REbus : un bus de communication facilitant la coopération entre outils d'analyse de sécurité

Philippe Biondi, Xavier Mehrenberger et Sarah Zennou
philippe.biondi@airbus.com
xavier.mehrenberger@airbus.com
sarah.zennou@airbus.com

Airbus Group Innovations

Résumé L'analyse de sécurité (forensics, pentest, binaires, etc.) nécessite l'utilisation de nombreux outils. Leurs interfaces sont disparates, ce qui rend l'automatisation des analyses et la reproduction de leurs résultats difficile.

Nous présentons ici REbus, un bus de communication libre facilitant la coopération entre outils d'analyse de sécurité.

1 Motivation

1.1 Coopération entre programmes d'analyse

En sécurité informatique, une multitude de programmes existent pour assister l'humain dans la résolution d'un problème. Ces programmes étant très spécialisés, il est souvent nécessaire d'en utiliser plusieurs pour un besoin donné.

Le besoin de les faire travailler de concert s'est d'abord fait ressentir sur nos propres outils. Par exemple, nous utilisons les outils suivants lors du *reverse engineering* de binaires :

- elfesteem [3] pour la manipulation de fichiers ELF et PE ;
- miasm [8] pour de l'assemblage/désassemblage/émulation/JIT/exécution symbolique supportant plusieurs architectures, etc. ;
- amoco [2] pour de l'assemblage/désassemblage/exécution symbolique supportant plusieurs architectures ;
- grandalf [4], un framework de dessin et navigation de graphes ;
- des analyseurs statiques ;
- des outils de classification de binaires ;
- des unpackers ;

Ce besoin s'est vite étendu à d'autres utilitaires tels que IDA, BinDiff, gdb, strace, PEID, libmagic, ou plate-formes telles que cuckoo, FireEye MAS, IRMA et également à d'autres thématiques que l'analyse de binaires.

Dans le domaine de la découverte de services réseau/pentest, il est en effet intéressant de faire coopérer les outils suivants :

— un *scanner* de port (nmap, scapy, etc.)
— un *banner grabber* (nc, nmap, etc.)
— un outil déterminant des *ciphersuites* & protocoles supportés (openssl, etc.)

On peut également citer d'autres domaines pour lesquels des travaux d'analyse nécessitent l'utilisation conjointe de plusieurs outils : l'analyse forensique de disque dur ; l'analyse de certificats ; l'analyse de code JavaScript.

Ce besoin de croiser les résultats de différents outils pose plusieurs problèmes :

— comment stocker les résultats intermédiaires ? En effet, ceux-ci ont un intérêt, particulièrement lors d'une première analyse « à tâtons » ;
— comment rendre compatibles les différentes interfaces d'entrée et sortie (stdin/stdout, écriture de fichiers sur le disque, appels REST, etc.) ;
— même question pour les formats des entrées/sorties ?

De plus, certaines tâches sont répétitives et gagneraient à être entièrement automatisées.

1.2 Vue générale de REbus

Nous présentons ici une réponse à ces problèmes, sous la forme d'un bus de communication, nommé REbus. Ce bus est utilisé par les programmes à faire collaborer. Ces programmes seront dénommés *agents* dans ce qui suit et la communication entre ces agents est appelée *message*. Les objectifs principaux de REbus sont décrits ci-dessous.

La reproduction et l'enregistrement des analyses. REbus facilite les expérimentations et le développement de nouvelles techniques d'analyse : les expériences peuvent être rejouées et les résultats sont consignés pour pouvoir être étudiés.

L'intégration facile de nouveaux agents. REbus fournit un ensemble de services pour ajouter un agent sans avoir à intervenir sur le bus lui-même ni sur les agents déjà présents, et ce *même si ce nouvel outil doit interagir avec eux.*

Un fonctionnement décentralisé. REbus privilégie l'exhaustivité de l'analyse et la simplicité des interfaces à l'efficience du traitement : chaque agent prend indépendamment la décision de travailler ou non sur les messages qui lui sont envoyés.

Trois modes d'utilisation. À partir du même ensemble d'agents, REbus peut être utilisé de trois façons :

- en mode *outil* afin de créer un nouveau « super-outil » rendant un service spécialisé. Par exemple, pour du *reverse engineering*, en combinant un agent qui unpacke un binaire donné en entrée, un autre qui extrait le CFG (*Control Flow Graph* – graphe de flot de contrôle) du binaire obtenu, et un troisième qui l'affiche ;
- en mode *infrastructure*, mettant à disposition les services des agents sur une durée indéfinie,
- en mode *feu de camp*, où plusieurs analystes font tourner des agents sur leur machine, et les connectent à un bus partagé le temps d'un projet (rétro-ingénierie d'une application complexe, caractérisation d'un ensemble de *malware*, etc.).

Sécurité. REbus n'intègre pas d'authentification, de contrôle d'accès, ni de vérification d'intégrité des messages échangés. L'exploitation d'éventuelles vulnérabilités dans les outils d'analyse utilisés (ex. des vulnérabilités ont déjà existé dans `strings` [12], antivirus, etc.) est possible. Il est recommandé de manipuler des données potentiellement dangereuses (ex. *malware*, fichiers produits par un tiers non de confiance) dans des environnements isolés (machines virtuelles, docker, conteneur LXC, seccomp, etc.). L'utilisation automatique de mécanismes d'isolation lors du lancement d'outils externe est une évolution prévue de REbus.

Le reste de ce document est structuré de la façon suivante : la section 2 décrit REbus ; la section 3 fournit des exemples d'utilisation de REbus et la section 4 conclut cet article avec une comparaison de REbus à d'autres solutions, un exemple de cas d'utilisation réelle, et donne des perspectives de développement de REbus.

2 Description de REbus

2.1 Vocabulaire

Un outil d'analyse est intégré dans le bus de communication par le développement d'un court programme Python nommé *agent*, qui se charge

de piloter l'outil, de lui fournir ses entrées venant du bus et de placer ses sorties dans le bus.

Les entrées/sorties des outils sont transportés dans des messages qui sont des objets Python appelés *descripteurs*.

Un composant de stockage, également en Python, enregistre tous les *descripteurs*.

Un composant appelé *bus master* est responsable de la circulation des messages entre les agents et également à destination du composant de stockage.

Les agents peuvent effectuer des requêtes vers le composant de stockage à travers le *bus master*.

Ces éléments sont résumés à la figure 1.

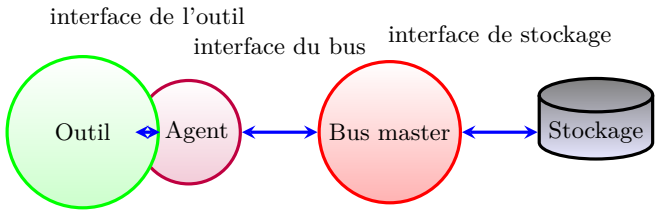


Figure 1. Interfaces entre les composants de REbus

2.2 Descripteurs

Un descripteur encapsule une valeur, représentée par une chaine de caractères et quelques métadonnées associées.

Les descripteurs sont produits par des agents, à partir d'une donnée exogène (entrée utilisateur, fichier, etc.) ou à partir des données contenues dans un ou plusieurs descripteurs déjà présents dans le bus, appelés les *parents* du nouveau descripteur.

Les descripteurs sont des objets Python comportant les propriétés suivantes :

selector chaine de caractères qui remplit plusieurs fonctions :
- décrire le type de la données encapsulée ($\simeq$ type MIME) : image, fichier binaire, etc.
- décrire le format de la donnée (ex. JPG ou PNG pour une image, PE ou ELF pour un binaire, etc.),

  - identifier de manière unique un descripteur par un *hash* (SHA256) dépendant :
    - de la valeur stockée,
    - du nom de l'agent ayant généré le descripteur,
    - du sélecteur des éventuels descripteurs parents,
    - du début de la chaine du sélecteur (tout sauf le *hash*).

label nom compréhensible par un humain.

uuid identifiant unique regroupant les descripteurs liés à un même objet analysé.

value valeur du descripteur – contenu du message transporté.

precursors liste de sélecteurs des descripteurs parents.

agent name nom de l'agent ayant généré ce descripteurs.

domain permet de séparer deux analyses sans rapport dans un même bus.

version numéro de version.

processing time temps ayant été nécessaire à la génération de ce descripteur.

Les descripteurs comportent également des méthodes qui permettent de :
- générer un nouveau descripteur, lié à la même analyse (même UUID) ou non ;
- générer une nouvelle version d'un descripteur ;
- créer un lien avec un autre descripteur, en précisant la raison ;
- (dé)sérialiser.

À propos des sélecteurs. Le sélecteur de chaque nouveau descripteur est envoyé à chaque agent, qui l'utilise pour décider s'il souhaite traiter ce descripteur (ex. un agent calculant le *hash* MD5 peut être intéressé par les binaires au format PE), l'ignorer, ou demander plus d'informations au bus pour déterminer s'il souhaite effectuer le traitement.

Un sélecteur peut avoir la forme suivante : `/signature/md5/%6e1d5169661a50(...)f989129a583f92b9dee`. Le préfixe `/signature/md5` décrit le type de données ; le format de la valeur du descripteur n'est pas indiqué ici, et dépendra de l'agent l'ayant généré. Il pourra s'agir d'une valeur brute (128 bits) ou de sa représentation hexadécimale (32 caractères). Dans le jeu d'agents public `rebus_demo` [6], la seconde option a été choisie.

2.3 Agents

Les agents sont des programmes Python réalisant l'interface entre un outil (module Python, programme externe, etc.) et REbus.

Le bus master notifie tous les agents lorsqu'un nouveau descripteur est ajouté au bus. Chaque agent décide alors s'il souhaite traiter le nouveau descripteur, par la surcharge (facultative) des méthodes suivantes de la classe `Agent` :

— `selector_filter` qui permet un premier niveau de décision connaissant uniquement le sélecteur du nouveau descripteur ;
— `descriptor_filter` qui permet le filtrage en disposant du descripteur complet.

Lorsque ces deux méthodes renvoient `True` (valeur par défaut en l'absence de surcharge), la méthode `process` de l'agent est appelée. Le traitement peut alors commencer.

Voici à titre d'exemple le code d'un agent très simple calculant le *hash* MD5 des descripteurs dont le sélecteur commence par `/binary/`. À l'issue du traitement, l'agent crée un descripteur fils ayant pour sélecteur `/md5_hash/` et pousse ce descripteur ainsi que le *hash* dans le bus.

```
from rebus.agent import Agent
import hashlib


@Agent.register
class Hasher(Agent):
    _name_ = "hasher"
    _desc_ = "Return md5 of a binary"

    def selector_filter(self, selector):
        # Indicate that this agent is only interested in descriptors
        # whose selector starts with "/binary"
        return selector.startswith("/binary/")

    def process(self, desc, sender_id):
        # Compute md5 hash value
        md5_hash = hashlib.md5(s).hexdigest(desc.value)

        # Create a new child descriptor
        new_desc = desc.spawn_descriptor("/md5_hash",
                                         unicode(md5_hash),
                                         self.name)

        # Push the new descriptor to the bus
        self.push(new_desc)
```

Listing 1. Agent REbus calculant le hash MD5 de fichiers binaires

Lorsque plusieurs instances d'un même agent s'exécutent simultanément, le premier agent commençant le traitement d'un nouveau descripteur empêchera automatiquement les autres instances de le faire.

De plus, les agents peuvent surcharger des méthodes `get_internal_state` et `set_internal_state` exécutées lors de l'arrêt du bus et de son redémarrage, pour sauvegarder et restaurer leur état interne.

Enfin, deux modes de fonctionnement peuvent être supportés par les agents :

— le mode automatique, dans lequel tous les descripteurs acceptés par les méthodes de filtrage décrites plus haut sont traités ;
— le mode interactif, dans lequel l'agent indique les descripteurs qu'il est capable de traiter, le traitement se faisant sur demande de l'utilisateur.

Le choix de l'un des deux modes est fait au lancement de l'agent.

De plus, lors du lancement de cet agent, le bus master lui communique les sélecteurs de tous les descripteurs déjà présents sur le bus qui n'ont pas déjà été traités par une autre instance du même agent.

2.4 Stockage

Tous les descripteurs envoyés sur le bus sont enregistrés par le système de stockage. Ce système possède une API offrant les fonctionnalités suivantes :

— recherche de descripteur par UUID, par *regex* sur le sélecteur ou la valeur ;
— obtention de liste des analyses (UUID) existantes ;
— enregistrement et restauration de l'état interne des agents (utile lors de l'arrêt/redémarrage du bus) ;
— suivi du traitement de chaque descripteur par chaque agent et en particulier fourniture de la liste des descripteurs non traités.

Deux implémentations de cette API sont disponibles :

RAMStorage stocke toutes les données en RAM, qui seront donc perdues lors de l'arrêt du bus ;

DiskStorage enregistre les descripteurs et les états internes des agents sur le disque, ce qui permet l'arrêt puis la reprise des analyses depuis une configuration donnée.

L'écriture d'une implémentation de cette API effectuant un stockage distribué sur plusieurs machines est prévue.

2.5 API du bus de communication

Le bus expose une API qui rend les services suivants :
— permettre aux agents de diffuser des descripteurs à tous les autres agents ;
— permettre aux agents de recevoir les nouveaux descripteurs disponibles ;
— enregistrer l'historique de traitement des descripteurs pour chaque agent ;
— répartir des descripteurs entre plusieurs instances du même agent (ex. : agents effectuant des traitements longs) ;
— enregistrer des méthodes pouvant être appelées par d'autres agents pour post-traitements ;
— sauvegarder et restaurer l'état interne des agents lors de l'arrêt ou la reprise du bus ;
— transporter les requêtes vers le système de stockage de descripteurs.

Deux implémentations de cette API sont disponibles : *LocalBus* et *DBusBus*.

LocalBus Le bus LocalBus permet de combiner plusieurs outils pour créer un nouveau "super-outil" en ligne de commande. Tous les agents et le *Bus Master* s'exécutent au sein du même processus. Le bus s'arrête lorsque tous les traitements sont terminés.

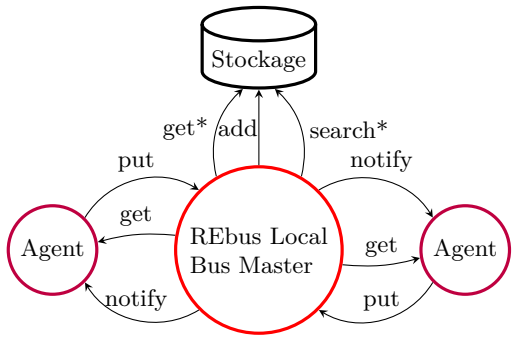


Figure 2. Composants du bus LocalBus

REbus au dessus de D-Bus Le bus utilisant D-Bus permet une utilisation plus interactive du bus :

— chaque agent s'exécute dans un processus séparé ;
— les agents s'arrêtent lorsque le *Bus Master* est arrêté par l'utilisateur ;
— les agents peuvent être exécutés sur des hôtes distants.

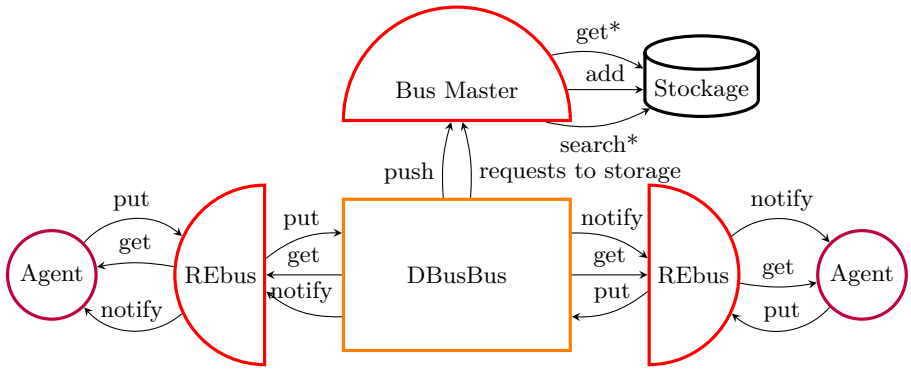


Figure 3. Composants du bus DBusBus

2.6 Agents disponibles

L'ajout de capacités d'analyse au bus se fait à travers l'ajout d'un ensemble d'agents portant sur une thématique données (analyse de *malware*, reconnaissance réseau, analyse forensique d'images disque, etc.) contenus dans un package Python, dont le nom sera passé en paramètre lors du lancement de REbus. Ce découplage permet le développement d'agents indépendamment du développement de REbus. Il est possible d'utiliser simultanément des agents appartenant à des ensembles différents. La liste des ensembles d'agents publics est consultable sur [7].

Agents génériques fournis avec REbus Plusieurs agents sont fournis avec le bus. Ils sont génériques, et n'effectuent pas de traitement spécifique à un type de problème à résoudre. Ils servent principalement à interagir avec le bus comme listé ci-dessous et résumé à la figure 3 :

inject : injecte des fichiers dans le bus.

unarchive : extrait les archives et fichiers compressés (zip, tgz, tbz2, etc.), injecte leur contenu sur le bus.

ls : liste les sélecteurs connus correspondant à la *regex* fournie.

return : renvoie sur *stdout* la valeur des descripteurs dont le *sélecteur* correspond à la *regex* fournie.

search : renvoie sur *stdout* le sélecteur des descripteurs dont la valeur correspond à la *regex* fournie.

httplistener : lance un serveur HTTP et injecte sur le bus tout contenu transmis par méthode POST.

link_finder : crée automatiquement des liens (descripteurs dont le sélecteur commence par `/link/`) entre les descripteurs ayant la même valeur.

link_grapher : crée des graphes montrant les liens existants entre descripteurs.

dotrenderer : effectue le rendu de graphes au format dot vers SVG.

web_interface : interface web permettant l'interaction avec le bus.

Agents de démonstration Un ensemble d'agents de démonstration est public [6]. Il comporte les agents suivants :

hasher : calcul le *hash* MD5 de binaires

stringer : renvoie la sortie de `strings` exécuté sur un binaire

grep : renvoie sur *stdout* la valeur des descripteurs de type `/string/` correspondant à la *regex* fournie

3 Exemples d'utilisation

3.1 Cinématique de fonctionnement

Voici un exemple à but pédagogique de « super-outil » simple s'appuyant sur plusieurs outils pour afficher sur la sortie standard le *hash* MD5 des fichiers contenus dans une archive tgz.

```
$ rebus_agent -m rebus_demo.agents hasher unarchive \
                                   inject ~/apt1.tgz -- \
                                   return --short md5_hash
```

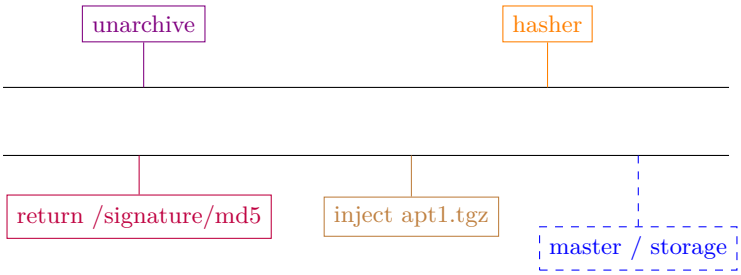
Listing 2. Lancement de REbus et agents pour calcul de MD5

Cette ligne de commandes comporte les éléments suivants :

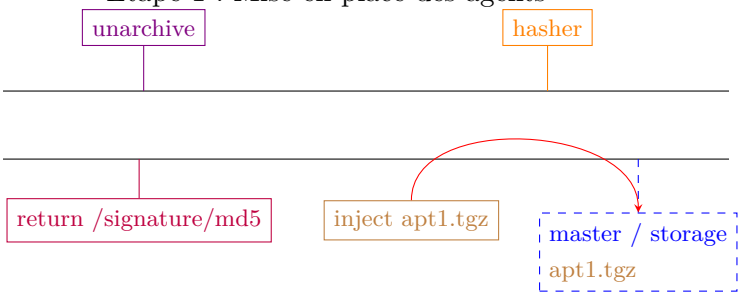
— importe les agents présents dans le package python `rebus_demo.agents` ;

— le lancement de l'agent `inject` comportant un argument ; les tirets marquent la fin de la liste des arguments de cet agent, dont le nombre n'est pas fixé (il est possible d'injecter plusieurs fichiers avec le même agent `inject`) ;
— le lancement de l'agent `return` affichant la valeur des *hash* MD5 calculés sur la sortie standard ;
— le lancement de plusieurs agents sans argument.

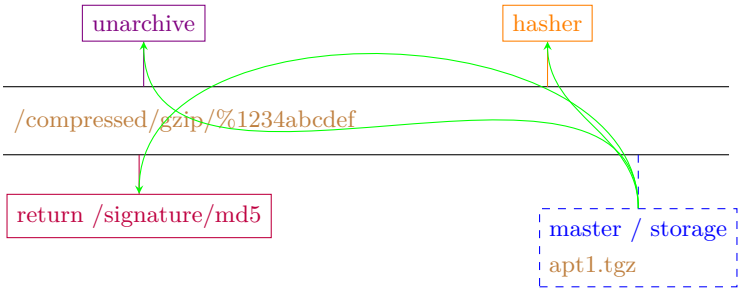
L'exécution de cette commande peut se décomposer en dix étapes comme suit :



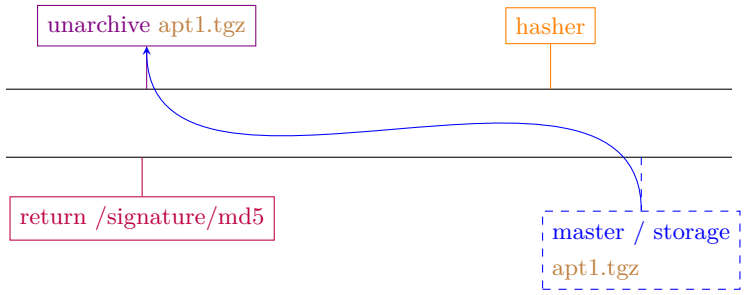
Étape 1 : Mise en place des agents



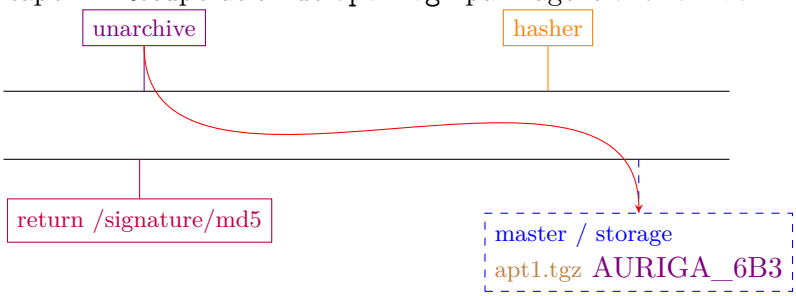
Étape 2 : Envoi de apt1.tgz au Bus Master par l'agent `inject`



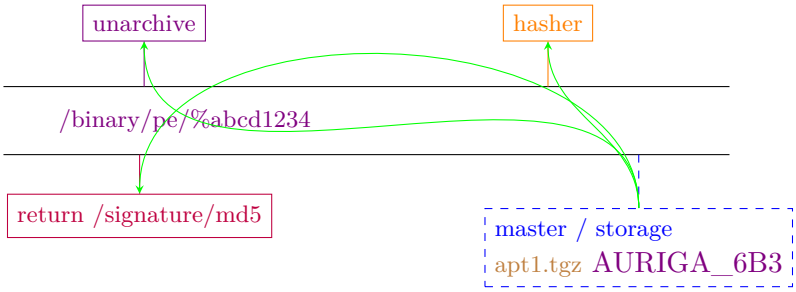
Étape 3 : Annonce du sélecteur du nouveau descripteur par le Bus Master



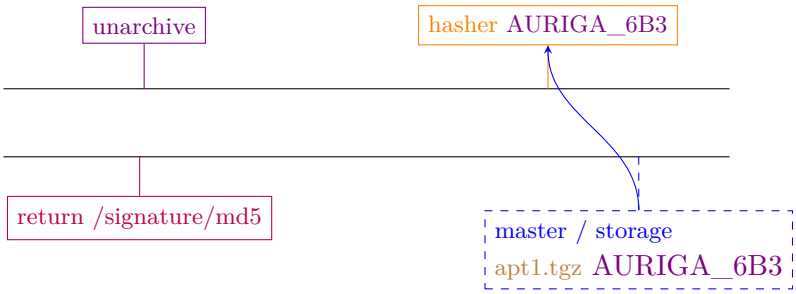
Étape 4 : Récupération de `apt1.tgz` par l'agent `unarchive`



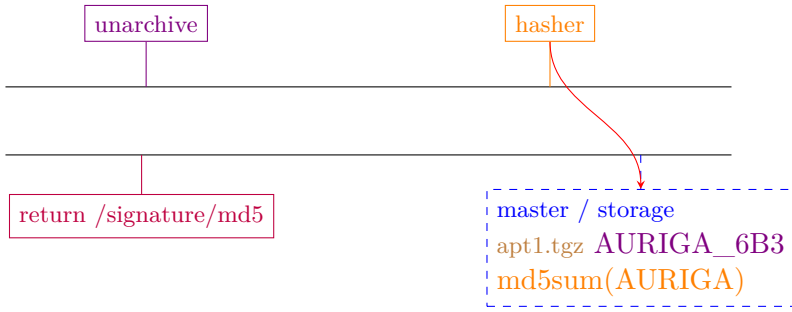
Étape 5 : Envoi du fichier extrait `AURIGA...` au Bus Master par l'agent `unarchive`



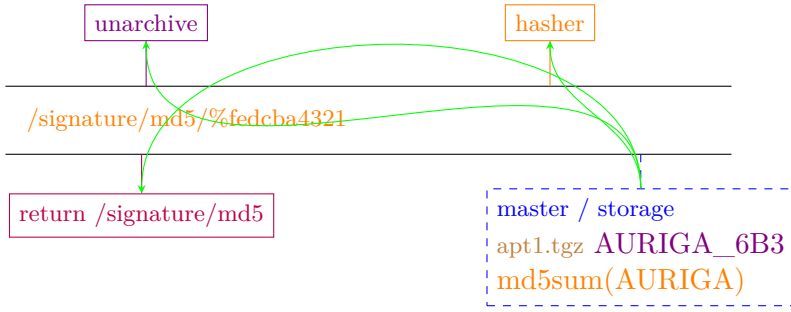
Étape 6 : Annonce du sélecteur du nouveau descripteur par le Bus Master



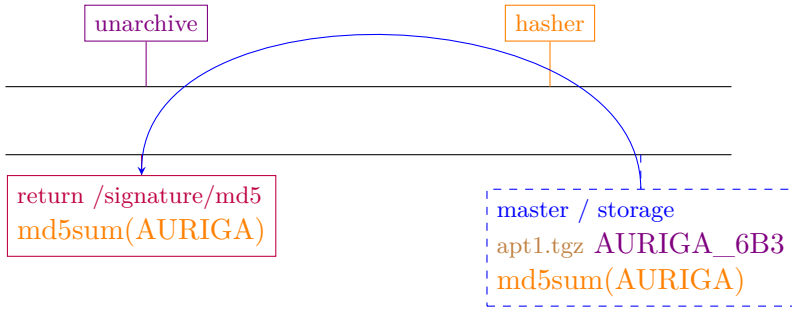
Étape 7 : Récupération de `AURIGA...` par l'agent `hasher`



Étape 8 : Envoi du *hash* MD5 du fichier `AURIGA...` par l'agent `hasher`



Étape 9 : Annonce du sélecteur du nouveau descripteur par le Bus Master



Étape 10 : Récupération du *hash* MD5 par l'agent `return` et affichage sur la sortie standard

Enfin, l'agent `return` affiche son résultat comme illustré sur le listing 3.

```
apt1.tgz:AURIGA_6B31344B40E2AF9C9EE3BA707558C14E =
    6b31344b40e2af9c9ee3ba707558c14e
apt1.tgz:AURIGA_CDCD3A09EE99CFF9A58EFEA5CCBE2BED =
    cdcd3a09ee99cff9a58efea5ccbe2bed
apt1.tgz:BANGAT_468FF2C12CFFC7E5B2FE0EE6BB3B239E =
    468ff2c12cffc7e5b2fe0ee6bb3b239e
```

```
...
```

Listing 3. Sortie de l'agent `return`

Seul le calcul du premier *hash* est représenté dans les schémas ci-dessus ; le traitement continue de manière similaire pour le traitement des fichiers suivants.

Les préfixes de *hashs* représentés sur les schémas après le symbole % sont ceux des descripteurs.

3.2 Génération d'un graphe reliant les binaires ayant le même importhash

Voici un autre exemple, utilisable en pratique, d'un autre « super-outil » générant un graphe reliant les binaires ayant la même valeur d'*import hash* [9]. Il s'appuie sur les agents suivants :

inject injecte les binaires sur le bus ;

file_identification calcule la valeur de l'*import hash* ;

link_finder déclare des liens entre descripteurs ayant la même valeur (ici, `/signature/imphash/`) ;

link_grapher crée un graphe montrant tous les descripteurs reliés par un lien de type `/link/link_finder/signature-imphash`. Il a la particularité d'effectuer une requête pour obtenir tous les liens correspondant aux critères donnés ayant déjà été injectés dans le bus ;

dotrenderer effectue le rendu de graphes en image SVG ;

return affiche sur *stdout* l'image SVG produite.

La ligne de commande à exécuter est la suivante :

```
$ rebus_agent -m bnew.agents link_finder -- file_identification
    inject * --\
    :: link_grapher '/link/link_finder/signature-imphash' --\
    dotrenderer return '/graph/svg' --raw> ~/links-apt1.svg
```

Listing 4. Lancement de REbus et agents pour génération du graphe

On notera sur cette ligne de commande l'utilisation du symbole `::` séparant des étapes de traitement : les agents décrits avant ce symbole s'exécutent d'abord ; une fois que tous les descripteurs ont été traités par tous les agents, ceux-ci s'arrêtent. Les agents décrits après ce symbole s'exécutent ensuite.

On obtient les résultats présentés dans le listing 4 pour un corpus de *malware* APT1. Les nœuds rectangulaires du graphe correspondent à des *malware* du corpus APT1 [11] ; ils sont reliés à des nœuds ronds, symbolisant une valeur d'*import hash*. On peut voir que ce nouveau « super-outil » permet de relier entre eux tous les *malware* ayant la même valeur d'*import hash*. Le texte de ces nœuds, illisible ici, n'a que peu d'importance.

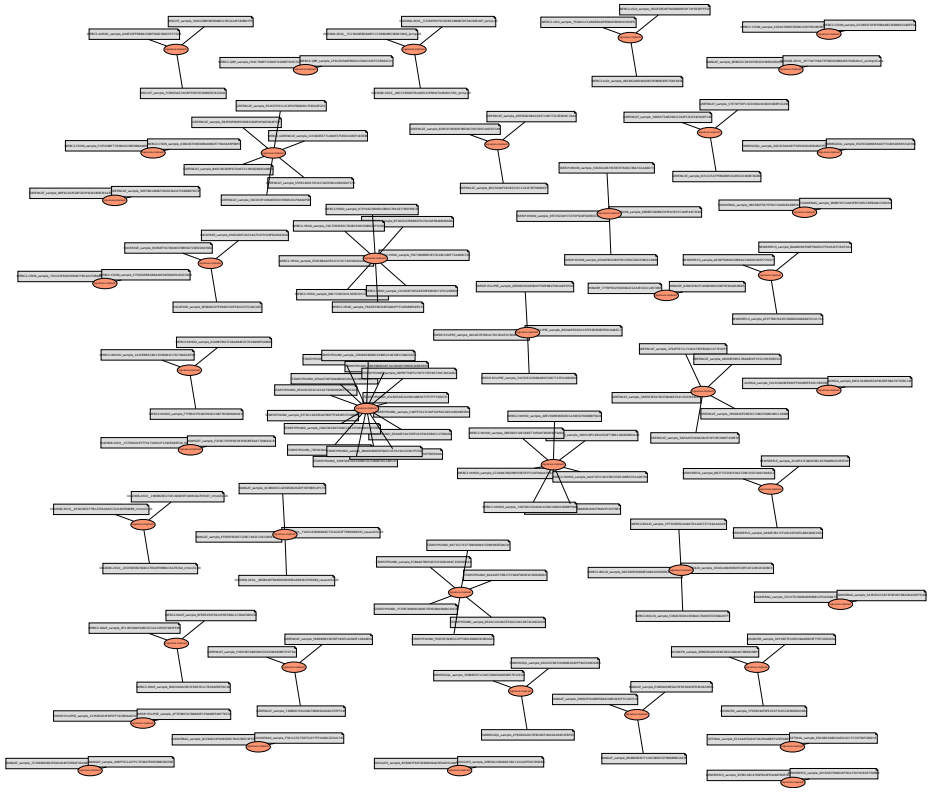


Figure 4. Lien entre *malware* APT1 ayant la même valeur d'import hash

3.3 Utilisation en mode infrastructure

Les deux exemples précédents utilisent le bus LocalBus. Voici un exemple d'utilisation du mode infrastructure, basé sur D-Bus, en utilisant les agents de l'ensemble `rebus_demo` [6].

Pour utiliser REbus en mode infrastructure, il faut commencer par lancer le *bus master* :

```
$ rebus_master_dbus --help
usage: rebus_master_dbus [-h] [-f LOGFILE] [-v LEVEL]
                         {diskstorage,ramstorage} ...

Rebus DBUS master

positional arguments:
  {diskstorage,ramstorage}
                        Storage backends

optional arguments:
  -h, --help            show this help message and exit
  -f LOGFILE, --logfile LOGFILE
                        Destination log file
  -v LEVEL, --verbose LEVEL
                        Verbosity, 0 is most verbose, 50 least

Uses the ramstorage backend if none is specified.

$ rebus_master_dbus diskstorage --help
usage: rebus_master_dbus diskstorage [-h] [--path PATH]

optional arguments:
  -h, --help   show this help message and exit
  --path PATH  Disk storage path (defaults to /tmp/rebus)

# lancement du bus master
$ rebus_master_dbus
```

Listing 5. Aide de rebus_master_dbus et lancement du bus master

Il faut ensuite lancer les agents que l'on souhaite utiliser. Les agents utilisés ici ont été présentés plus haut.

```
for agent in web_interface unarchive link_finder hasher stringer
  do rebus_agent -m rebus_demo.agents --bus dbus --daemon $agent
done
```

Listing 6. Lancement des agents

L'interface web est alors accessible à l'adresse `http://localhost:8080`.

On peut utiliser l'agent `inject` pour injecter le contenu d'un fichier sur le bus. Cette opération peut être effectuée avant ou après le lancement des agents.

```
$ rebus_agent --bus dbus inject /tmp/foo.tgz
```

Listing 7. Injection de foo.tgz

```
$ rebus_agent --bus dbus -m rebus_demo.agents grep 12345
foo.tgz:cpio = 0123456789ABCDEF
foo.tgz:bash = 0123456789abcdef
foo.tgz:bash = 0123456789ABCDEF
```

Listing 8. Recherche de la chaine 12345 parmi les strings injectées

4 Conclusion

Cet article a présenté REbus un bus de communication facilitant la coopération entre outils d'analyse de sécurité.

Le bus, ainsi que certaines analyses – implémentées via le mécanisme des *agents* – sont disponibles respectivement ici [5] et ici [6], sous licence BSD.

Le reste de cette section section est consacrée à l'état de l'art de ce type de problématique (section 4.1) ; à des considérations techniques qui montrent notamment la robustesse de notre solution (section 4.2) et enfin des perspectives d'évolution de REbus (section 4.3).

4.1 Comparaison avec des solutions existantes

Il est possible d'envisager d'autres façons de faire coopérer des outils d'analyse de sécurité.

Il est possible de fournir des bibliothèques enveloppant les outils et offrant une API cohérente ou unifiée. L'ajout de nouveaux outils nécessitent d'étendre ou d'ajouter une bibliothèque. Mais il reste encore à la charge du programmeur de les appeler, de s'occuper de la nouvelle circulation des messages entre les différents outils et de l'intendance.

L'approche *framework* dispense tout au moins de l'intendance. La circulation des messages peut être prise en charge par exemple via l'utilisation d'un mécanisme de transport « sans intelligence », n'ayant pas connaissance des interactions entre les agents :

— systèmes de fichiers (ex. ZeroMQ, RabbitMQ) ;
— RPC (ex. D-Bus) ;
— microservice REST pour chaque outil.

Il est également possible de concentrer toute la logique d'interaction dans un composant central, qui s'occupera d'appeler chaque outil lorsque cela est nécessaire :

— « orchestrateur » interfacé avec chaque outil ;
— framework interactif (ex. `viper` [1]).

L'utilisation directe de chacun de ces choix implique la connaissance *a priori* des agents existants, des types de données échangés ou des graphes de flots de données à respecter.

REbus peut utiliser l'un ou l'autre de ces mécanismes de transport mais y ajoute une couche d'abstraction permettant de définir la circulation des messages de manière implicite et *a posteriori*.

À chaque agent est laissé le choix de traiter ou non un message, via un filtrage effectué dans un premier niveau sur le type du message (sur son sélecteur, ou sur toute caractéristique du message dans un second temps). Ainsi, chaque agent a un unique interlocuteur (le bus) et n'a pas besoin de connaitre ni de prendre en compte les différents outils avec lesquels il va coopérer.

L'ajout d'un nouvel outil à un écosystème existant est aisé : il suffit d'écrire un agent se chargeant de récupérer les données traitables par l'outil depuis le bus, au format défini par les agents déjà existants produisant des données de ce type, puis de pousser les résultats obtenus au format attendu par les agents existants consommant des données de ce type, sans se soucier de savoir quels seronts les agents lancés produisant ou consommant des types de données.

4.2 Robustesse de REbus

REbus a été principalement utilisé à des fins d'analyse et de classification de *malware*. L'objectif est d'évaluer la dangerosité d'un grand nombre de programmes injectés dans REbus, et de les regrouper par familles.

L'avantage principal de REbus pour ce type de travail est sa modularité : la classification de *malware* repose sur les étapes suivantes :

— extraction de caractéristiques (par exemple, la liste des imports) ;
— calcul de distances entre ces *malware* en utilisant les caractéristiques extraites ;
— utilisation d'algorithmes d'apprentissage et de classification.

REbus facilite l'expérimentation : la modification d'une des étapes de traitement (ex. extraction de nouvelles caractéristiques) ne nécessite pas de modification sur les autres étapes.

Notre participation à un concours de classification de *malware* [10] a montré la robustesse de l'implémentation de REbus à l'injection de 21 757 listings assembleur.

Cependant, les fichiers ayant une taille supérieure à 150 Mo ont dû être pré-traités pour en réduire la taille en supprimant les informations inutiles : les communications entre agents et bus master via D-Bus échouaient à

cause d'expiration de délai D-Bus. Des travaux sont en cours pour résoudre ce problème, *cf* les perspectives.

4.3 Perspectives

Les améliorations suivantes au bus sont à l'étude :

— l'injection de fichiers de grande taille pose problème. Les évolutions envisagées sont : une nouvelle implémentation de l'API *Bus* le passage des valeurs des descripteurs par référence ;
— la gestion des dépendances de descripteurs : actuellement il n'est pas possible de déclencher un traitement lorsque plusieurs descripteurs liés à une analyse sont présents. Par exemple, extraire les chaines de caractères lisibles (*strings*) d'un binaire uniquement si l'agent de détection de *packer* a déterminé qu'il n'était pas packé ;
— l'ajout de méthodes permettant l'exécution de programmes tiers dans un environnement isolé, par exemple par l'utilisation de namespaces linux ou de seccomp ;
— implémentation d'un nouveau *Bus*, basé sur une technologie plus robuste que D-Bus (peut-être zeromq, services REST ou MPI) ;
— support de la transmission de valeurs par référence, pour faciliter l'analyse de fichiers de grande taille ;
— indexation et recherche intégrée à l'interface web.

Références

1. Airbus Group Innovations. Viper – binary management and analysis framework. `http://viper.li/`, 2014.
2. Airbus Group Innovations. Dépôt de code amoco. `https://github.com/bdcht/amoco`, 2015.
3. Airbus Group Innovations. Dépôt de code elfesteem. `https://bitbucket.org/LouisG/elfesteem`, 2015.
4. Airbus Group Innovations. Dépôt de code grandalf. `https://github.com/bdcht/grandalf`, 2015.
5. Airbus Group Innovations. Dépôt de code rebus. `https://bitbucket.org/iwseclabs/rebus`, 2015.
6. Airbus Group Innovations. Dépôt de code rebus_demo. `https://bitbucket.org/iwseclabs/rebus_demo`, 2015.
7. Airbus Group Innovations. List of public REbus agent sets. `https://bitbucket.org/iwseclabs/rebus/wiki/Sets%20of%20rebus%20agents`, 2015.
8. CEA IT Security. Dépôt de code miasm. `https://github.com/cea-sec/miasm`, 2015.

9. Mandiant. Tracking Malware with Import Hashing. https://www.mandiant.com/blog/tracking-malware-import-hashing/, 2014.
10. Microsoft Kaggle. Microsoft Malware Classification Challenge (BIG 2015). www.kaggle.com/c/malware-classification, 2015.
11. Parkour, Mila. Mandiant APT1 samples categorized by malware families. http://contagiodump.blogspot.fr/2013/03/mandiant-apt1-samples-categorized-by.html, 2013.
12. Zalewski, Michał. PSA : don't run 'strings' on untrusted files (CVE-2014-8485). http://lcamtuf.blogspot.fr/2014/10/psa-dont-run-strings-on-untrusted-files.html, 2014.