Polyglottes binaires et implications

Ange Albertini
ange.albertini@gmail.com

Corkami.com

Résumé De l'exploitation à l'infection, les *malwares* modernes utilisent de nombreux formats de fichier binaires. Il est crucial de pouvoir correctement les identifier et les analyser, si possible de manière automatique. Bien que ces formats soient a priori clairement différenciés, il est parfois possible de combiner certains d'entre eux dans un seul et même fichier. On parle alors de fichiers *polyglottes*. De tels fichiers *polyglottes* ont donc dans un premier temps été créés. Ensuite, plusieurs caractéristiques non documentées de chaque format concerné ont été rajoutées, pour mettre en évidence l'importance du problème entre les limites des documentations officielles, et la réalité (du monde des virus). Les conséquences sur le fonctionnement des outils de sécurité sont finalement mises en évidence, avec ce que ça implique pour l'utilisateur final.

1 Introduction

1.1 État de l'art

Un fichier polyglotte est un fichier qui peut être interprété dans plusieurs langages différents. Un des plus simple d'entre eux, représenté par le Polyglot de David Kendall dans le listing 1, fonctionne en Ruby, Perl, PHP, ksh, Scheme, Lisp, Clojure, Plan. Les *polyglottes* peuvent aller beaucoup plus loin, jouant sur les différences d'interprétation et de *pré-processing* de chaque langage inclus dans le fichier.

```
(print "Hello, world!\n");
```

Listing 1. Un programme polyglotte simple.

La première utilisation de fichiers polyglottes dans le monde des virus est GIFAR [4] :

un GIFAR résulte de l'ajout d'une classe *Java* stockée au format *JAR* à la suite d'une image au format *GIF*. En effet, le *GIF* est un format d'image couramment utilisé sur le Web, notamment pour les avatars dans les forums de discussion. Une fois le fichier sur le serveur, il est ensuite utilisable aussi comme une classe *Java*. Si on utilise cette classe dans un script quelconque sur le serveur, le navigateur laissera accèder à la

classe, même si elle est malveillante, car elle est maintenant hébergée sur le même site.

Julia Wolf [3] a introduit le concept de *ZIP* et *Portable Document Format* — dit *PDF*— combinés, ce qui permet de tromper les outils de sécurité et ainsi permettre d'exfiltrer des informations, ou passer pour un fichier innocent.

Jonas Magazinius [2] s'est également penché sur la question, en combinant notamment image *JPG* et document *PDF*.

En revanche, les formats de fichiers binaires sont, eux, rarement compatibles, car exclusifs : en effet, la majeure partie d'entre eux doivent nécessairement débuter par un marqueur (on parle aussi de *magic number* en anglais), spécifique à chaque format, à de rares exceptions près (tel `0xcafebabe`, pour le format *Mach-O* universel et les classes *Java*).

Il ne semble donc pas possible de combiner des formats binaires en général. Cependant, il y a quelques exceptions :

- tout d'abord, certains formats de fichier n'ont aucun en-tête, tels que le *Master Boot Record* — dit *MBR* — ou les fichiers *COMMAND FILE* — dit *COM*. Il est donc possible de créer un secteur de démarrage à la fois au format *GRUB* et au format *MBR* [1], ou bien un fichier *COM* qui soit également un fichier exécutable linux *ELF*.
- d'autre parts, d'autre formats de fichiers n'imposent pas que le marqueur soit présent au début du fichier : c'est le cas des archives (*ZIP*, *RAR*. . .), des pages *HTML*, et des documents *PDF*.
- enfin, certains formats, bien que commencant obligatoirement au premier déplacement, autorisent un grand espace entièrement modifiable juste après la signature, tels que le format d'exécutable *Portable Executable* — dit *PE*— et le format d'image *PICT*.

Hélas, certains formats utilisables à des fins malveillantes sont librement combinables dans un seul et même fichier, mais de plus, il s'agit des formats parmi les plus utilisés ces dernières années pour la prolifération de *malware* : en effet, la chaîne d'infection complète par un virus la plus répandue est une page *HTML*, contenant du Javascript, qui va charger un fichier *Java* profitant d'une vulnérabilité, qui lui-même va télécharger le virus final, sous forme d'un exécutable *Windows*, au format *PE* — et on peut effectivement combiner une page *HTML*, une classe *Java*, et un fichier binaire au format *PE*.

Un autre vecteur d'infection répandu est un *PDF* activant une vulnérabilité, dont le *shellcode* téléchargera un *PE*. Et là encore, *PE* et *PDF* sont combinables dans un seul et même fichier.

1.2 Pivot

Le format pivot de la grande majeure partie des virus est le format de binaire universel de *Windows*, le format *PE*. Pour ce rôle unique dans la chaîne virale, il a donc été choisi comme point de départ.

2 Exploration du format *PE*

Ce format stipule que le fichier commence par une structure `IMAGE_DOS_HEADER`, de `0x40` octets :

- les 2 premiers octets de cette structure définissent un champ appelé `e_magic`, qui contient obligatoirement la signature `MZ` : Il est donc impossible de le combiner à tout autre format imposant une signature spécifique au déplacement 0.
- tous les champs suivants, à l'exception du dernier, ne concernent que la fonctionnalité *DOS* de l'exécutable — qui se borne en majeure partie à afficher un message d'erreur. Elle est totalement ignorée quand le fichier est chargé en tant que *PE*.
- le dernier champ, `e_lfanew`, est un pointeur sur 32 bits vers la structure suivante de l'exécutable.

On sait donc que notre fichier doit commencer par `M` et `Z`, et qu'au déplacement `0x3C` doit se trouver un pointeur. Entre les deux, on peut y faire ce que l'on veut, tout en gardant la fonctionalité *PE* intacte. On peut même mettre l'en-tête *PE* à la fin du fichier sans problème.

2.1 Création d'une page *HTML* dans un exécutable

Le laxisme omniprésent dans les pages *HTML* fait qu'au final, les navigateurs se contentent d'ignorer un peu tout — quand bien même ce serait de l'information non encodable en *ASCII* — au cas où un tag <HTML> finirait par être présent.

Ainsi, le simple fait de rajouter à la fin d'un exécutable du code *HTML* va en faire une page web fonctionnelle, pour peu qu'on renomme l'extension du fichier, comme le montre la figure 1.

Cela dit, bien que le fichier soit fonctionnel dans les deux formats, les données binaires perturbent l'affichage de la page web, bien que celles-ci soient placées avant le tag ouvrant <HTML>. On va donc jouer sur les *Cascaded Style Sheet* pour effacer ces données et ainsi avoir un résultat visuellement parfait. Ou sinon, on peut insérer ce tag juste après la signature du *PE*, auquel cas on aura juste un `MZ` affiché sur la page.

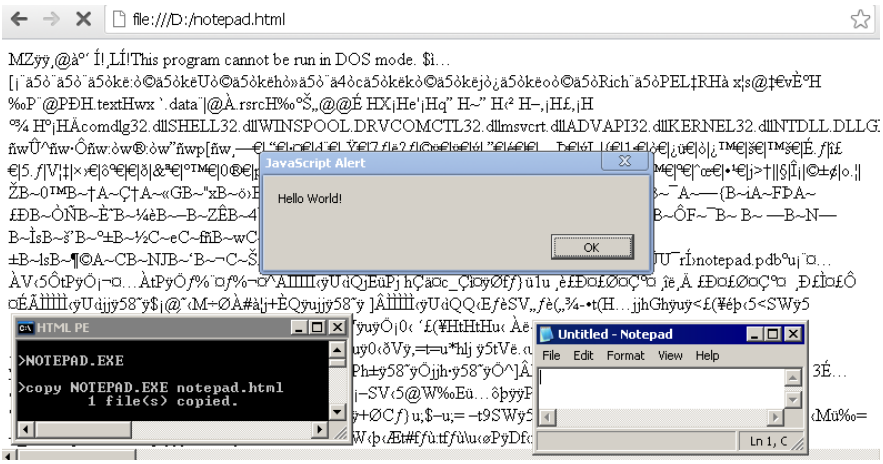

Figure 1. Le bloc-note *Windows* avec une page HTML ajoutée à la fin.

De même, si on souhaite rajouter quelque chose après la page web et son script, il nous suffit d'ouvrir un bloc de commentaire non fermé `<!--`.

À noter que si une page web doit avoir une extension précise pour être visualisée correctement, un exécutable *Windows* n'a besoin d'aucune extension particulière pour être exécuté via les APIs *Windows*.

2.2 Concaténation d'un *PDF* et d'un exécutable

La spécification officielle stipule que la première ligne d'un *PDF* doit être sa signature :

> **7.5.2 File Header**
>
> The first line of a PDF file shall be a *header* consisting of the 5 characters %PDF– followed by a version number of the form 1.N, where N is a digit between 0 and 7.

En pratique, il n'en n'est rien, il est juste requis sous *Adobe Reader* qu'une signature valide soit présente dans les 1024 (`0x400`) premiers octets.

On peut donc, dans un premier temps, rajouter à la fin d'un petit exécutable un *PDF* dans son intégralité, et les deux fonctionneront encore comme on s'y attend, comme dans la figure 3.

Dans le cas d'un *PE* plus gros que `1024` octets, on devra rajouter la signature *PDF* dans l'en-tête *PE*.

Il arrive cependant que le *PE*, stocké au format binaire, contienne des séquences de caractères qui seront interprétées par la visionneuse *PDF*. Pour cela, il suffit d'entourer le reste du *PE* d'une structure d'objet *stream* fictive, et ainsi son contenu ne sera pas analysé, puisqu'il n'est pas référencé dans la hiérarchie du document, comme dans la figure 4.

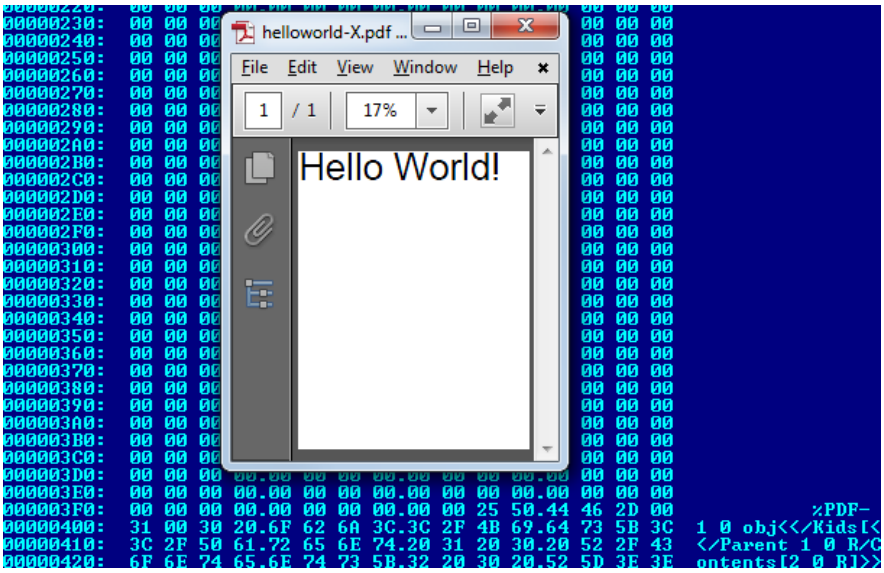

Figure 2. Un *PDF* dont la signature est à la fin des `1024` premiers octets est toujours reconnu par *Adobe Reader*.

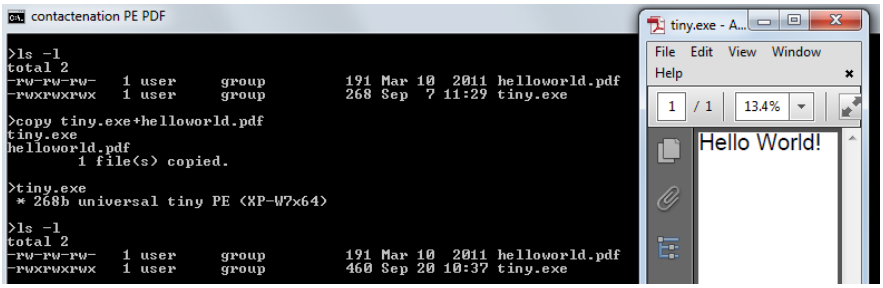

Figure 3. Un *PDF* greffé à la fin d'un petit exécutable *PE*.

2.3 Une archive *ZIP* dans un exécutable

Le format *ZIP* n'impose rien quant au début du fichier — il est lu en partant de la fin, et tolère des données avant ou après la structure du format elle-même. Un fichier *ZIP* peut donc se trouver n'importe où dans un autre fichier.

Une classe *Java* peut être exécutée directement, ou en étant contenue dans un fichier *Java Archive*, dit *JAR*. Un fichier *JAR* n'est qu'une archive *ZIP* contenant une ou plusieurs classes *Java*, et un fichier texte (`manifest`) au contenu bien défini.

Contrairement à un *ZIP* standard, un fichier *JAR* doit terminer strictement le fichier : si on rajoute ne serait-ce qu'un seul octet à la suite, *Java* ne reconnaît plus ce fichier comme valide.

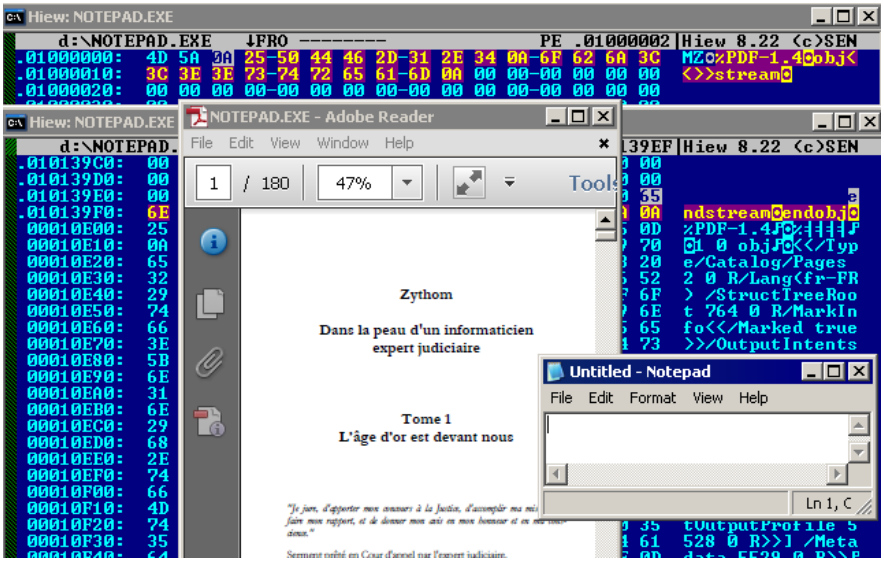

Figure 4. Le bloc-note *Windows* avec un fichier *PDF* ajouté à la fin.

2.4 Récapitulons

À l'aide des manipulations décrites précédemment, nous obtenons donc :

1. le début d'un *PE*
2. une signature *PDF* dans ses `1024 - 6` premier octets
3. la suite du *PE*, inclue dans un objet `stream` *PDF*
4. dans un ordre quelconque :
   - le reste du *PDF*, finissant l'objet fictif
   - la page *HTML*
5. un *ZIP*, optionnellement un *JAR*

3 Détails de mise en oeuvre

Un simple fichier polyglotte est donc faisable via des opérations de concaténations et quelques modifications manuelles.

Pour pousser l'expérience quant au comportements d'outils de sécurité, des propriétés anormales seront ajoutées aux formats inclus, dans le but de tester leur robustesse.

Afin d'y arriver, l'intégralité du fichier sera faite à la main, pour en garder le contrôle complet.

Toutes les structures sont crées à partir de zéro, en assembleur : en effet, un assembleur standard a la possibilité de définir chaque octet

manuellement, et aussi de créer des fonctions et des structures, qui vont faciliter la création du fichier, tout en gardant le contrôle de chaque octet.

Quand un fichier est stocké dans une archive *ZIP* sans compression, il est stocké dans son intégralité, à l'identique : inclure un fichier dans une archive ne revient donc qu'à ajouter en-tête et pied de page.

3.1 Un *PE* fait main

Peu d'éléments définis Notre exécutable contient `Hello World` compilé en assembleur standard, mais seulement 18 éléments sont absolument indispensables, toutes structures confondues, tel que le montre la figure 5.

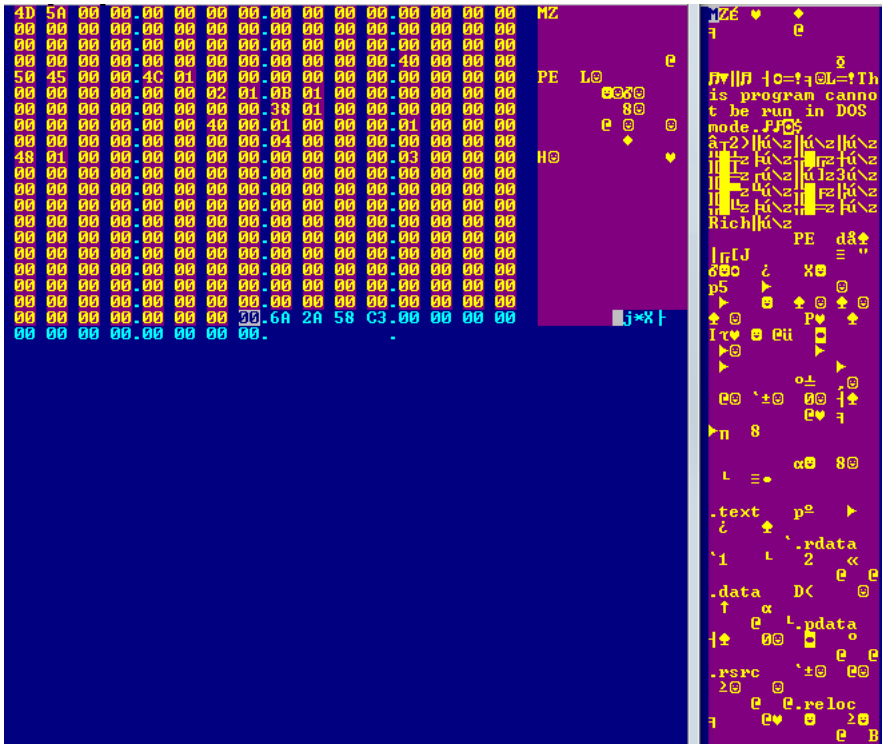

FIGURE 5. Comparaison d'en-têtes *PE*.

On obtient donc un *PE* qui va faire échouer les outils naïfs qui déterminent la validité du fichier d'après des éléments non indispensables.

Pas de sections Quand un exécutable utilise des alignements faibles, tout l'exécutable est chargé en mémoire indépendamment de la table des sections. Il devient dans ce cas possible de ne définir aucune section

(*NumberOfSections = 0*), et de ne pas avoir de table de section (*SizeOfOptionalHeader = 0*).

Là encore, il fera échouer les outils qui analysent ce format section par section.

Utilisation d'instructions non documentées Un outil de sécurité avancé va essayer d'émuler l'exécution du début du fichier, pour déterminer s'il est malveillant ou non.

Certaines instructions, non documentées, peuvent amener un tel émulateur à abandonner, car il ne peut déterminer comment procéder. La présence de telles instructions ne peut vraisemblablement pas amener à qualifier le fichier de malveillant, car il pourrait s'agir de nouvelles instructions d'un processeur récent.

Certaines de ces instructions comme `SetALC` ou les `nop` multi-octets sont absentes de la documentation Intel (figures 6 et 8), mais présentes dans la documentation AMD (figures 7 et 9), parfois uniquement sous le status *réservé*.

Ces instructions ne sont reconnues dans aucun outil Microsoft (figure 10), car ils suivent la documentation Intel à la lettre.

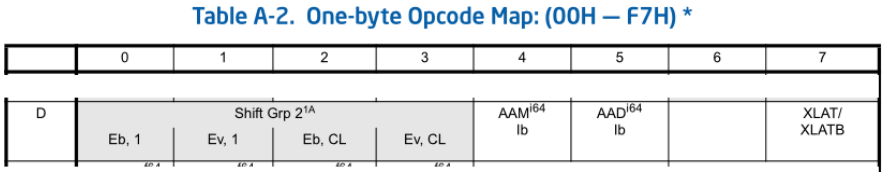

Figure 6. La documentation officielle n'indique rien pour l'octet `D6`.

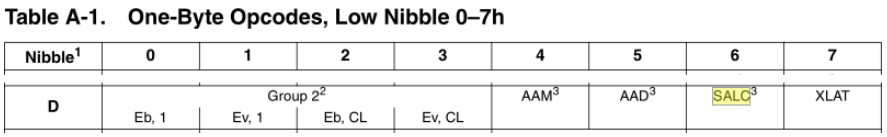

Figure 7. SetALC est documenté par AMD.

Pour implémenter ces instructions, on doit générer certaines à la main, un assembleur standard ne les gérant pas forcément.

Une structure d'import très compacte le *DataDirectory* des imports pointe vers une liste de *descripteurs*. Cette liste se termine en théorie

Table A-3. Two-byte Opcode Map: 08H — 7FH (First Byte is 0FH) *

	pfx	8	9	A	B	C	D	E	F
0		INVD	WBINVD		2-byte Illegal Opcodes UD2[1B]		NOP Ev		
1		Prefetch[1C] (Grp 16[1A])							NOP Ev
		vmovaps	vmovaps	cvtpi2ps	vmovntps	cvttps2pi	cvtps2pi	vucomiss	vcomiss

FIGURE 8. La documentation Intel ne mentionne NOP que pour les octets `0F 1F`.

Table A-4. Second Byte of Two-Byte Opcodes, Low Nibble 8–Fh

Prefix	Nibble[1]	8	9	A	B	C	D	E	F
n/a	0	INVD	WBINVD	invalid	UD2	invalid	Group P[2] PREFETCH	FEMMS	3DNow! See "3DNow!™ Opcodes" on page 351
n/a	1	Group 16[2]	NOP[3]	NOP[3]	NOP[3]	NOP[3]	NOP[3]	NOP[3]	NOP[3]

FIGURE 9. Le NOP de plusieurs octets est complètement documenté par AMD.

par un descripteur entièrement nul, mais en réalité uniquement si ses champs `Name` ou `FirstThunk` sont nuls.

La table des adresses, si elle est suffisamment petite, peut-être intégrée dans un des descripteurs, alors qu'elle utilise habituellement son propre espace du fichier.

De même, le nom de la DLL, même amputé de son extension, peut être inclus dans un des descripteurs, ainsi que la structure `IMAGE_IMPORT_BY_NAME` de l'import, alors qu'eux aussi sont censés être instanciés séparément, hors des descripteurs.

On obtient donc une table d'import très compacte, avec beaucoup d'irrégularités, mais pourtant 100 % gérée et fonctionnelle (figure 11) par *Windows*.

Ces astuces font que les outils les plus avancés échouent à analyser le *PE* parfaitement. Notamment, à l'heure de création de ces recherches, la dernière version de Hiew et IDA échouent à les gérer (figure 12).

3.2 Un *PDF* très compact

La signature *PDF*, bien qu'officiellement `%PDF-1.?`, peut en fait être tronquée, selon la visionneuse employée.

La signature de fin `%%EOF` est absolument superflue, ansi que toute référence à une longueur dans les objets `stream`.

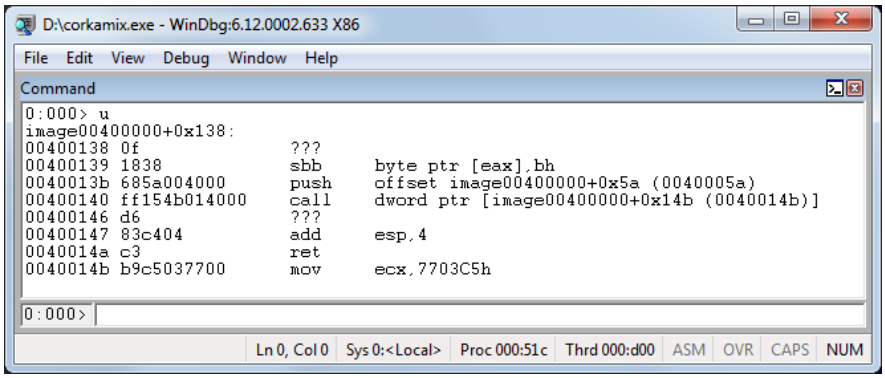

FIGURE 10. WinDbg montre plusieurs instructions inconnues.

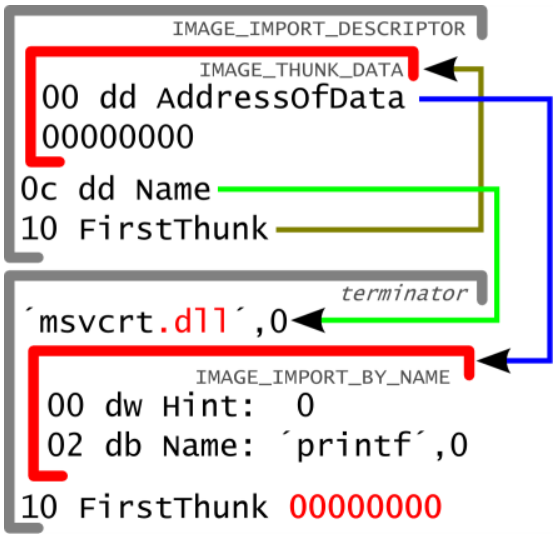

FIGURE 11. Une structure d'imports avec beaucoup de malformations.

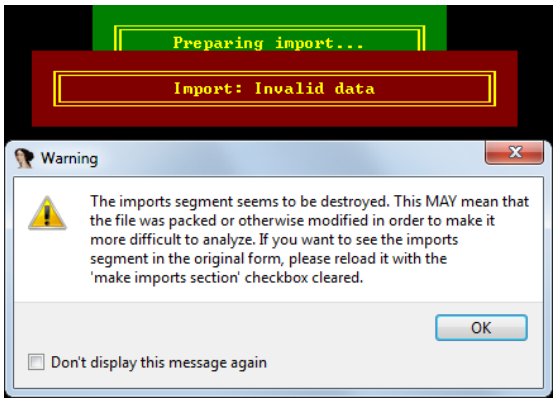

FIGURE 12. Une structure d'imports non gérée par Hiew ou IDA.

De même, la table `xref` est supprimable, ansi que les terminateurs d'objet `endobj`.

Enfin, le trailer final est réduit à son strict minimum.

On obtient donc, dans des cas extrêmes, un fichier *PDF* particulièrement réduit de 36 octets fonctionnant sous Adobe Reader (listing 2).

```
%PDF- trailer<</Root<</Pages<<>>>>>>
```

Listing 2. Un *PDF* valide de 36 octets.

3.3 Un *patchwork* binaire

Au final, on obtient un fichier fonctionnel dans tous les formats inclus (figure 13), et de nombreuses caractéristiques non documentées, qui font échouer des outils d'analyse et des anti-virus répandus.

On peut rajouter de nombreux formats dans le fichier *ZIP* lui-même, mais ça ne présente aucun challenge particulier.

Un tel mélange binaire ne nécessite pas de faire ces fichiers à la main : on peut prendre des fichiers tout à fait standards et obtenir un résultat analogue, bien qu'ayant un air plus familier (figure 14).

3.4 Autres systèmes d'exploitations

Un esprit moqueur serait tenté de se contenter de blâmer *Microsoft* et *Adobe* pour leur laxisme, et le fait d'accepter ainsi n'importe quel format binaire.

Linux En regardant le format *PDF* dans un premier temps, on s'aperçoit que les visionneuses *PDF* standards sous Linux, bien que n'acceptant pas le *PDF* particulier généré précedemment, n'en sont pas pour autant vraiment rigoureuses.

En effet, en creusant un peu, on s'aperçoit que de nombreux outils *PDF* tels qu'Evince ou la visionneuse standard d'Ubuntu, acceptent un *PDF* ne contenant même pas de signature `%PDF` — même partielle — ce qu'*Adobe Reader* refusera immédiatement.

Le format binaire Linux, *Executable File Format* — dit *ELF*— n'impose rien de particulier concernant le fichier lui-même. Créer un tel polyglotte basé sur un *ELF* plutôt qu'un *PE* ne présente alors pas de difficulté particulière, tel que le montre la figure 15.

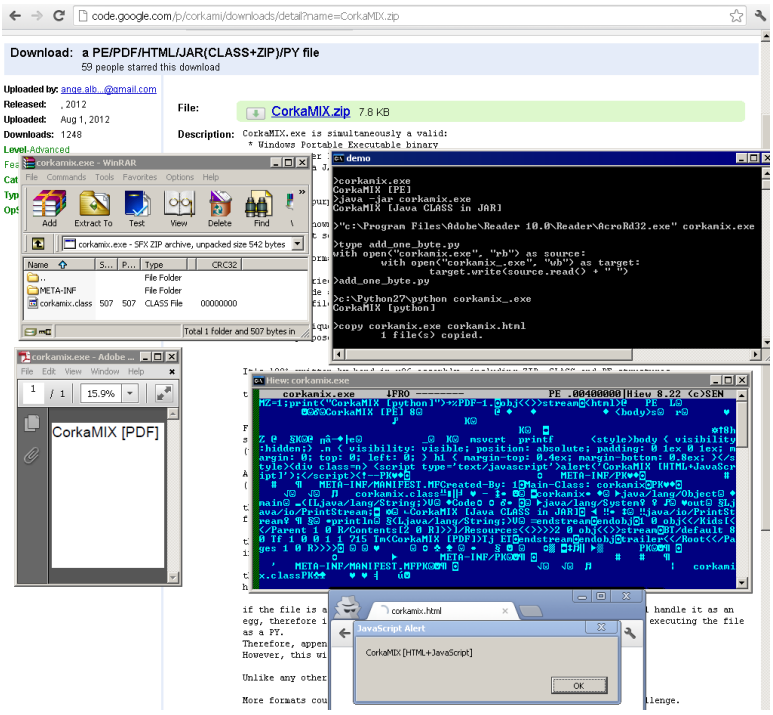

FIGURE 13. Un joli *patchwork*

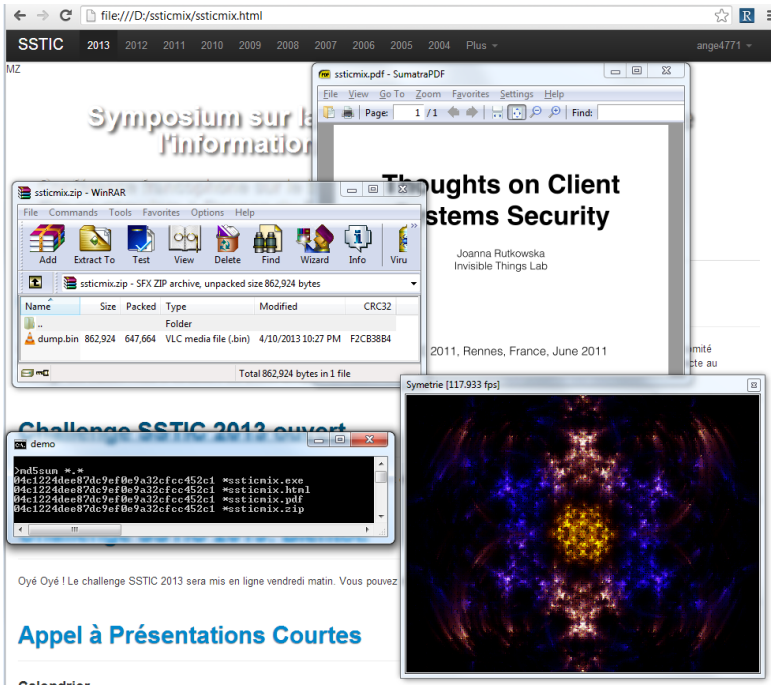

FIGURE 14. Un polyglotte binaire avec un air familier.

Mac OS X Les visionneuses *PDF* standards Mac OS X et iOS sont elles, beaucoup plus strictes.

Elles refusent toute signature incomplète ou absente. Les seules malformations théoriques acceptées sont l'absence de `xref`, longueurs de flux (`stream /length`), et signature de fin de fichier `%%EOF`.

Le format binaire de Mac OS X, le *Mach-O* (pour *Mach Object*) ne présente, à l'instar des formats *PE* et *ELF*, aucune difficulté particulière une fois qu'on a la maîtrise de leur structure binaire (figure 16).

3.5 comparaison d'autres visionneuses *PDF* répandues

Bien que le standard *PDF* soit publiquement défini, chaque visionneuse tolère des malformations particulières.

Ceci peut être exploité pour créer un document qui contient un code malveillant qui ne sera vu et traité que par une visionneuse spécifique.

En examinant des visionneuses standards telles que *Sumatra*, *Chrome* et *Adobe Reader*, on s'aperçoit qu'il est possible de faire un document, qui, bien que fonctionnant sans problème sous les trois, se révèle en fait être composé de trois documents indépendants fusionnés, dont chaque élément co-existe en s'ignorant (cf figures 17 et 18).

Chaque élément spécifique à une visionneuse étant ignoré par les autres, on peut donc le mutiler au maximum, ce qui fait qu'il pourrait être perçu comme corrompu, et ainsi passer outre des filtres de sécurité : la figure 18 met en valeur les différences de chaque élement, et ainsi les largesses de chaque visionneuse par rapport aux autres.

La visionneuse *PDF.JS* est un cas à part : elle bien plus stricte que toutes les autres. Notamment, elle impose :

– une table de références `xref`.
– une longueur déclarée à chaque `stream`.

qui sont des éléments dont l'absence est tolérée par toutes les autres.

4 Conséquences pour la sécurité

Tels des marques de lessive, les outils de sécurité sont comparés selon de trop simples critères : vitesse d'analyse et taux de détection d'un même ensemble empirique de fichiers.

Tout outil d'analyse commercial grand public a donc la lourde responsabilité de ne pas trop s'attarder sur un fichier particulier. En conséquence, il est crucial que le moteur de celui-ci détecte au plus tôt, et efficacement, le type du fichier analysé.

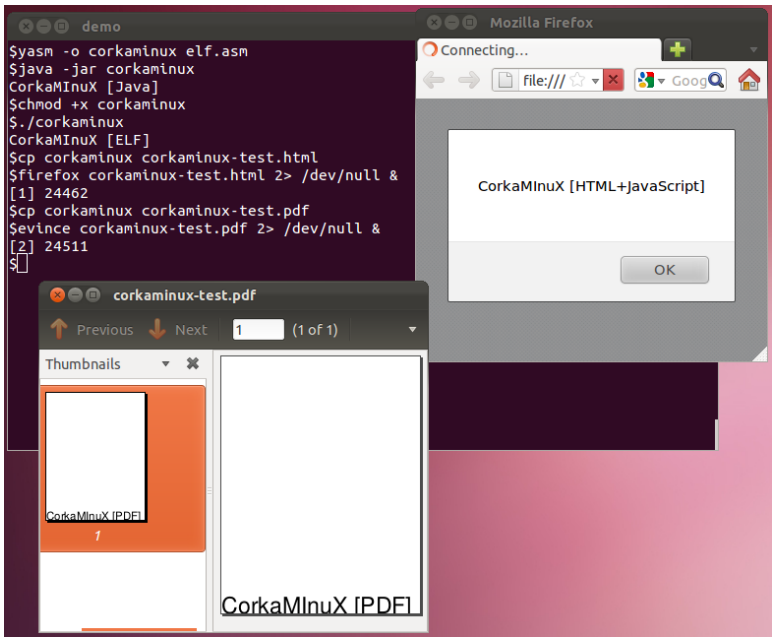

FIGURE 15. Un polyglotte binaire sous Linux.

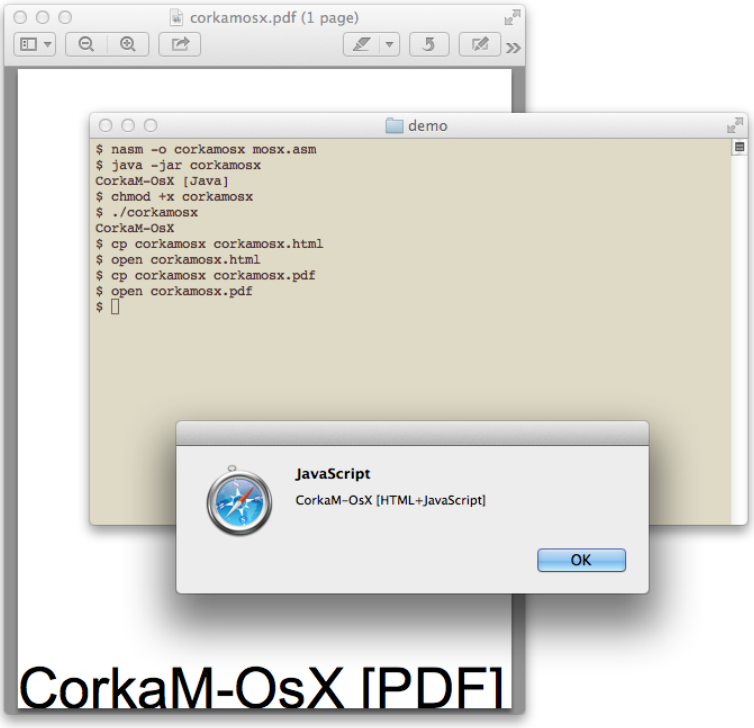

FIGURE 16. Un polyglotte binaire sous Mac OS X.

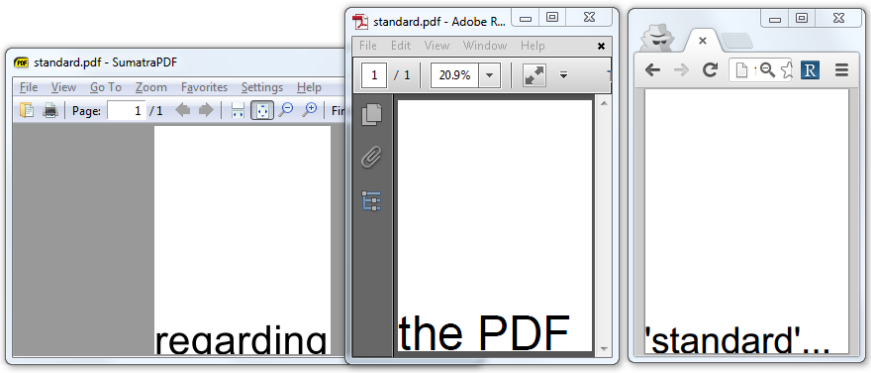

FIGURE 17. Un document interprété différemment par 3 visionneuses distinctes.

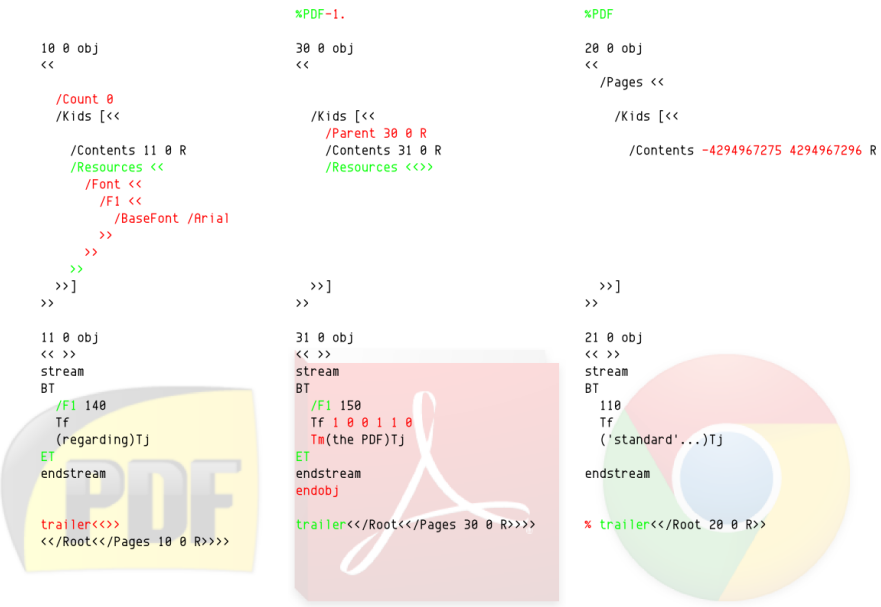

FIGURE 18. Comparaison des différences possibles entre 3 visionneuses.

On peut donc imaginer le début de l'exécution d'un tel outil comme une grosse gare de triage : il n'est donc pas surprenant qu'un fichier, une fois détecté comme correspondant à un format connu (comme le montre la figure 19), sera analysé comme tel, et donc ne sera pas réanalysé par la suite comme étant d'un autre format.

Quand bien même il le serait, l'outil ne va pouvoir se permettre de réanalyser totalement le fichier comme étant d'un autre format, car il va au bout d'un moment couper l'analyse, pour ne pas s'attarder trop longtemps sur un seul fichier.

Soit la machine qui traite le fichier passe beaucoup de temps (déni de service sur l'anti-virus, ou du moins perte de qualité en matière de vitesse

Figure 19. Un exécutable identifié uniquement comme *PDF*.

aux yeux des magazines informatiques, ce qui est hors de question pour un produit commercial), soit l'anti-virus jette un fichier qu'il ne peut analyser de manière exhaustive (déni de service sur la passerelle de fichiers, qui est généralement jugé inacceptable).

Une première démarche pour l'auteur de virus est donc de légitimement feindre un format de fichier : l'outil de sécurité analysera ce fichier comme tel, puis, ne trouvant rien de malveillant dans ce format particulier, déclarera le fichier comme sain. Ça peut donc se faire simplement en commençant un fichier par la signature *PE*, puis en continuant le fichier comme un fichier *HTML* ou *PDF*.

De plus, certains outils utilisent un système de cache pour savoir s'ils ont déjà analysé un fichier. Donc, la réutilisation du même fichier — même sous un autre format — sautera toute nouvelle analyse via une interception dynamique lors de l'exécution.

Puisqu'un outil de sécurité se doit d'être le plus rapide possible et le plus simple possible, il ne rend comme verdict qu'un booléen, sans tenir compte du format possible du fichier : un booléen par fichier, et non par type possible de fichier. Le vrai problème, c'est qu'un autre logiciel vulnérable, tel que *Java*, une visionneuse *PDF* ou un navigateur, va peut-être interpréter ce fichier de manière différente.

Une deuxième démarche consiste à inclure un maximum de format, pour que, quelque soit l'ordre d'analyse de l'outil, il atteigne sa limite imposée de temps/cycles, et ainsi qu'il abandonne avant d'avoir pu déterminer si le fichier est sain ou non.

Une autre possibilité est que le fichier cause des problèmes au moteur de l'outil, et qu'il échoue directement.

La première priorité pour un outil de sécurité est de déterminer au plus tôt qu'il y a une combinaison inhabituelle de formats binaires dans un seul fichier, ce qui augmente la suspicion. Si cette combinaison ne concerne aucun cas légitime, tels qu'un *HTML* combiné à un exécutable *PE*, le fichier peut être classé comme malveillant selon cet unique critère. Malheureusement, cette combinaison de formats peut être perçue comme un avantage technique par des développeurs peu renseignés, et, bien que déconseillée en pratique, peut être rencontrée dans la nature.

D'autre part, tous les navigateurs modernes analysent l'intégralité d'un fichier pour son contenu Web, même s'il est hors norme : par exemple, un fichier constitué d'un exécutable *PE* de 27 Mo suivi d'une page Web fonctionnera toujours dans les 2 formats. Un outil de sécurité grand public se devra de se limiter *a priori* aux premiers méga-octets, pour optimiser sa performance.

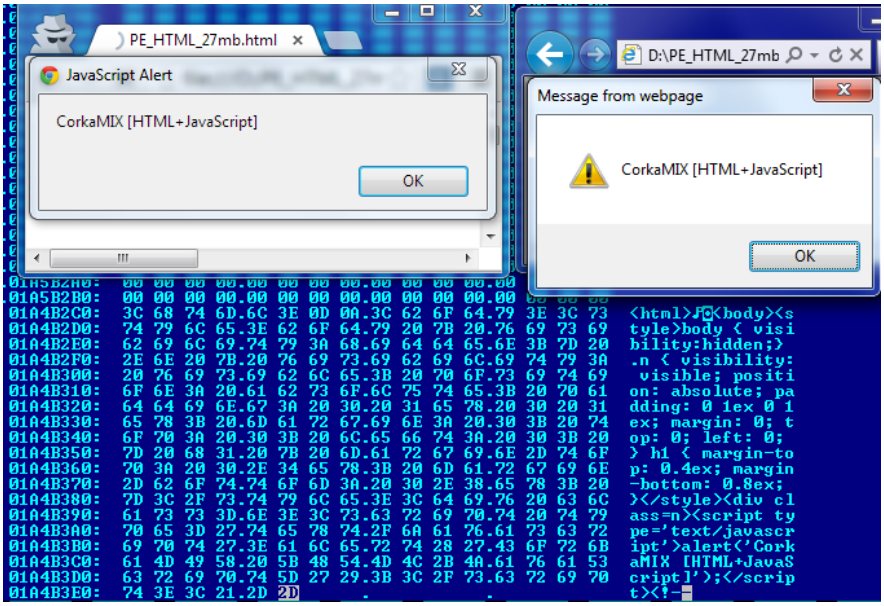

Figure 20. Une page *HTML* à la suite d'un *PE* de 27Mo.

5 Conclusion

Bien que la plupart excluent d'être combinés à d'autres, certains formats de fichiers autorisent des largesses, ce qui rend la combinaison de formats binaires dans un seul fichier possible.

Ceci a des conséquences de sécurité importantes liées à la confusion entre formats : exécution de virus via les navigateurs, exfiltration de fichier, abandon de l'analyse par les outils de sécurité, menant à considérer un fichier malveillant comme sain (de plus, les malformations possibles de ces formats compliquent l'analyse automatisée et manuelle de virus).

Il semble donc indispensable de ne créer que des standards déterminés par une signature unique au début du fichier.

Adobe Reader rejette d'ailleurs, depuis la version 10.1.5 de janvier 2013, tout fichier *PDF* commençant par une signature connue telle que *PE*, *PNG*, *JPG* . . . Il ne devrait pourtant en théorie n'accepter que les *PDF* commençant par une signature *PDF*, mais ce n'est souvent pas le cas dans la pratique.

Les autres visionneuses (*Chrome*, *Sumatra*, *Evince*) acceptent toujours n'importe quel début de fichier. En particulier, *Sumatra* est devenu récemment (version 2.2) plus laxiste en tolérant l'absence de `endobj` après `endstream`, alors qu'*Adobe Reader*, au contraire, et comme on s'y attend, renforce progressivement les contrôles de la structure du document.

Une autre possibilité serait d'imposer au système le type de fichier : quand un fichier est téléchargé, il est étiqueté comme son type original, et ensuite le système impose son type, tant qu'une opération externe de dévérouillage de type n'a pas été effectuée.

De même, l'exécution d'un fichier via les APIs *Windows* n'impose aucune extension, ce qui rend les exécutables — pourtant omniprésents dans les virus — plus facile à lancer, alors qu'il n'y a pas de raison fondamentale qu'on accepte de lancer comme un *PE* un document *PDF* ou une page *HTML*.

Références

1. Elizabeth Scott. Introducing Metalkit. `http://scanlime.org/2008/03/introducing-metalkit/`, 2008.
2. Jonas Magazinius. Polyglots : Crossing Origins by Crossing Formats. submitted at the 22nd USENIX Security Symposium, 2013.
3. Julia Wolf. OMG WTF PDF - What you didn't know about Acrobat. `http://events.ccc.de/congress/2010/Fahrplan/events/4221.en.html`, 2010.
4. Wikipedia. Gifar. `http://en.wikipedia.org/wiki/Gifar`, 2008.