Incident Response and Malware Analysis
IRMA : plate-forme d'analyse de fichiers

Alexandre Quint, Guillaume Dedrie et Fernand Lone-Sang
{aquint,gdedrie,flonesang}@quarkslab.com

QuarksLAB – 71, Avenue des Ternes – 75017, Paris, France

Résumé De nos jours, faire reposer la sécurité d'un système uniquement sur un anti-virus est un pari osé. Même si les anti-virus restent un outil nécessaire dans la détection des vecteurs d'attaque génériques, leur efficacité reste encore à démontrer.

L'idée pour les exploiter au mieux n'est pas nouvelle. Elle consiste à combiner les résultats d'un ensemble d'anti-virus permettant ainsi de réduire la menace de codes malveillants qui pèse sur les systèmes. D'ailleurs, faire analyser un code par plusieurs anti-virus est souvent la première étape pour un analyste de malware. Celui-ci va vouloir, par la suite, pousser son analyse en le désassemblant, en l'exécutant dans une *sandbox*, mais aussi en appliquant des outils internes.

Cet article décrit IRMA [1] (Incident Response and Malware Analysis), une plate-forme privée et *open-source* d'analyse de fichiers. Nous allons rappeler les objectifs de cette plate- forme. Puis, nous détaillerons le fonctionnement de celle-ci en mettant en avant son aspect modulaire. Cet article est aussi l'occasion de faire un retour sur son développement (problématiques techniques rencontrées), de présenter quelques statistiques sur les anti-virus, mais surtout d'expliquer comment l'enrichir facilement.

1 Introduction

1.1 Objectifs

La lutte contre les *malwares* ne fait que s'intensifier au cours du temps. En effet, même si leur typologie a changé, leur nombre a, quant à lui, considérablement augmenté. Nos systèmes sont, de plus en plus, sous la menace des nouveaux codes malveillants qui apparaissent quotidiennement (390000 nouvelles menaces par jour selon AV-TEST [2]).

En outre, la diversité des anti-virus a également un coût organisationnel et financier :

— organisationnel, car cela demande aux équipes de gérer, sur les postes de travail et les passerelles en tout genre, des logiciels qui ne sont pas toujours compatibles les uns avec les autres ;

1. http://irma.quarkslab.com
2. http://www.av-test.org/fr/statistiques/malware/

— financier, car ces logiciels coûtent cher, et qu'il est impossible de mettre de nombreux anti-virus sur un poste de travail.

De plus, une organisation est souvent liée contractuellement avec un éditeur d'anti-virus. Or un même *malware* est rarement unanimement reconnu par tous les anti-virus. En terme de sécurité, mettre un seul anti-virus sur un poste n'est pas satisfaisant.

Pour ces différentes raisons, disposer d'une solution anti-virale multi-vendeurs est une brique de réponse à ces problèmes et c'est à cette problématique que s'attaque la plate-forme IRMA.

1.2 Présentation de la plate-forme IRMA

Actuellement, IRMA est, uniquement, une solution d'analyse de fichiers : il est possible de soumettre plusieurs fichiers simultanément afin de détecter certains codes malveillants ou de faire une recherche sur le nom ou le *hash* d'un fichier déjà connu du système.

Dans sa philosophie, IRMA tente de se rapprocher de Metasploit. En effet l'objectif, plus que de fournir une plate-forme dans laquelle vous avez un contrôle total de vos données, est de fournir un *framework* d'analyse de fichier. N'importe qui peut, grâce à une courbe d'apprentissage très faible, y ajouter son propre outil d'analyse tout en ignorant les problèmes de soumission, d'archivage, de distribution des tâches. La plus-value est grande pour un analyste de *malware*. En effet, grâce à cela, il sera très facile pour ce dernier d'automatiser les différentes étapes de son analyse.

1.3 Architecture et fonctionnement détaillé

L'architecture a été découpée en trois parties distinctes :

Le frontend est responsable de la partie interface avec le monde extérieur via l'API. C'est lui qui va s'occuper du stockage des données (fichiers soumis ainsi que les résultats d'analyse des *probes*) ;

Le brain est responsable de la partie soumission des tâches d'analyse aux différentes *probes* du système ;

Les probes sont responsables de l'analyse du fichier reçu et de l'envoi du résultat vers le *frontend*.

Dans la suite de l'article, on parle de module d'analyse et de *probe*. Le module d'analyse est la partie du code en charge du support d'une analyse (un antivirus, un site externe...) dans IRMA et la probe désigne l'application complète qui exécute ce module.

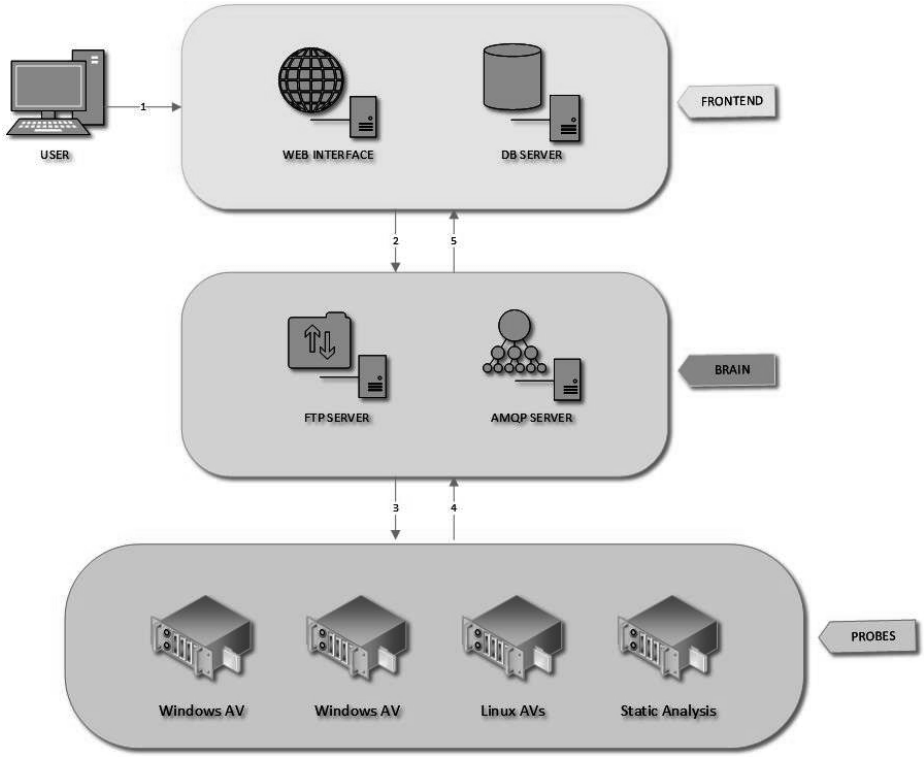

Figure 1. Architecture globale

Fonctionnement détaillé En se référant aux numéros de la figure 1 le parcours d'un fichier dans IRMA est le suivant :

1. Un utilisateur soumet via l'interface web un ou plusieurs fichiers à analyser. Les fichiers sont stockés sur le *frontend* et indexés en base de données.

2. Une tâche asynchrone sur le *frontend* transfère les fichiers sur le serveur FTP et déclenche à son tour une tâche asynchrone sur le *brain* qui lance l'analyse.

3. Le *brain* divise cette analyse en sous-tâches pour chacun des fichiers soumis et chacune des *probes* demandées.

4. Chaque probe renvoie son résultat d'analyse au *brain* qui gère l'avancement de la tâche globale d'analyse.

5. Le *brain* renvoie les résultats vers le *frontend* qui les stocke en base de données et les met à disposition de l'utilisateur à travers l'API.

2 Évolution des technologies utilisées

IRMA est un projet développé principalement en python qui s'appuie sur de nombreux composants standards (API web servie par Nginx et uWSGI par défaut), un serveur FTP pour le transfert des fichiers (Pure-FTPd par défaut), une base de donnée hybride SQL et NoSQL (PostgreSQL et MongoDB par défaut) et enfin Celery pour la gestion des tâches asynchrones. La partie Celery gère les files d'attentes au niveau du *frontend*, du *brain* ou des *probes*, permettant à un processus de traitement d'avoir une file de tâches, avec la possibilité de les gérer (gestion des priorités, annulation, reprise après échec).

2.1 Retours sur le packaging

Packaging PIP La première version publique d'IRMA (version 1.0.2) date du 2 juin 2014 et a été annoncée en RUMP session de l'édition du SSTIC de la même année. Elle supportait quelques modules d'analyses et demandait un effort conséquent de la part de l'utilisateur pour son installation. Le code source d'IRMA pour chacun des rôles logiques devait être récupéré sur les dépôts github éponymes. Il fallait ensuite installer tous les composants annexes (serveur FTP, RabbitMQ, uWSGI, Nginx, Redis, MongoDB) et configurer chacune des machines hébergeant un rôle avec les adresses et données d'authentification pour chacun de ces services.

Packaging Debian La version 1.0.4, sortie le 28 août 2014, a grandement facilité l'installation en s'appuyant sur le système de packaging Debian. Tous les serveurs requis étaient installés en tant que dépendance des paquets sources et des scripts de post-installation proposaient à l'utilisateur de générer les fameux fichiers de configuration évoqués plus haut.

Technologies DevOps Malgré l'amélioration du processus d'installation en 1.0.4, la génération des fichiers de configuration restait une étape trop fastidieuse. La version 1.1.0 a introduit une toute nouvelle méthode d'installation automatisée, en utilisant Ansible [3], un logiciel open- source de déploiement de configuration au travers d'une connexion ssh.

De plus, et afin de faciliter le développement du projet, nous avons mis en place plusieurs environnements (développement, production ou test) à l'aide de Vagrant [4], un logiciel open-source facilitant la création, la

3. `http://www.ansible.com/`
4. `https://www.vagrantup.com/`

configuration et l'administration d'environnements virtualisés (utilisant VirtualBox, VMWare, Libvirt ...).

Une fois l'environnement initialisé à l'aide de Vagrant, c'est Ansible qui va se charger d'installer tous les composants nécessaires au bon fonctionnement d'IRMA. C'est aussi lui qui va générer les fichiers de configurations et assurer ainsi la synchronisation des mots de passes entre les différents clients et serveurs. Un autre avantage d'Ansible est qu'il peut être piloté directement par Vagrant, ainsi la création des machines virtuelles et leur configuration se fait en une seule commande.

2.2 Suppression de Redis

Par défaut, Celery recommande l'utilisation d'un serveur d'ordonnancement AMQP (Advanced Message Queuing Protocol [5]), RabbitMQ dans le cas d'IRMA, et de Redis [6], une base de donnée clé-valeur, comme gestionnaire d'état et d'archivage des résultats des tâches.

Les versions d'IRMA avant 1.1.0 utilisaient Celery dans cette configuration et Redis était déployé sur la machine hébergeant le *brain*, tout en étant accessible depuis le *frontend* ou les *probes*. Laissant ouvert un problème de sécurité important. Redis n'a effectivement aucun schéma d'authentification et sur le site officiel est indiqué que Redis est supposé tourner dans un environnement de confiance [7].

Outre les aspects sécurités, la suppression de Redis permet aussi de réduire le nombre de dépendances, déjà importantes, du projet. La partie permettant le cache des résultats a été migré vers AMQP (un autre mode de fonctionnement courant de Celery) et la partie permettant l'annulation des tâches, et donc la connaissance à tout moment, de l'état de toutes les sous-tâches a été ré-implémenté en SQLite, composant déjà présent sur le *brain*.

2.3 Passage à une base hybride SQL+NoSQL

De même dans les versions d'IRMA inférieures à 1.1.0, toutes les informations étaient stockées dans une base de données MongoDB. Le principal avantage à utiliser une base NoSQL, était de ne pas devoir figer le format des résultats issus des modules d'analyses. Le problème est que les données sont très fortement liées, une analyse comporte plusieurs fichiers et plusieurs résultats pour chacun de ces fichiers. Une nouvelle analyse

5. `https://www.amqp.org/`
6. `http://redis.io/`
7. `http://redis.io/topics/security`

doit être capable de lier les fichiers de celle-ci avec ceux déjà présents en base, indiquer de quelles analyses chacun fait partie et de retrouver les résultats liés à chacune des analyses. Toutes ces relations peuvent être implémentées dans une base NoSQL mais au prix de recopie inutile et de surcoût en espace de stockage et performance. L'approche a donc été de séparer la partie fortement relationnelle de celle qui ne l'est pas. Stocker dans la base de données SQL, les données relatives à l'analyse nommée *Scan*, aux fichiers nommés *ScanFile*, à leurs résultats nommées *ScanResults*. Et stocker dans la base de données NoSQL les données qu'on ne voulait pas contraindre dans leur format : les résultats bruts d'analyses.

3 Installation

Toutes les instructions de cet article sont données pour une machine Linux. Le but est d'installer un environnement de développement pour guider le lecteur dans la création d'un nouveau module d'analyse. Les instructions sont également valables pour le lecteur qui voudrait simplement tester IRMA.

3.1 Installation de l'environnement de développement

L'installation automatisée de l'environnement de développement décrite est réalisée à l'aide de Vagrant et Ansible, la partie virtualisation repose sur VirtualBox (ce sont les trois seules dépendances à installer sur la machine hôte). Dans la suite de l'article, la machine hôte sera notée `host` et la machine virtuelle hébergeant IRMA `vm`.

Pour installer Vagrant dans une version récente, se référer directement à leur documentation [8].

```
host-$ vagrant --version
   Vagrant 1.7.2
```

Pour installer Ansible dans une version récente, il suffit d'utiliser le gestionnaire de paquets de python (ou votre package manager, mais attention à la version disponible qui doit être supérieure à 1.8) :

```
host-$ sudo pip install ansible
host-$ ansible --version
   ansible 1.8.2
```

Avant de lancer l'installation d'IRMA, il faut cloner les dépôts sources depuis la branche stable (master) :

8. `https://www.vagrantup.com/downloads.html`

```
host-$ git clone --recursive https://github.com/quarkslab/irma-
    frontend
host-$ git clone --recursive https://github.com/quarkslab/irma-
    brain
host-$ git clone --recursive https://github.com/quarkslab/irma-
    probe
```

et le dépôt github contenant les scripts d'installation IRMA pour Vagrant et Ansible :

```
host-$ git clone --recursive https://github.com/quarkslab/irma-
    ansible
```

Une fois cette étape faite, il faut installer les rôles Ansible, qui peuvent être vus comme des plugins facilitant la configuration des serveurs classiques (web, ftp...). Cette installation se fait via la commande `ansible-galaxy` installée avec Ansible, les rôles requis étant listés dans le fichier `ansible-requirements.yml` :

```
host-$ cd irma-ansible
host-$ ansible-galaxy install -r ansible-requirements.yml
```

Des modèles de configurations (plus précisément des environnements Vagrant) permettent de facilement passer d'une configuration à une autre. Les configurations fournies par défaut sont les suivantes :

- `prod.yml` environnement de production, chaque machine virtuelle n'héberge qu'un seul rôle logique. Le code source est cloné de la branche master des dépôts IRMA.
- `allinone_prod.yml`, environnement de production allégé, tout est installé sur une seule machine virtuelle. Le code source est cloné de la branche master des dépôts IRMA. C'est l'environment par défaut.
- `dev.yml`, environnement de développement, chaque machine virtuelle n'héberge qu'un seul rôle logique. Le code source est synchronisé depuis la machine hôte.
- `allinone_dev.yml`, environnement de développement allégé, tout est installé sur une seule machine virtuelle. Le code source est synchronisé depuis la machine hôte. Nous allons utiliser l'environnement `allinone_dev` dédié au développement. Il a de plus l'avantage de tourner sans trop de pré-requis sur la machine hôte. Pour la synchronisation, si tous les dépôts n'ont pas été cloné au même endroit, il faut mettre à jour le chemin des sources dans le fichier `environments/allinone_dev.yml` :

```
    shares:
      - share_from: ../irma-frontend ← mettre à jour avec
          votre chemin
        share_to: /opt/irma/irma-frontend/current
        share_exclude:
          - .git/
          - venv/
          - web/dist
          - web/node_modules
          - app/components
      - share_from: ../irma-brain ← mettre à jour avec votre
          chemin
        share_to: /opt/irma/irma-brain/current
        share_exclude:
         - .git/
          - venv/
          - db/
      - share_from: ../irma-probe ← mettre à jour avec votre
          chemin
        share_to: /opt/irma/irma-probe/current
        share_exclude:
          - .git/
- venv/
```

Pour lancer l'installation, il suffit de lancer Vagrant en précisant l'environnement choisi :

```
host-$ cd irma-ansible
host-$ VM_ENV=allinone_dev vagrant up
```

Au premier lancement, l'image de référence qui va servir de base pour toutes les machines virtuelles Linux est téléchargée depuis le dépôt officiel des images Vagrant de référence nommé VagrantCloud (ou récemment renommé en Atlas [9]). Cette image de référence, une Debian 7.7 64 bits, a été générée et mise en ligne par nos soins (les sources de génération sont accessibles à l'adresse `https://github.com/quarkslab/debian`). On obtient ainsi une machine virtuelle complètement installée, accessible à l'adresse 172.16.1.30, nommée brain.irma dans le fichier de configuration Vagrant. À l'aide d'un navigateur, on peut vérifier que le système est fonctionnel et connaître la liste des *probes* activées en affichant les réglages avancés comme sur la figure 2.

Après un test du système en soumettant un ou plusieurs fichiers à analyser le lien vers les résultats détaillés d'un fichier ressemble à la figure 3.

La partie installation est terminée et la partie développement commence. Dès qu'un des répertoires sources du *frontend*, du *brain* ou d'une

9. `https://atlas.hashicorp.com/`

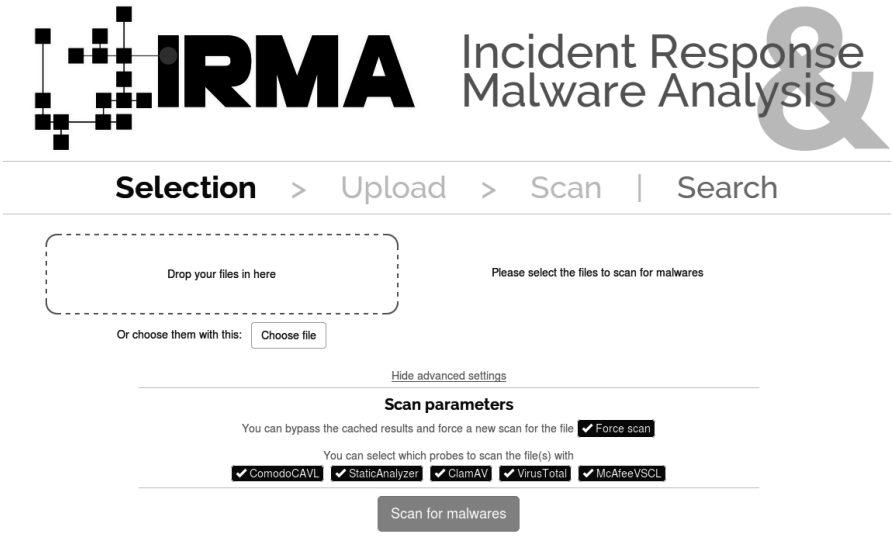

Figure 2. Page d'accueil de l'interface web

probe sera modifié, il faudra le synchroniser avec la machine virtuelle en utilisant la commande Vagrant 3.1.

```
host-$ cd irma-ansible
host-$ VM_ENV=allinone_dev vagrant rsync
```

Si vous ne souhaitez pas avoir à retaper cette commande à chaque changement, l'alternative revient à dédier un terminal à la commande :

```
host-$ VM_ENV=allinone_dev vagrant rsync-auto
```

Pour se connecter facilement à la machine virtuelle créée on peut utiliser la commande Vagrant :

```
host-$ cd irma-ansible
host-$ VM_ENV=allinone_dev vagrant ssh brain.irma
```

De même pour éteindre ou allumer la machine virtuelle la commande :

```
host-$ cd irma-ansible
host-$ VM_ENV=allinone_dev vagrant halt
host-$ VM_ENV=allinone_dev vagrant up
```

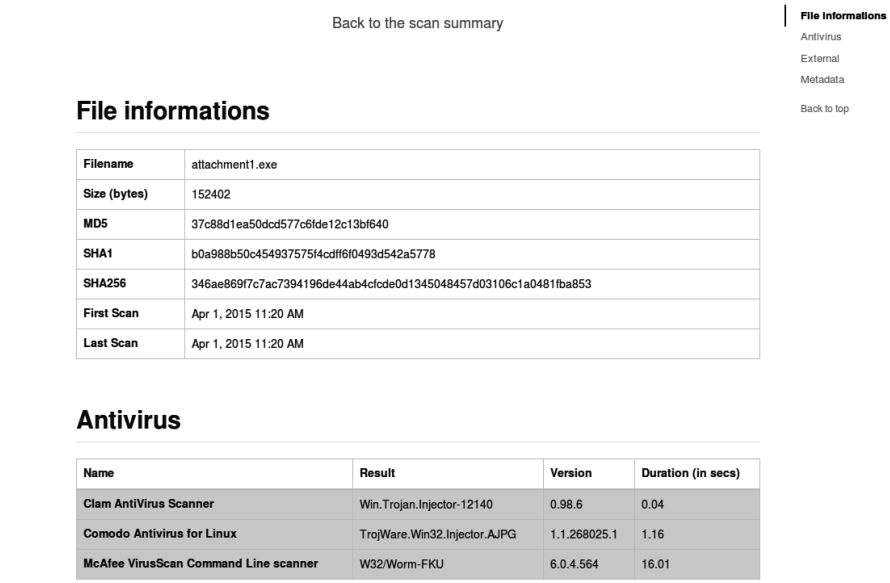

Figure 3. Page des résultats détaillés

4 Introduction aux modules d'analyse

4.1 Organisation du code de irma-probe

L'organisation du code source des *probes* est le suivant :

```
host-$ tree -L 1 -d irma-probe/
irma-probe/
|__ config
|__ debian
|__ extras
|__ lib
|__ modules
|__ probe
|__ tests
|__ tools
```

— *config* : contient la configuration de la *probe* avec notamment les adresses et données d'authentification des serveurs RabbitMQ, SQL et FTP.
— *modules* : contient tous les modules d'analyses supportés par IRMA (c'est ici que sera rajouté le code du nouveau module d'analyse).
— *probe* : contient le code de la tâche Celery exécutée au démarrage de la *probe* nommé tasks.py.

4.2 Découverte des modules d'analyses

La partie intéressante de ce fichier tasks.py est la suivante :

```
    # discover plugins located at specified path
    plugin_path = os.path.abspath("modules")
    if not os.path.exists(plugin_path):
        log.error("path {0} is invalid, cannot load probes".format(
            plugin_path))
        sys.exit(1)
    manager = PluginManager()
    manager.discover(plugin_path)
```

Le répertoire *modules* est parcouru à la recherche de modules d'analyses capables de se lancer. À chaque démarrage de la *probe*, les modules sont automatiquement découverts et activés pour ceux qui peuvent l'être.

4.3 Exemple de module existant

Dans le répertoire `modules` on retrouve le code de tous ceux supportés dans la version stable 1.1.1 :

```
    host-$ tree -L 2 -d modules/
    modules/
    |__ antivirus
    |   |__ clamav
    |   |__ comodo
    |   |__ eset
    |   |__ fprot
    |   |__ gdata
    |   |__ kaspersky
    |   |__ mcafee
    |   |__ sophos
    |   |__ symantec
    |__ custom
    |   |__ skeleton
    |__ database
    |   |__ nsrl
    |__ external
    |   |__ virustotal
    |__ metadata
        |__ pe_analyzer
        |__ yara
```

On peut noter la présence d'un module d'exemple, nommé `skeleton` qui peut servir de modèle dans le développement d'un nouveau module. Prenons l'exemple du module *clamav* :

```
  host-$ tree modules/antivirus/clamav/
  modules/antivirus/clamav/
  |__ clam.py
  |__ __init__.py
  |__ plugin.py
```

Le fichier `clam.py`, spécifique au module, renseigne toutes les particularités de l'antivirus : le nom de l'exécutable en ligne de commande, les options à utiliser, la liste des fichiers de signatures impliqués dans la détection (pour pouvoir suivre l'évolution des détections indépendamment de la version retournée par l'éditeur) et les expressions régulières capables de traiter la sortie standard.

Le fichier `plugin.py` fait l'interface avec IRMA en gérant les dépendances, et les formats des entrées / sorties. Par exemple, avant de charger la *probe clamav*, on va tester que l'environnement est bien un Linux et que le binaire *clamdscan* est présent.

4.4 Définition des dépendances

Pour décrire toutes les dépendances d'un module d'analyse, l'utilisateur dispose des classes de dépendances suivantes :

- BinaryDependency : dépendance sur l'existence d'un binaire dans le *PATH*;
- ModuleDependency : dépendance sur l'existence d'un module dans le *PATH* python ;
- FileDependency : dépendance sur l'existence d'un fichier ;
- FolderDependency : dépendance sur l'existence d'un répertoire ;
- PlatformDependency : dépendance sur le type de plate-forme.

Pour *clamav* on trouve donc :

```
_plugin_dependencies_ = [
    PlatformDependency('linux'),
    BinaryDependency(
        'clamdscan',
        help='clamdscan is provided by clamav-daemon debian
            package.'
    ),
]
```

Il faut aussi spécifier les dépendances *python* en ajoutant un fichier `requirements.txt`. Cela facilitera l'automatisation de l'installation avec Ansible.

4.5 Format de retour

Le format des retours est libre mais oblige la présence de certaines valeurs :

- *name* : nom de la *probe*
- *type* : catégorie de la *probe* (`antivirus`, `external`, `database` ou `metadata`)

— *version* : numéro de version
— *platform* : type de plate-forme (`windows`, `linux`...)
— *duration* : durée en seconde
— *status* : code de retour (<0 erreurs, >0 spécifique par *probe*)
— *error* : vide si pas d'erreur, chaîne avec l'erreur sinon
— *results* : résultats de la *probe*
avec comme type pour chacune des sous-clés :

```
{
  'name': str(),
  'type': str(),
  'version': str(),
  'platform': str(),
  'duration':int,
  'status':int,
  'error':None or str(),
  'results':object
}
```

5 Création d'un module d'analyse

5.1 Écriture du nouveau module

Pour commencer l'écriture d'un nouveau module, le plus simple est de partir de l'exemple fourni situé dans le répertoire `irma-probe/modules/custom/skeleton`, les métadonnées du nouveau module sont les premières choses à modifier. Ensuite il faut lui donner un nom, remplir la partie auteur, et lui assigner une des catégories (existante de préférence pour être prise en compte par l'interface). Ces métadonnées sont renseignées dans la partie suivante :

```
class SkeletonPlugin(PluginBase):

    class SkeletonResult:
        ERROR = -1
        FAILURE = 0
        SUCCESS = 1

    # =================
    #  plugin metadata
    # =================
    _plugin_name_ = "Skeleton"
    _plugin_author_ = "<author name>"
    _plugin_version_ = "<version>"
    _plugin_category_ = "custom"
    _plugin_description_ = "Plugin skeleton"
    _plugin_dependencies_ = []
```

Une méthode *verify* permet de rajouter un test avant le chargement d'un module. Un exemple concret est la vérification d'un fichier de configuration. La dépendance *FileDependency* va s'assurer que le fichier de configuration existe et la méthode *verify* s'assurera que le fichier contient bien les valeurs requises. Notez que le module d'exemple lève systématiquement une exception pour ne pas être chargé :

```
    # =============
    #  constructor
    # =============

    def __init__(self):
        pass

    @classmethod
    def verify(cls):
        raise PluginLoadError("Skeleton plugin is not meant to be
            loaded")
```

Pour faciliter la création des retours on peut instancier un objet de type *PluginResult* et le mettre à jour avec les données issues de notre analyse notamment la durée (*duration*), le code de retour (*status*) et les résultats (*results*).

```
    # ==================
    #  probe interfaces
    # ==================
    def run(self, paths):
        response = PluginResult(name=type(self).plugin_name,
                                type=type(self).plugin_category,
                                version=None)
        try:
            started = timestamp(datetime.utcnow())
            response.results = "Main analysis call here"
            stopped = timestamp(datetime.utcnow())
            response.duration = stopped - started
            response.status = self.SkeletonResult.SUCCESS
        except Exception as e:
            response.status = self.SkeletonResult.ERROR
            response.results = str(e)
        return response
```

5.2 Test du nouveau module

Une fois que l'écriture de notre nouveau module est terminée, on peut facilement le tester, après avoir synchronisé le code source avec la machine virtuelle et en s'y connectant :

```
host-$ cd irma-ansible
host-$ VM_ENV=allinone_dev vagrant rsync
host-$ VM_ENV=allinone_dev vagrant ssh brain.irma
```

Une fois connecté, si nécessaire, on installe les dépendances du nouveau module spécifié dans le fichier `irma-probe/modules/<category>/<probe>/requirements.txt`. Ensuite, on dispose d'un script en ligne de commande capable de détecter les modules présents et de les exécuter sur un fichier :

```
vm-$ venv/bin/python -m tools.run_module <probe> <fichier>
{'duration': 0.0008828639984130859,
 'error': None,
 'name': '<probe>',
 'platform': 'linux2',
 'results': <results>,
 'status': 0,
 'type': <category>,
 'version': <version>}
```

Si le résultat est correct, le module est prêt à être intégré dans une *probe*. Pour cela, il suffit de relancer la partie applicative correspondante à l'aide de Supervisord :

```
vm-$ sudo supervisorctl status
vm-$ sudo supervisorctl restart probe_app
probe_app: stopped
probe_app: started
vm-$ sudo supervisorctl tail probe_app
WARNING:probe.tasks: *** [antivirus] Plugin ComodoCAVL
    successfully loaded
WARNING:probe.tasks: *** [metadata] Plugin StaticAnalyzer
    successfully loaded
WARNING:probe.tasks: *** [antivirus] Plugin ClamAV successfully
    loaded
WARNING:probe.tasks: *** [antivirus] Plugin McAfeeVSCL
    successfully loaded
WARNING:probe.tasks: *** [<category>] Plugin <probe> successfully
    loaded
WARNING:probe.tasks: *** [external] Plugin VirusTotal successfully
     loaded
```

Et de vérifier que notre module `<probe>` est bien démarré. Le nouveau module est désormais activé.

5.3 Aller plus loin avec Ansible

Si le nouveau module d'analyse utilise des outils particuliers, il est tout a fait possible d'utiliser Ansible pour en automatiser l'installation.

Commencez par créer un nouveau rôle dans le répertoire `irma-ansible/roles`. La convention est de le nommer `NomAuteur.NomRole`.

Dedans, il va falloir créer un fichier `tasks/main.yml`, qui est le fichier d'entrée utilisé par Ansible pour l'exécution du rôle.

Celui-ci contiendra un ensemble de tâches qui seront exécutées à la suite les unes des autres.

Chaque tâche est organisée de la façon suivante :

```
---

- name: Nom de la âtche
  nom_module: # Voir la liste des modules : http://docs.ansible.com/
      list_of_all_modules.html
    module_arg1: valeur
    module_args2: valeur
  tache_arg: valeur # par exemple, executer une tache grace a sudo
```

Prenons par exemple l'installation de ClamAV :

```
---

- name: ClamAV | Install package
  apt:
    name: "{{ item }}"
    state: latest
  with_items:
    - clamav
    - clamav-base
    - clamav-freshclam
    - clamav-daemon
    - clamav-unofficial-sigs

- name: ClamAV | Update virus definition
  shell: freshclam --quiet
  sudo: true

- name: ClamAV | restart daemon
  service:
    name: clamav-daemon
    state: restarted
```

La première tâche va installer à l'aide d'*apt* les paquets requis, la deuxième tâche va lancer une mise à jour des signatures et la dernière tâche redémarrera le service *clamav-daemon*.

Après avoir construit votre rôle selon vos besoins, il faut maintenant l'ajouter à la liste des rôles à exécuter lors de l'installation d'IRMA. Pour cela, il faut modifier le fichier `playbooks/provisioning.yml`.

```
- name: NomRole provisioning
  hosts: NomRole
  sudo: yes # si besoin
  roles:
    - { role: NomAuteur.NomRole, tags: 'NomRole' }
```

La dernière étape consiste à informer Vagrant de quelles machines devront exécuter ce rôle. Dans notre cas, le fichier à modifier se trouve dans le répertoire `environments`. Il s'agit du fichier `allinone\_dev.yml` que nous avons utilisé tout au long de cet article. Il suffit d'y ajouter, avant `"probe:children":` :

```
    NomRole:
      - brain.irma
```

Le nouveau module sera déployé automatiquement lors de la prochaine installation. Pour tester, on peut détruire notre environnement et le recréer :

```
host-$ vagrant destroy
host-$ VM_ENV=allinone_dev vagrant up
```

5.4 Documentation de l'API

L'API web qui tourne sur le *frontend* permet de créer son propre client IRMA. Par défaut, le projet est livré avec deux exemples de clients de cette API :

— une interface WEB statique ;
— un client ligne de commande en python.

Une documentation dynamique, décrivant les routes accessibles de cette API ainsi que les paramètres associés, est accessible sur la machine virtuelle à l'adresse `http://172.16.1.30/swagger` (cf. figure 4).

6 Retour sur les Anti-virus

Les résultats donnés dans cet article ont été obtenus sur notre serveur de test (cpu Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz (8 cœurs), 32Go de mémoire vive). Les tests ont été effectués avec 2465 *malwares* publics téléchargés sur virussign [10]. L'environnement repose sur 23 machines virtuelles : un frontend, un brain et 21 *probes* linux.

10. `http://samples.virussign.com/samples/`

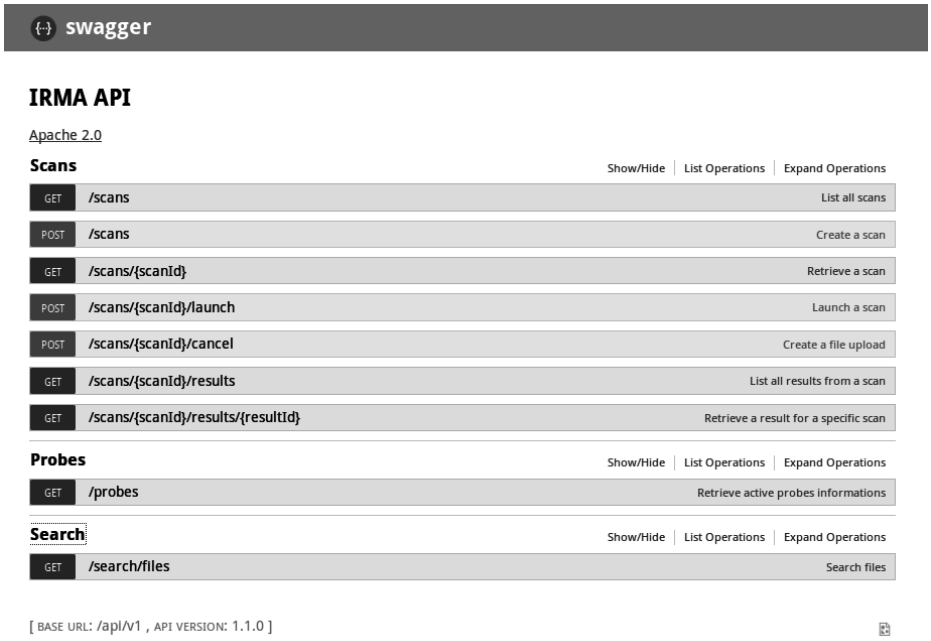

Figure 4. Documentation de l'API

6.1 Vitesse d'exécution

probe	min	max	moyenne
FSecure	0.04s	37.57s	0.483s
VirusTotal	0.16s	59.78s	0.572s
EScan	1.08s	22.57s	1.488s
AvastCoreSecurity	0.01s	1.32s	0.039s
ClamAV	0.01s	12.21s	0.087s
ComodoCAVL	1.14s	8.36s	1.342s
McAfeeVSCL	13.73s	33.33s	18.307s
StaticAnalyzer	0.0s	22.19s	0.196s
AVGAntiVirusFree	1.36s	4.39s	2.119s
BitdefenderForUnices	2.91s	36.78s	4.097s
Zoner	0.0s	31.01s	0.078s
VirusBlokAda	2.13s	31.31s	2.81s

Pour pallier la lenteur d'antivirus comme McAfee, dans les 21 *probes*, 4 sont des *probes* McAfee.

6.2 Taux de détection

probename	infected	errors	ratio
FSecure	2357/2465	0	95%
VirusBlokAda	2298/2465	16	93%
EScan	2455/2465	0	99%
AvastCoreSecurity	2456/2465	0	99%
ClamAV	1605/2465	0	65%
ComodoCAVL	2454/2465	0	99%
McAfeeVSCL	2152/2465	0	87%
AVGAntiVirusFree	2373/2465	0	96%
BitdefenderForUnices	2441/2465	0	99%
Zoner	1151/2465	3	46%

L'intérêt d'un outil comme IRMA est aussi de pouvoir évaluer sur un jeu d'exemples donné, les capacités de détection d'un antivirus. Toujours en notant, que le test est réalisé sur une base de malwares publics assez vieux et donc, normalement, assez bien détectés.

6.3 Répartitions des nombres de détections

Nb. détection	Nb. fichiers
0	1
2	5
3	2
4	4
5	14
6	103
7	178
8	503
9	798
10	857

Enfin on peut aussi, par exemple, définir des seuils de faux positifs, faux négatifs en observant la répartition des malwares ayant été détectés par un nombre donné d'antivirus. En donnant à ceux qui ont détecté seul un malware, meilleure réputation que ceux qui se contentent de détecter les fichiers qui le sont aussi par tous les autres.

D'autres retours d'expériences seront abordés lors de la conférence et seront accessibles dans la présentation associée.

7 Conclusion

IRMA est un framework open-source d'analyse de malwares, facilement extensible et adaptable. S'il ne constitue pas la réponse ultime face aux menaces, il en constitue définitivement une brique indispensable autant pour une entreprise désireuse de renforcer la sécurité de son système, que pour un analyste de malware.