"
```

Le script est le même que dans l'essai précédent, il accepte uniquement les connexions TCP vers les ports 22 ou 80. On lance le démon (notez que celui-ci fera appel à Netfilter pour lui passer les paquets) :

```
% sudo haka -c haka.conf --no-daemon
info  core: load module 'log/syslog.ho', Syslog logger
info  core: load module 'alert/syslog.ho', Syslog alert
info  core: load module 'alert/file.ho', File alert
info  core: load module 'packet/nfqueue.ho', nfqueue
info  nfqueue: installing iptables rules for device(s) lo
info  core:    loading rule file 'tcpfilter.lua'
info  core:    initializing thread 0
info  dissector: register new dissector 'raw'
info  dissector: register new dissector 'ipv4'
info  dissector: register new dissector 'tcp'
info  dissector: register new dissector 'tcp_connection'
info  core:      1 rule(s) on event 'tcp_connection:new_connection'
info  core:      1 rule(s) registered

info  core:      starting single threaded processing
```

Et on tente quelques connexions (143 = IMAP) :

```
info  Accepted:  flow from 127.0.0.1 to 127.0.0.1:80
info  Dropped:   flow from 127.0.0.1 to 127.0.0.1:143
info  Dropped:   flow from 127.0.0.1 to 127.0.0.1:143
info  Dropped:   flow from 127.0.0.1 to 127.0.0.1:143
```

Et en effet, les clients IMAP vont désormais "*timeouter*". Dès qu'on arrête le démon, avec un Control-C, IMAP remarche :

```
% telnet 127.0.0.1 imap2
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
* OK [CAPABILITY IMAP4rev1 LITERAL+ SASL-IR LOGIN-REFERRALS ID ENABLE IDLE STARTTLS AUTH=PLAIN] Dovecot ready.
```

J'ai beaucoup aimé la possibilité de journaliser ou de jeter un flot TCP entier, pas en agissant paquet par paquet. Pour la partie IP au début de cet atelier, Haka a un concurrent évident, Scapy (mêmes concepts mais Python au lieu de Lua). Mais Scapy ne sait pas gérer les flots TCP, il ne travaille que par paquet.

Exercice suivant, le protocole du Web, HTTP. On va modifier les pages HTML en vol (et vous comprendrez pourquoi il faut toujours utiliser HTTPS). Il faut changer haka.conf pour indiquer le nouveau script, `blurring-the-web.lua`. D'abord, on se contente d'afficher les requêtes HTTP :

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

-- On charge un nouveau dissecteur
local http = require ('protocol/http')

-- Pas d'indication du protocole supérieur dans TCP, seulement du
-- numéro de port, donc il faut être explicite
http.install_tcp_rule(80)
```

```
haka.rule{
    -- Et on utilise un nouvel évènement
    hook = http.events.request,
    eval = function (connection, request)
      haka.log("HTTP request", "%s %s", request.method, request.uri)
      -- Il faut être sûr que les données ne soient pas comprimées,
      -- pour que la modification ultérieure marche. On modifie donc
      -- la requête.
      request.headers['Accept-Encoding'] = nil
      request.headers['Accept'] = "*/*"
    end
}
```

Haka fait beaucoup de choses mais il ne décomprime pas les flots HTTP. Comme la compression est souvent utilisée sur le Web <https://www.bortzmeyer.org/gzip-compression-apache.html>, on modifie les en-têtes de la requête pour prétendre qu'on n'accepte pas la compression (cf. RFC 7231 [1], sections 5.3.4 et 5.3.2).

On lance maintenant le démon Haka avec ce nouveau script. Pour tester que les requêtes du navigateur Web ont bien été modifiées, on peut aller sur un site qui affiche les en-têtes de la requête comme http://www.bortzmeyer.org/apps/env. Par défaut, le Firefox de la machine virtuelle envoie :

```
HTTP_ACCEPT: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
HTTP_ACCEPT_ENCODING: gzip, deflate
```

alors qu'une fois le démon en route, on n'a plus que (mettre une entrée de tableau à `nil` la détruit) :

```
HTTP_ACCEPT: */*
```

Attention au passage : Firefox peut maintenir des connexions TCP persistantes avec le serveur HTTP. Si vous lancez Firefox **avant** Haka, vous risquez de récupérer fréquemment des :

```
alert: id = 8
time = Sat Jul 19 16:31:32 2014
severity = low
description = no connection found for tcp packet
sources = {
address: 127.0.0.1
service: tcp/56042
}
targets = {
address: 127.0.0.1
service: tcp/80
}
```

C'est parce que Haka a vu passer des paquets d'une connexion TCP antérieure à son activation et qu'il ne sait donc pas rattacher à un flot qu'il suit. (C'est le problème de tous les filtres à état.)

Attention aussi, Haka ne gère pas IPv6. Si on se connecte à un serveur Web en indiquant son nom, on peut utiliser IPv6 (c'est le cas de http://localhost/ sur la Debian de la machine virtuelle) et Haka ne verra alors rien. D'où l'option `-4` de wget, pour tester :

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc7231.txt

```
% wget -v -4 http://localhost/
...
# Haka affiche
info  HTTP request: GET /
```

Maintenant, on ne va pas se contenter d'observer, on modifie la réponse HTTP en ajoutant du CSS rigolo :

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

local re = require('regexp/pcre')

-- Ce CSS rend tout flou
local css = '<style type="text/css" media="screen"> * { color: transparent !important; text-shadow: 0 0 3px blac

http.install_tcp_rule(80)

-- Bien garder cette règle, pour couper la compression, qui est
-- activée par défaut sur beaucoup de serveurs
haka.rule{
    hook = http.events.request,
    eval = function (connection, request)
      haka.log("HTTP request", "%s %s", request.method, request.uri)
      request.headers['Accept-Encoding'] = nil
      request.headers['Accept'] = "*/*"
    end
}

-- Deuxième règle, pour changer la réponse
haka.rule{
    hook = http.events.response_data,
    options = {
        streamed = true,
    },
    eval = function (flow, iter)
      -- Chercher la fin de l'élément <head>
      local regexp = re.re:compile("</head>", re.re.CASE_INSENSITIVE)
      local result = regexp:match(iter, true)
      -- Si on a bien trouvé un <head>
      if result then
         result:pos('begin'):insert(haka.vbuffer_from(css))
      end
    end
}
```

Et re-testons :

```
% wget -O - -v -4 http://localhost/
--2014-07-19 16:41:48--  http://localhost/
Resolving localhost (localhost)... 127.0.0.1, 127.0.0.1
Connecting to localhost (localhost)|127.0.0.1|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 190 [text/html]
Saving to: 'STDOUT'

 0% [                                       ] 0           --.-K/s              <html><head><style type="
100%[==========================================>] 190         --.-K/s   in 0s

2014-07-19 16:41:48 (45.0 MB/s) - written to stdout [190/190]
```

Le CSS a bien été inséré ! Et, avec un vrai site Web, on voit bien l'effet :

Attention, le `haka.conf` indiqué ici n'écoute que sur l'interface `lo`. Pour tester sur le Web en grand, pensez à changer le `haka.conf` (et rappelez-vous que Haka n'aura aucun effet sur les sites Web accessibles en IPv6 comme `<http://www.ietf.org/>`, si la machine virtuelle a une connectivité IPv6).

Voilà, sur ce truc spectaculaire, ce fut la fin de l'atelier. À noter que nous n'avons pas testé les performances de Haka, question évidemment cruciale pour des filtrages en temps réel. En première approximation, Haka semble bien plus rapide que Scapy pour les tâches d'analyse de pcap mais il faudra un jour mesurer sérieusement.