



11 FEVRIER 2020

RAPPORT PLATEAU PROJET

DETECTION DE MALWARE PE PAR MACHINE LEARNING

MAHAMAN BACHIR ATTOUMAN OUSMANE
UNIVERSITÉ POLYTECHNIQUE HAUTS DE FRANCE
Master 2 CDSI Valenciennes

Table des matières

Introduction.....	3
1. Le format PE	3
2. Dataset (Jeu de données).....	4
3. Extraction de caractéristiques (feature).....	4
3.1. Attributs headers.....	5
3.2. Histogramme sur la représentation hexadécimale	5
3.3. Libraires importées.....	5
3.4. Extraction sur les fichiers et concaténation	6
3.5. Désassemblage.....	7
4. Création de la Dataframe pandas.....	8
5. Entraînement du modèle	9
5.1 Description et réduction.....	9
5.2. Création du modèle.....	9
5.3. Résultats	11
5.4. Enregistrement du modèle.....	11
6. Utilisation du modèle	12
7. Analyse dynamique	12
7.1. Cuckoo	12
7.2. Analyse des Faux positifs.....	13
8. Limites de la détection par machine learning en analyse statique.....	14
Conclusion	15
Références.....	16

Tables des figures

Figure 1. Format Portable exécutable [1].....	3
Figure 2 librairie lief.....	4
Figure 3. Sections	5
Figure 4.Histogramme des octets	5
Figure 5. Nombre de fonctions importées	6
Figure 6. Concaténation des vecteurs	6
Figure 7.Désassemblage	7
Figure 8. Rassembler toutes les features en une ligne	8
Figure 9. Nom des colonnes	8
Figure 10. Exemple : features d'un exécutable	9
Figure 11. Description dataset	9
Figure 12.Faux positifs.....	11
Figure 13 Démonstration	12
Figure 14. Fichiers de configuration Cuckoo	13
Figure 15. Fichiers de configuration cuckoo.....	13
Figure 16. Résultats Analyse cuckoo	14
Figure 17. Résultats Any.run	14
Figure 18. Modification avec lief	15

Introduction

Ce document présente le rapport de projet dans le cadre du module « plateau projet ». Il est question de détection de malware par machine learning. Nous nous focalisons uniquement sur le format PE windows. Dans un premier temps nous présenterons brièvement le format portable exécutable, puis un état de l'art de la matière (ML et détection de malware), ensuite nous ferons une extraction de caractéristique statiques, entraînerons des algorithmes de machine learning à détecter la nocivité d'un exécutable. Nous installerons et utiliserons un environnement d'analyse dynamique et enfin nous verrons une méthode de contournement d'algorithme de détection par machine learning.

1. Le format PE

C'est le format utilisé par les systèmes Windows et dérive du format COFF. Il est utilisé pour les DLL, les applications et les pilotes. Sa structure générale est la suivante.

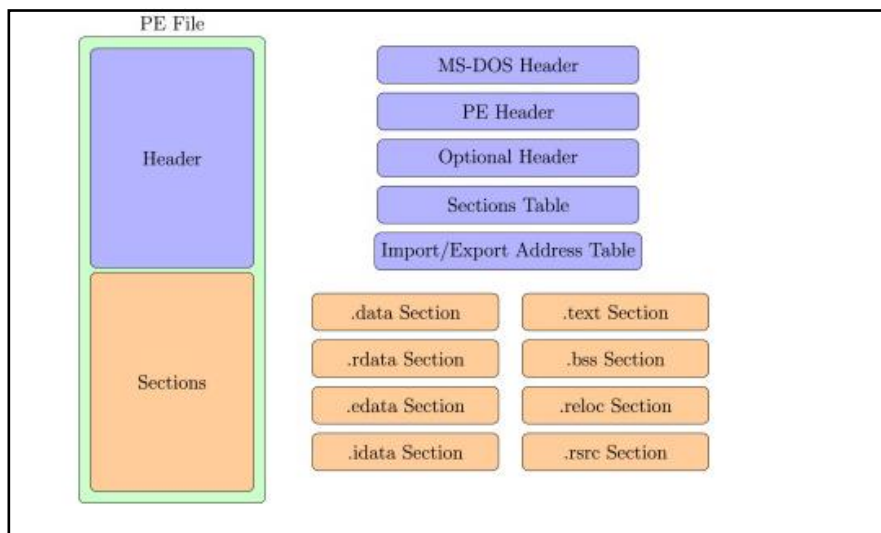


Figure 1. Format Portable exécutable [1]

Nous allons brièvement présenter les entêtes sans être exhaustifs :

- **DOS Header** : Elle contient un message qui indique que le fichier ne peut être exécuté sous MS-DOS.
- **PE header** : En-tête contenant la signature du PE (PE\0\0) et l'image file header informations pour l'exécution en mémoire comme le type de machine qui a compilé le code, le nombre de sections, le pointeur vers les données entre autres
- Ensuite nous avons des données qui ne sont pas du tout facultatives comme le nom le ferait penser (**Optional Header**). Au total 31 champs parmi lesquels :
 - Le magic number (10b pour les 32 bits et 20b pour les 64 bits)
 - Des informations sur l'éditeur de liens, la taille des données et du code
 - Le point d'entrée, les adresses relatives des différentes sections
 - L'adresse de base (adresse de montage en mémoire), toutes les adresses relatives le sont par rapport à elle.

- Des informations sur l'alignement des sections, les caractéristiques des DLL, les versions de système d'exploitation, taille stack et Heap, le checksum etc. ... voir [2]

- **La table des sections** : De taille variable, le nombre des sections est donné par le PE header. L'entête contient des informations sur la mémoire à réserver à une section (lors du chargement), son offset, sa taille sur le disque et un pointeur sur les données. Windows autorise au maximum 96 sections.

- **La table d'importation** : Sur le disque elle contient le nom des fonctions API à importer d'une DLL. Le PE loader convertit ce nom en adresse puis charge la fonction dans l'espace mémoire du processus.

Les sections contiennent les variables non initialisées (.bss), les variables initialisées (.data) ou encore le code assembleur : .text de l'exécutable entre autres. Les noms diffèrent en fonction des compilateurs et sont de nombre variable. Toutes les informations nécessaires à leur lecture ou exécution se trouvent au niveau des tables.

2. Dataset (Jeu de données)

Nous ferons de l'apprentissage supervisé et pour cela nous utiliserons des fichiers déjà labélisés – c'est-à-dire qui ont été préalablement définis comme malveillants ou non. Pour les malwares nous obtenons un Dataset sur VirusShare [3]. Et pour les fichiers *sains* nous récupérons grâce à un petit script python tous les exécutables PE présents dans un système Windows 10 (en usage). Un scan complet est d'abord effectué sur la machine avec l'antivirus Norton. Les DLLs sont exclues car pouvant entraîner des biais.

3. Extraction de caractéristiques (feature)

Afin de déterminer si un malware est bénin (non malicieux) ou malicieux (maligne), nous devons d'abord choisir des paramètres qui détermineront sa nature. Les *features* statiques ont pour avantage d'être rapides à extraire. Notre environnement de travail est sous Ubuntu 18.04, le code est écrit en python et nous utilisons la librairie ***lief*** développée par quarkslabs pour parser un fichier PE. [4]

Toutes les informations seront mises sous forme de vecteurs contenant des valeurs numériques entre 0 et 1. À cause du format d'entrée de la librairie de machine learning les valeurs seront de type numpy.Float64

```
import lief
exe = lief.PE.parse("/mnt/c/Downloads/vlc.exe")
```

Figure 2 librairie lief

3.1. Attributs headers

Il paraît tout d'abord nécessaire de connaître la présence de certaines sections. La librairie que nous utilisons renvoie sous forme de booléen la présence ou non de certains attributs. Par exemple si le fichier contient une signature ou utilise TLS (thread local Storage) ou encore s'il possède une table d'exception ou non.[5]

```
def get_flags(exe):
    # ex [0,1,1,0,...] indique has not configuration, has debug, has exceptions, has not exports
    pe_properties = ["has_configuration", "has_debug", "has_exceptions",
                    "has_exports", "has_import", "has_nx",
                    "has_relocations", "has_resources",
                    "has_rich_header", "has_signature", "has_symbol",
                    "has_tls"]

    temp = []
    for has in pe_properties:
        if getattr(exe, has):
            temp.append(1.0)
        else:
            temp.append(0.0)
    return (np.array(temp))
```

Figure 3. Sections

3.2. Histogramme sur la représentation hexadécimale

Plusieurs documents de l'état de l'art indiquent que l'entropie permet de détecter de l'obfuscation, un fichier empaqueté ou chiffré (une grande entropie élevée par section). [6] D'autres utilisent des n-grams sur les séquences de bytes du fichier et parviennent ainsi à faire ainsi de la classification [1]. Nous utiliserons un histogramme sur les bytes (0x00 à 0xFF), c'est à dire la fréquence. Ce qui donne une information du même ordre que l'entropie ou des 1-gram.

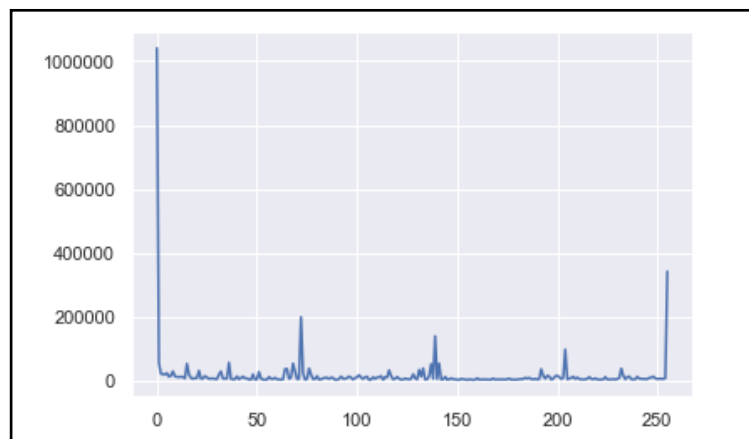


Figure 4. Histogramme des octets

Par ailleurs la distribution des bytes est une méthode utilisée pour déterminer l'écart d'un fichier par rapport à ce qu'il devrait être [8]. Certaines études utilisent des histogrammes pour déterminer le type d'un fichier. [9]

3.3. Libraires importées

Ici nous utilisons les 160 librairies les plus fréquentes – (Voir fichier *top_160_libs.txt*) de notre jeu de donnée. Ensuite pour chaque PE nous vérifions non seulement l'import de la librairie mais aussi le nombre de fonctions importées par ce dernier (ce qui permet d'avoir une plus grande différenciation

entre les fichiers). Par exemple si un fichier n'importe pas la librairie Win32.dll la valeur à ce point sera de sera 0, sinon la valeur sera le nombre de fonctions importées divisé par le nombre total (afin de formater entre 0 et 1 vu que l'ordre des grandeurs est très variable)

```
def libcount(exe):
    pe_dll = {}
    number_of_functions = []
    global common_libraries

    for dll in exe.imports:
        pe_dll.update( { dll.name.lower() : float(len(dll.entries)) } )
    # eg      from      "kernel32"          12          functions has been imported
    for func in common_libraries:
        if func in pe_dll:

            number_of_functions.append(pe_dll[func])

        else:

            number_of_functions.append(0.0)
    somme = sum(number_of_functions)
    if somme > 0 :
        return np.array(number_of_functions) / sum(number_of_functions)
    return np.array(number_of_functions,dtype=np.float64)
```

Figure 5. Nombre de fonctions importées

3.4. Extraction sur les fichiers et concaténation

Les résultats des 3 features précédente sont concaténées pour former un vecteur. L'opération est répétée sur tous les fichiers (le hash de chaque fichier est d'abord calculé afin de retrouver à quel fichier appartient un vecteur de features). On obtient alors un tableau à deux dimensions. Nous sauvegardons les données sous forme de tableau numpy sérialisés

```
for file in files:
    f_exe = path.join(directory,file)
    pe = parse(f_exe)
    if pe:
        count+=1
        parsed = pe[0]
        raw = pe[1]

        i_d = get_hash(raw)
        flags = get_flags(parsed)
        hist = histogram(raw)
        n_libs = libcount(parsed)

        pe_features = np.concatenate([flags,hist,n_libs])
        features.append(pe_features)
        hashes.append(i_d)
        print(count, end="\r")

print("")
hashes = np.array(hashes)
features = np.array(features)
```

Figure 6. Concaténation des vecteurs

Code pour extraction des features d'un seul fichier voir `lief_features.py`.

3.5. Désassemblage

Etant donné qu'il n'est pas envisageable de réaliser une analyse statique sans désassembler le fichier nous allons extraire de façon statistique des features issues du code désassemblé.

Nous allons le faire avec Capstone et pefile (qui tout comme lief permet de parser des PE windows) [7].

Désassembler un PE présente quelques particularités. Etant donné que nous voulons désassembler uniquement le code exécutable (beaucoup de séquences correspondent à un opcode).

Revenons d'abord certains points sur des champs de l' « Optional Headers » :

- RVA : l'adresse virtuelle relative, c'est l'offset (de l'adresse d'un élément) lorsqu'il est chargé en mémoire (RVA = adresse virtuelle – adresse de base)
- L'adresse du point d'entrée (AddressOfEntrypoint) : C'est l'adresse à partir de laquelle le PE est exécuté. Elle est relative à l'image de base.

```
def disassemble(file):
    ops=[]
    pe = pefile.PE(file)
    entryPoint = pe.OPTIONAL_HEADER.AddressOfEntryPoint
    data = pe.get_memory_mapped_image()[entryPoint:]
    if pe.FILE_HEADER.Machine == 0x14c:
        cs = Cs(CS_ARCH_X86, CS_MODE_32)
    else :
        cs = Cs(CS_ARCH_X86,CS_MODE_64)
    for i in cs.disasm(data, 0x1000):
        ops.append(i.mnemonic + " " + i.op_str)

    return np.array(ops)
```

Figure 7.Désassemblage

Etape de désassemblage :

Le désassemblage commence à partir du point d'entrée de la fonction.

Du code exécutable peut se trouver au niveau de diverses sections. L'alignement sur le disque n'est pas le même qu'en mémoire. Les adresses doivent être absolues : ImageBase + offset. Aussi les sections avec une adresses virtuelles au-delà de 0x10000000 sont souvent caduques et servent à tromper les outils d'analyse.[10]

Pour utiliser le code sous forme de features nous considérons la fréquence des symboles suivants :

```
["-", "+", "*", "[", "?", "@", "db", "dw", "dd"]
```

Les premiers sont typiques d'un code écrit pour rendre l'analyse statique difficile ou faire des appels indirect de librairies (l'adresse est obtenue en fonction d'une opération mathématique ou d'un XOR). Ayant remarqué que certains malware contiennent en majorité dans leurs code que des définition de données nous ajoutons aussi dd,db,et dw comme features.[11]

Les données ainsi obtenues sont concaténées aux résultats précédents. Code voir *disass.py*


```
import lief_features
import disass
import numpy as np
import sys

def features(file):
    a = lief_features.get_l_features(file)
    b = disass.get_features(file)
    return np.concatenate([a,b])
```

Code5.

4. Création de la Dataframe pandas

Les features sont calculées sur les deux jeux de données : « benign » pour les exécutable ordinaire et « malign » pour les malveillants.

Nous avons nos features sous formes de tableau numpy qui sont concaténées pour donner un tableau numpy de dimension (2031, 437) donc 2031 échantillons et 436 colonnes (les features). Nous ajoutons une colonne (y) la classe du fichier, malware ou non.

```
# les y qui indiquent malware ou pas
y = [1 for i in range(len(benign))] + [0 for i in range(len(malign))]

#type np.float64 entrée du réseau de neurones
benign = np.array(benign,dtype = np.float64)
malign = np.array(malign,dtype=np.float64)
y      = np.array([y],dtype = np.float64)

#concatenation
t_array = np.concatenate((benign,malign),axis=0)
t_array = np.concatenate((t_array,y.T),axis = 1)
```

Figure 8. Rassembler toutes les features en une ligne

Ensuite pour avoir un fichier de Dataset compréhensible nous mettons les noms des colonnes et créons une dataframe pandas avant de tout sauvegarder au format csv.

```
#nom des colonnes
h_features= ["has_configuration", "has_debug", "has_exceptions","has_exports", "has_import", "has_nx",
            "has_relocations", "has_resources","has_rich_header", "has_signature", "has_symbol","has_tls"]
hbytes    = [hex(i) for i in range(256)]
symb      = ["-", "+", "*", "[", "?", "@", "db", "dw", "dd"]
colnames  = h_features+hbytes+libs+symb+["classe"]

# conversation en dataframe pandas
dataset = pandas.DataFrame(t_array, index=rownames, columns=colnames)

#conversion en csv pour stockage sur disque
dataset.to_csv(r"C:\downloads\Mydataset.csv",index=False)
```

Figure 9. Nom des colonnes

Exemple d'un échantillon :

```
X[45]
array([1.00000000e+00, 1.00000000e+00, 1.00000000e+00, 0.00000000e+00,
       1.00000000e+00, 1.00000000e+00, 1.00000000e+00, 1.00000000e+00,
       1.00000000e+00, 1.00000000e+00, 1.00000000e+00, 0.00000000e+00,
       2.24455524e-01, 1.22027753e-02, 6.20307742e-03, 5.34016689e-03,
       8.66687586e-03, 3.21049206e-03, 3.42839876e-03, 3.47779095e-03,
       9.73897683e-03, 2.36791948e-03, 1.99021454e-03, 2.08609348e-03,
       7.36524650e-03, 2.09190433e-03, 1.76359157e-03, 1.14502708e-02,
       9.70411175e-03, 1.63575297e-03, 1.94953862e-03, 1.92338981e-03,
       4.15184901e-03, 3.09427516e-03, 1.69967227e-03, 1.24642633e-03,
       3.46907468e-03, 1.29872394e-03, 1.15635823e-03, 1.18250703e-03,
       2.23717546e-03, 1.24061548e-03, 1.20575041e-03, 1.60379332e-03,
       5.15131441e-03, 1.22027753e-03, 1.35973781e-03, 1.78974037e-03,
       3.37029031e-03, 1.27257513e-03, 8.62910536e-04, 1.13020942e-03,
       1.85075425e-03, 9.35546103e-04, 8.54194268e-04, 2.98677452e-03,
       1.94953862e-03, 3.67535969e-03, 1.44690050e-03, 1.33358901e-03,
       4.43948586e-03, 2.52100680e-03, 2.12967483e-03, 6.81031076e-03])
```

Figure 10. Exemple : features d'un exécutable

5. Entraînement du modèle

5.1 Description et réduction

Visualisons d'abord les colonnes de la Dataset, pandas possède une méthode qui nous permet de le faire de façon très simple :

```
: filepath = "C:/Downloads/dataset2.csv"
dataset = pandas.read_csv(filepath)
dataset.describe()
```

	has_configuration	has_debug	has_exceptions	has_exports	has_import	has_nx
count	2031.000000	2031.000000	2031.0	2031.000000	2031.0	2031.000000
mean	0.285081	0.805515	1.0	0.073855	1.0	0.642541
std	0.451564	0.395902	0.0	0.261600	0.0	0.479370
min	0.000000	0.000000	1.0	0.000000	1.0	0.000000
25%	0.000000	1.000000	1.0	0.000000	1.0	0.000000
50%	0.000000	1.000000	1.0	0.000000	1.0	1.000000
75%	1.000000	1.000000	1.0	0.000000	1.0	1.000000
max	1.000000	1.000000	1.0	1.000000	1.0	1.000000

8 rows × 438 columns

Figure 11. Description dataset

On peut voir des colonnes de la Dataset qui sont « inutiles » pour l'entraînement car n'apportant aucune information décisive (tous les échantillons ont la même valeur pour cette features) donc nous pouvons l'éliminer.

5.2. Création du modèle

Les X sont les features et les Y sont nos labels (0 ou 1)

```
# valeurs on ne prend pas les noms des colonnes et les hashes
array = dataset.values
# X = features 0 à 436 (inclus) de chaque ligne
X = array[:,0:437]
# Y malware ou pas malware à la dernière colonne de chaque ligne
Y = array[:,437]
```

Nous travaillons avec « Keras » une API au-dessus de Tensorflow. Nous allons utiliser des réseaux de neurones MLP, Multi Layer Perceptron.

```
model = Sequential()
```

Un modèle séquentiel par opposition à simultané, reçoit les données (les X) par paquets. Les réseaux de neurones sont alors entraînés progressivement. A ce point nous avons défini un réseau de neurones « vide »

```
model.add(Dense(450, input_dim=437, activation='relu'))
```

Ici nous ajoutons la première couche (la couche d'entrée) le modèle prend en entrée les 437 features de notre Dataset. Nous définissons aussi la première couche cachée (deuxième couche) avec comme fonction d'activation « Relu » [def relu(X) ; return Max (0.0, X)] [12]

```
model.add(Dense(300, activation='relu'))
model.add(Dense(200, activation='relu'))
model.add(Dense(120, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Nous ajoutons 3 autres couches de taille respectivement (300, 200, 120) avec toujours comme fonction d'activation « relu ». A savoir que le nombre de couche ou de neurones sont des paramètres dont on ne peut prédire avec exactitude l'efficacité. Le choix se fait selon l'expérience ou intuitivement et constitue une discipline à part entière .

Pour la dernière couche (la sortie) la fonction sigmoïde est choisie afin d'avoir des résultats entre 0 et 1 qui constitueront les prédictions.

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

La fonction de perte utilisée est l'entropie croisée : $-y \log(f(x)) - (1 - y) \log(1 - f(x))$. Elle permet de calculer l'erreur de prédiction et c'est grâce à sa variation (gradient par rapport aux poids) que le modèle apprend les pondérations. Dans le cas d'un réseau de neurones à plusieurs couches l'apprentissage se fait par rétropropagation du gradient : le gradient de l'erreur par rapport aux poids d'une couche est calculé à partir du gradient de l'erreur par rapport aux poids d'une couche supérieure.

$W_j^{(t+1)} = w_j^{(t)} - \eta [y^{(i)} - f(x^{(i)})] x_j^{(i)}$ est l'équation d'ajustement des poids, η est le taux d'apprentissage et ce qui suit est la dérivée de l'entropie croisée.

Optimisateur : La méthode Adam calcule des taux d'apprentissage adaptatifs (η varie selon les poids ou couches considérées) individuels pour différents paramètres. Il est recommandé de l'utiliser par

défaut car il améliore la vitesse de l'apprentissage, sa justesse et diminue les ressources matérielles nécessaires.

```
model.fit(X, Y, epochs=150, batch_size=12)
```

La ligne précédente permet de réaliser l'entraînement du modèle. Elle prend pour entrée les données d'entraînement (X), les prédictions (Y). Le batch size (taille des lots) est de 12, donc la Dataset est découpée en lots de 12 (modèle séquentiel), l'erreur est ajustée chaque 12 échantillons. Epochs ou époque en français est le nombre de fois que l'opération va être réitérée sur le jeu de donnée entier.

5.3. Résultats

```
2031/2031 [=====] - 0s 33us/step
Accuracy: 99.70
```

Le modèle donne une précision de l'ordre de 99,70 % sur la Dataset, la matrice de confusion est la suivante :

	Malware	Benign
Malware	987	0
Benign	6	1038

Fig4. Matrice de confusion

Trouvons ces exécutables labelisé « benign » qui ont été désignés comme malware par le modèle

```
print([rownames[i] for i in range(len(Y)) if Y[i]!=Ypredict[i]] )
```

Il s'agit des exécutables avec les md5 suivants :

```
3a9f737dbb4a7bc6aa149693faf540f0
f752052b9412ee0c1048dbf39a794e17
e2312f199976d03a7cf41e453c5af246
e70ac976a621fec17460f1f234662ef8
03956494403ab2cdae8e892a7b293ff8
2a4627ddff6f94893eb054362fed7cb2
```

Figure 12.Faux positifs

5.4. Enregistrement du modèle

```
# serialize model to JSON
model_json = model.to_json()
with open("C:/Downloads/model.json", "w") as json_file:
    json_file.write(model_json)

# serialize weights to HDF5
model.save_weights("C:/Downloads/model.h5")
print("Saved model to disk")
```

Pour utiliser plus tard notre modèle nous le sauvegardons au format json et hdf5, la description du modèle (couches, optimisateur, fonction de perte) est dans le fichier json et les poids au format hfd5.

6. Utilisation du modèle

```
model_path = "C:/Downloads/model.json"
h5 = "C:/Downloads/model.h5"

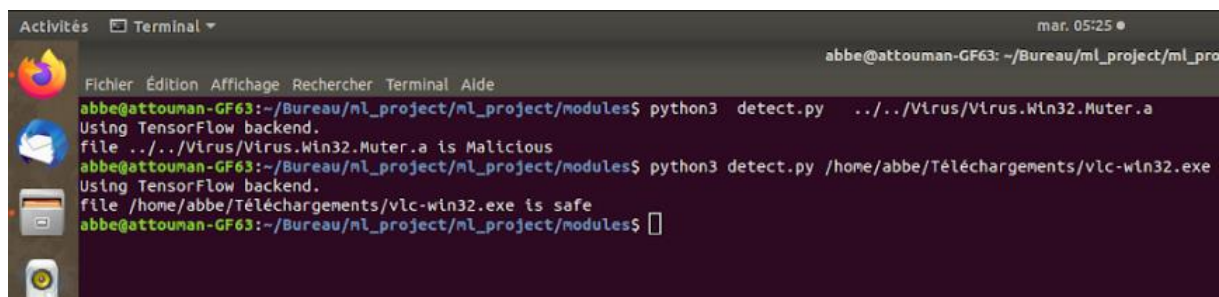
with open(model_path, 'r') as f:
    loaded_model_json = f.read()

loaded_model = model_from_json(loaded_model_json)
loaded_model.load_weights(h5)

f=features(sample)
prediction = get_nature(f)

if prediction :
    print("file %s is safe" %file_)
else:
    print("file %s is Malicious"%file_)
```

Notre code prend en entrée le chemin du fichier, calcule les features du fichier, charge le modèle à partir des données sauvegardées sur le disque et détermine si le fichier est malveillant ou pas.



```
mar. 05:25 ●
abbe@attouman-GF63: ~/Bureau/ml_project/ml_pro
Fichier Édition Affichage Rechercher Terminal Aide
abbe@attouman-GF63:~/Bureau/ml_project/ml_project/modules$ python3 detect.py ../../Virus/Virus.Win32.Muter.a
Using TensorFlow backend.
file ../../Virus/Virus.Win32.Muter.a is Malicious
abbe@attouman-GF63:~/Bureau/ml_project/ml_project/modules$ python3 detect.py /home/abbe/Téléchargements/vlc-win32.exe
Using TensorFlow backend.
file /home/abbe/Téléchargements/vlc-win32.exe is safe
abbe@attouman-GF63:~/Bureau/ml_project/ml_project/modules$
```

Figure 13 Démonstration

7. Analyse dynamique

Elle consiste à exécuter le fichier à l'aide d'un débogueur ou dans une sandbox (machine virtuelle configurée à cet effet) et observer son comportement. Elle peut se faire à plusieurs niveaux de niveau privilège (Ring 3 à Ring 0). Ici nous allons installer la sandbox cuckoo et ces dépendances pour analyser les faux positifs issu de notre analyse statique.

7.1. Cuckoo

Son principe est simple, elle utilise une machine virtuelle invité dans laquelle un agent est installé. *Cuckoo* démarre cette machine virtuelle, enregistre l'activité du malware, stocke ces informations sur la machine hôte, éteint la machine virtuelle, restaure l'état initial pour que la machine soit de nouveau opérationnelle pour une nouvelle analyse.

La mise en place de l'environnement consiste en l'installation des dépendances nécessaires à cuckoo (tcpdump, Mongodb, Moloch, mitm, diverses librairies.). Les possibilités sont multiples on peut y ajouter des IPS comme snort ou Suricata pour analyser des flux réseaux, l'interfacer avec le stack ELK

- Machine invitée

Ici nous utilisons Windows 7 sous VirtualBox : Nous commençons par installer l'agent cuckoo (code en python2) et le mettre en exécution automatique à chaque démarrage. Ensuite il faut désactiver Windows defender et le contrôle d'accès utilisateur. Installer des navigateurs,

Java, Office, copier des images et document afin de simuler un vrai système d'exploitation en usage. Les additions invités sont aussi à désinstaller. L'interface réseau est de type « réseau privé hôte », l'adresse de la machine est statique avec comme passerelle par défaut l'adresse d'écoute de cuckoo sur la machine hôte. Lorsque la machine est configurée, un instantané de l'image est sauvegardé. (C'est lui qui sera restauré après chaque exécution)

- Sur la machine hôte

La configuration se fait dans les fichiers conf de cuckoo dans CWD/conf/ (cuckoo working directory, par défaut ~/. Cuckoo/) [13]

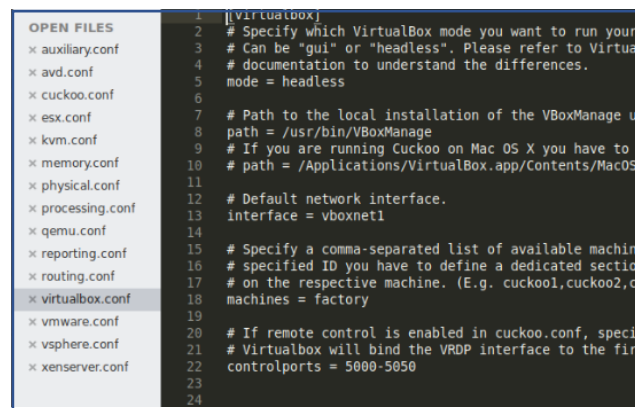


Figure 14. Fichiers de configuration Cuckoo

7.2. Analyse des Faux positifs

Ici nous allons analyser dynamiquement les faux positifs (fig5.) issus de l'analyse statique. Les fichiers sont les suivants :

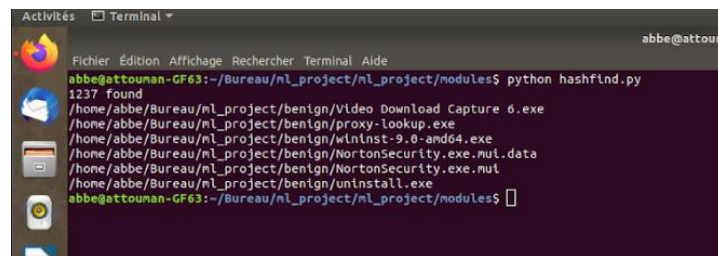


Figure 15. Fichiers de configuration cuckoo

Nous savons déjà souvent que les antivirus sont souvent confondus avec des exécutables malveillants. Lançons cuckoo via son interface web et soumettons le premier fichier.

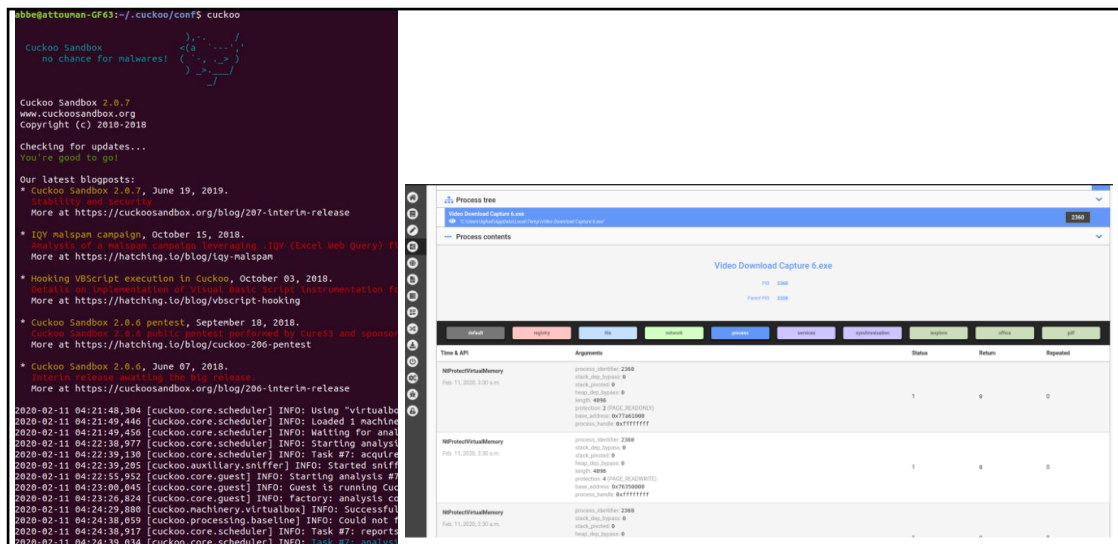


Figure 16. Résultats Analyse cuckoo

Les résultats n'indiquent aucun comportement malveillant, ni dans les processus, ni dans les accès réseaux, aucun fichier n'est écrit ou lu. En utilisant une autre sandbox (en ligne) sur any.run le seul comportement suspect est la tentative d'installation de chrome (qui était déjà installé sur notre sandbox cuckoo)

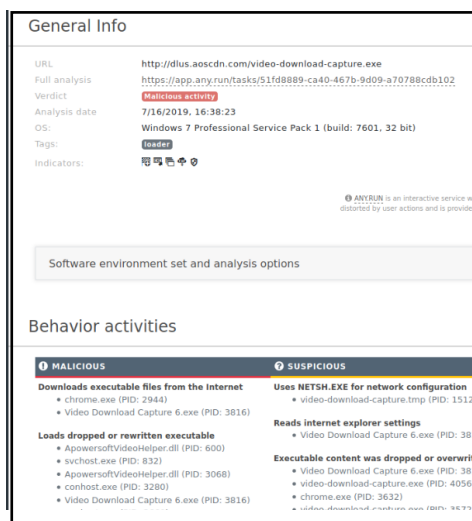


Figure 17. Résultats Any.run

8. Limites de la détection par machine learning et analyse statique.

Concernant les PE l'analyse statique est contournable du moment où le fichier peut être modifié sans altération de ses fonctionnalités :

- Overlay : le format PE autorise l'ajout de données à la fin du fichier et qui ne sont pas chargés en mémoire.
- Taille des sections : elles doivent être des puissances de 2, lorsque la taille des données utiles de la section n'atteint pas une puissance de 2 le reste est « paddé ». Les données du padding ne sont pas aussi chargées en mémoire
- Il est aussi possible d'ajouter des sections non référencées par le champ Optional Header

De ces observations précédentes, les features basées sur les histogrammes, l'entropie ou en lien avec la structure du fichier sont alors facilement modifiable. Nous pouvons alors ajouter des bytes jusqu'à avoir la même fréquence qu'un exécutable bénin. Ici nous le faisons avec lief.

```
import lief
file = "tweakpng.exe"
constant = 34
size = 100000
binary = lief.parse(file)

builder = lief.PE.Builder(binary)

section = lief.PE.Section(".Mysection")

section.content = [constant] * size

binary.add_section(section, lief.PE.SECTION_TYPES.DATA)

builder.build()

builder.write('output.exe')
```

output	11/02/2020 11:09	Application	435 Ko
rebuild	11/02/2020 11:09	Fichier PY	1 Ko
tweakpng	07/09/2014 18:03	Application	337 Ko

Figure 18. Modification avec lief

Le fichier est toujours exécutable mais ses « features » sont complètement changées. Ainsi nous pourrions faire des modifications jusqu'à ce que le « Benin » devienne « Malicious ». [14]

Conclusion

Au terme de ce rapport nous avons étudié le format PE, réalisé une détection par machine learning en utilisant des features statique. Nous avons aussi pu comprendre le fonctionnement de l'analyse dynamique avec cuckoo et enfin vu les limites du machines learning en analyse statique. Ce travail nous a permis une introduction au monde du reversing, de la détection de fichier malveillant, nous avons compris le fonctionnement de certains algorithmes de machine learning et améliorer nos compétences en python et git.

Code

https://github.com/ezios/ml_project [Actuellement en private]

Références

- [1] <https://doi.org/10.1016/j.jnca.2019.102526>
- [2] <https://docs.microsoft.com/fr-fr/windows/win32/debug/pe-format?redirectedfrom=MSDN>
- [3] <https://virusshare.com>
- [4] <https://lief.quarkslab.com/doc/stable/api/python/pe.html>
- [5] https://lief.quarkslab.com/doc/latest/doxygen/classLIEF_1_1PE_1_1Binary.html#a12c7ad2e6e4ed50db96fe830a6577a0c
- [6] <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.9861&rep=rep1&type=pdf>
- [7] https://www.capstone-engine.org/lang_python.html
- [8] <https://patentimages.storage.googleapis.com/33/aa/8d/4104254d4f6b5a/US20080276320A1.pdf>
- BYTE-DISTRIBUTION ANALYSIS OF FILE Publication Classification SECURITY (51) Int. Cl. G06F II/00 (2006.01)
- [9] Lee, Y. & Kim, J. (2015). File feature analysis for file identification based on histogram. Information (Japan). 18. 1047-1052.
- [10] <https://github.com/erocarrera/pefile/blob/master/pefile.py>
- [11] Mansour Ahmadi et Al. Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification
- [12] <https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/>
- [13] <https://cuckoo.sh/docs/installation/host/configuration.html>
- [14] <https://towardsdatascience.com/evading-machine-learning-malware-classifiers-ce52dabdb713>