`caradoc` : une boîte à outils pour décortiquer et analyser sereinement les fichiers PDF

Guillaume Endignoux[1,2] Olivier Levillain[3]

[1] EPFL [2] Kudelski Security [3] ANSSI

guillaume.endignoux@m4x.org
olivier.levillain@ssi.gouv.fr

Résumé

PDF est un format de document largement utilisé, complexe, et exploité pour la diffusion de logiciel malveillant. Il semble donc pertinent de l'étudier. Pour compléter la panoplie d'outils existants, nous présentons `caradoc`, une boîte à outils pour disséquer des fichiers PDF de manière robuste et fiable. L'apport de `caradoc` est d'insister sur les aspects bas-niveau de la dissection (*parsing*), là où de nombreux outils partent généralement d'une structure déjà interprétée pour leur analyse. Or l'étape d'interprétation des structures bas-niveau est connue pour être complexe et pour introduire de la confusion dans les lecteurs PDF.

Un article scientifique présentant notre démarche a été publié au troisième *workshop* LangSec en 2016 [7] [1]. L'outil est disponible sur GitHub [2] ainsi que sous la forme d'un paquet Debian [3].

1 Introduction

Depuis près de 25 ans, le format PDF est largement utilisé pour échanger des documents. Derrière l'idée encore répandue qu'il permet uniquement de décrire des ressources graphiques, PDF permet l'inclusion de code (JavaScript), d'objets en tout genre (vidéo, son, visualisation 3D, etc.), et comporte de nombreuses fonctionnalités *intéressantes* : compression, chiffrement, gestion en version du contenu. Le format, décrit dans un standard ISO [8], est donc complexe... et exploité pour la diffusion de logiciel malveillant.

L'outil présenté dans cet article propose une boîte à outils pour décortiquer de manière robuste et fiable les fichiers PDF, y compris au niveau

1. L'article et les planches présentées sont disponibles sur `http://spw16.langsec.org/papers.html#caradoc` et la vidéo sur `https://www.youtube.com/watch?v=VJGL18baEjM`
2. `https://github.com/ANSSI-FR/caradoc`
3. `http://paperstreet.picty.org/~yeye/2017/conf-sstic-EndignouxL17/`

de la structure bas-niveau. En effet, de nombreux outils d'analyse existants commencent leur travail après l'étape de *parsing*, qui est pourtant loin d'être anodine dans le cas de PDF.

`caradoc` est un logiciel libre développé initialement par Guillaume Endignoux dans le cadre d'un stage à l'ANSSI. Le logiciel est disponible sur GitHub, ainsi que comme paquet Debian compatible avec les versions Jessie et Stretch de la distribution (voir section 3.1).

Cet article présente dans un premier temps les motivations qui ont mené au développement de `caradoc` (section 2). Ensuite, la section 3 présente comment installer et utiliser simplement l'outil. Les apports potentiels de `caradoc` à la sécurité sont décrits dans la section 4. Des expérimentations réalisés sur des échantillons de fichiers PDF sont présentés dans la section 5. Enfin, la section 6 décrit les limitations actuelles de l'outil et présente des perspectives.

2 État de l'art et motivation

La structure basique d'un fichier PDF est décrite sur la figure 1. Un fichier PDF se compose de :

- un en-tête, qui contient un marqueur du format et la version ;
- le corps du fichier, qui contient des *objets indirects* ;
- une table de références croisées (*xref table*) qui donne la position des objets dans le fichier ;
- un en-queue (*trailer*) qui indique l'objet à la racine du document ;
- un marqueur de fin de fichier qui indique la position de la table de références croisées.

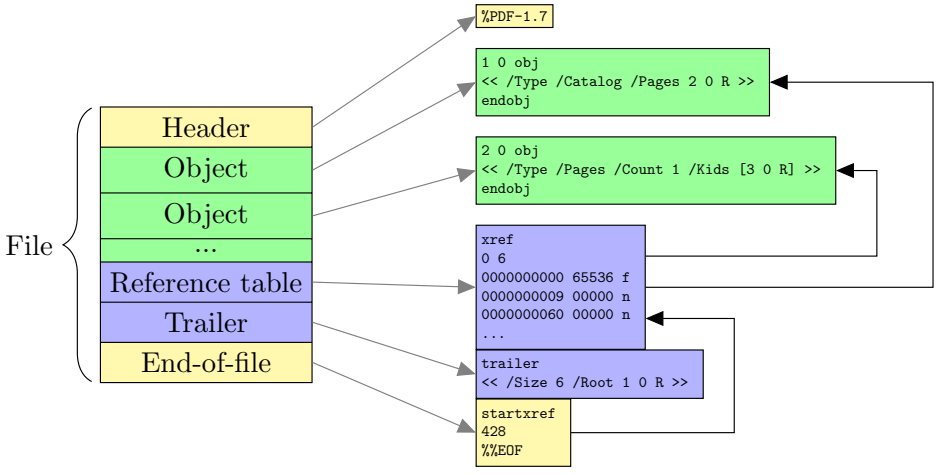

FIGURE 1 – Structure basique d'un fichier PDF.

L'annexe A présente le fichier PDF complet utilisé pour cet exemple. Au-delà de cette structure basique, à la logique déjà tourmentée, la réalité du format est extrêmement complexe, chaque nouvelle version du standard apportant son lot de nouvelles fonctionnalités. On peut citer :

— les objets et tables de références croisées compressés ;
— l'inclusion de scripts, médias et autres objets en tout genre ;
— les mises à jour incrémentales du document ;
— la *linéarisation*, qui ajoute des annotations pour afficher le début du document sans attendre la fin du fichier (utile dans un contexte web).

De plus, la spécification n'est pas toujours claire, et décrit de nombreuses structures redondantes. Il en résulte un flou quant à l'interprétation des fichiers non conformes, ce qui mène les développeurs de lecteurs PDF à accepter tout et n'importe quoi en entrée, en essayant de donner un sens à des structures aberrantes. Cela correspond aux conseils de la spécification, qui recommande de tenter de corriger les erreurs en cas d'incohérence :

> When a conforming reader reads a PDF file with a damaged or missing cross-reference table, it **may attempt** to rebuild the table by scanning all the objects in the file.
>
> — ISO 32000-1-2008 [8], annex C.2

Par le passé, plusieurs travaux ont été présentés sur PDF. Par exemple, les travaux de Frédéric Raynal, Guillaume Delugré et Damien Aumaitre sur les origamis [9] décrivaient des moyens d'injecter des codes malveillants dans des fichiers PDF, en particulier via les champs `Action`.

Plus récemment, Ange Albertini a présenté de nombreux exemples de fichiers PDF exotiques [1, 2, 3, 4, 5]. Certains sont des fichiers polyglottes, qui peuvent être interprétés de plusieurs manières différentes (en tant que PDF, mais aussi en tant qu'exécutables ou qu'image par exemple). D'autres exploitent des disparités entre lecteurs PDF pour adapter le rendu affiché en fonction du lecteur utilisé.

Il existe également des travaux liés à la détection de fichiers PDF malveillants, basés sur l'analyse de contenu. Le fonctionnement classique est d'extraire des éléments caractéristiques des fichiers (*features*) et d'utiliser des approches statistiques pour classifier les documents comme malveillants ou non. Un exemple est PDFRate [10]. Cependant, de tels outils travaillent généralement sur des fichiers déjà interprétés. Or, nous pensons que l'étape de *parsing*, souvent considérée comme inintéressante voire facile, est un sujet important à traiter pour la sécurité.

Avec `caradoc`, notre objectif était de proposer des outils pour analyser de manière fiable et robuste le format PDF, en partant des structures de base. Comme le format PDF est complexe et très riche, nous avons choisi de nous concentrer sur un sous-ensemble restreint du format, décrit dans l'article présenté en 2016 au *workshop* LangSec [7]. Cette restriction permet d'éviter les ambiguïtés introduites par une spécification parfois imprécise. De

plus, si un fichier respecte cette syntaxe restreinte, il est très probable que tous les lecteurs interprètent la même *structure*, ce qui est un bon début.

`caradoc` est une boîte à outils permettant de décortiquer les PDF respectant cette syntaxe restreinte, et dans une certaine mesure de traduire les fichiers raisonnables dans le dialecte restreint. Lorsque la traduction n'est pas possible, il est généralement intéressant d'en investiguer les raisons. La structure construite par `caradoc` permet de plus de réaliser des analyses sur des bases saines. Pour l'instant, les analyses proposées dans le projet concernent le contenu des flux graphiques, mais il est également possible d'extraire les objets pour faire l'analyse de scripts.

3 Prise en main de `caradoc`

3.1 Installation de `caradoc`

Il existe deux manières d'installer l'outil sur votre système Linux : le paquet Debian et la compilation depuis le dépôt GitHub.

Installation depuis le paquet Debian

Le paquet, compilé pour les architectures `amd64`, s'installe fonctionne sur les versions actuelles de Debian (Jessie, Stretch et Sid). Il suffit de télécharger un des fichiers `.deb` depuis l'URL `http://paperstreet.picty.org/~yeye/2017/conf-sstic-EndignouxL17/` puis de l'installer avec `dpkg -i`. Les différentes versions proposées sont décrites dans le tableau ci-dessous :

Nom du fichier et haché	Compatibilité	Interface `curses`

`caradoc_0.3-1_amd64.deb`
`729bfb75f9ddc683a347538bb2ff50f5`
`2e9ea84bc83414033c6439b8f3c25285`
	Jessie, Stretch, Sid	absente

`caradoc_0.3.1-1-jessie_amd64.deb`
`b4d8d9237b85eb13bd58fb63bbf38ec2`
`e941d96392e36ec98e18129a58c41761`
	Jessie	présente

`caradoc_0.3.1-1-stretch_amd64.deb`
`39bf79b378cecd57b029b992e4736c89`
`276809f6b6efe22b6c80dbd8a7bf904d`
	Stretch, Sid	présente

Installation depuis les sources

Le dépôt GitHub du projet est disponible à l'adresse `https://github.com/ANSSI-FR/caradoc`. À partir de là, il est possible de consulter la documentation concernant l'installation.

Sous Debian, en utilisant le système de gestionnaires de paquets OPAM d'OCaml, les commandes à exécuter sont les suivantes :

```
apt-get install ocaml opam zlib1g-dev
opam init
opam install ocamlfind cryptokit ounit menhir curses
make
BINDIR=/repertoire/ou/installer/ make install
```

3.2 Manipulations en ligne de commande

Aperçu d'un fichier

Pour obtenir quelques statistiques simples sur un fichier PDF, il suffit d'utiliser la commande suivante :

```
$ caradoc stats input.pdf
```

Vous obtiendrez alors un résultat similaire à la sortie suivante :

```
Version : 1.6
Incremental updates : 0
Neither updates nor object streams nor free objects nor encryption
Object count : 158
Filter : FlateDecode -> 39 times
Filter : DCTDecode -> 3 times
Filter : Raw -> 1 times
Objects of known type : 139
Known type rate : 0.879747
Some types were not fully checked
Graph has no known error
Content streams have no known error
/Producer : (GPL Ghostscript 9.20)
/Creator : (TeX)
/CreationDate : (D:20170125191914+01'00')
/ModDate : (D:20170125191914+01'00')
/ID : <345D4F447BF0A005A40F4612789F416F>
```

Validation du fichier

En ajoutant l'option `--strict` à la commande précédente, vous pouvez valider que le fichier est bien construit, et qu'il est conforme à la structure restreinte que nous avons choisie. Dans l'exemple précédent, un avertissement est levé lors de l'utilisation du mode strict :

```
Warning : Flate/Zlib stream with appended newline in object 38
```

Cet avertissement est courant, en l'occurrence un flux compressé contient un retour à la ligne supplémentaire avant la balise `endstream`.

Normalisation d'un fichier

Il est généralement possible de normaliser un fichier pour qu'il respecte le sous-ensemble restreint de la spécification que nous avons choisi. Pour cela, il est nécessaire que le fichier en entrée ne soit pas trop éloigné du sous-ensemble de fonctionnalités.

Il est ainsi possible d'aplatir un fichier contenant des mises à jour, de supprimer les objets inutilisés ou de corriger des erreurs classiques introduites par certaines implémentations. Cependant, certaines erreurs introduisent trop d'ambiguité et ne sont pas corrigées.

La normalisation se fait avec la commande suivante :

```
$ caradoc cleanup input.pdf --out output.pdf
```

Extraction d'éléments précis

Une fois le document interprété, `caradoc` peut extraire certains éléments du fichier :

— la table de références croisées, avec la commande `xref` ;
— l'en-queue du fichier, avec la commande `trailer` ;
— un objet spécifique du fichier, à partir de son numéro (et optionnellement de son numéro de génération), via la commande `object` ;

La commande `extract` permet d'extraire toutes ces informations à la fois. L'option `--dot` produit un graphe des objets, dont les arêtes sont les références entre objets (un exemple est donné dans l'annexe A).

```
$ caradoc extract --xref <xref output file>
                  --dump <objects output file>
                  --types <types output file>
                  --dot <graph output file>
                  input.pdf
```

Enfin, les commandes `findref` et `findname` permettent de trouver l'ensemble des références à un objet ou les occurrences d'un *nom*.

3.3 Interface interactive

Dans la version la plus récente, `caradoc` intègre une interface interactive en console, reposant sur la bibliothèque `ncurses` (disponible sur Linux).

```
$ caradoc ui file.pdf
```

Une notice d'aide s'affiche alors à l'écran, pendant que le fichier est chargé en mémoire en arrière-plan. Quelques commandes simples permettent de naviguer rapidement à travers le fichier. L'écran peut se diviser en plusieurs vues, ce qui facilite par exemple la comparaison entre objets.

L'ensemble des commandes disponibles dans l'interface en ligne de commande sont décrites dans le tableau suivant :

Commande	Action
`q`	Quite la vue en cours.
`s`	Duplique la vue en cours.
$\leftarrow, \rightarrow$	Change de vue.
`f`	Affiche le fichier brut en hexadécimal.
`i`	Affiche les statistiques du fichier.
`t`	Affiche le *trailer*.
`123 o`	Affiche l'objet numéro 123.
`d`	Décode le *flux* associé à l'objet en cours.
`123 r`	Cherche toutes les références à l'objet numéro 123.
`<ENTER>`	Affiche le résultat de recherche sélectionné.
$\uparrow, \downarrow, \Uparrow, \Downarrow, \nwarrow$, `<FIN>`	Navigue dans la vue en cours.
`123 g`	Avance à la ligne/octet 123.

4 Applications de `caradoc` à la sécurité

4.1 Validation et nettoyage

La première application de `caradoc` est la possibilité de valider et de normaliser des fichiers PDF. En effet, en plus d'interpréter la structure bas niveau du fichier, `caradoc` utilise un algorithme de typage pour s'assurer que les objets sont conformes à leur type, et que le graphe des objets est cohérent.

En 2015, nous avons ainsi identifié plusieurs problèmes dans les lecteurs PDF. Par exemple, l'ensemble des pages d'un fichier est décrit comme un arbre (ce qui semble déjà plus compliqué que l'idée naturelle d'utiliser une liste), mais les champs utilisés dans les objets pour décrire ces pages autorisent en réalité à décrire n'importe quel graphe. Il est donc possible de créer des cycles, ce qui pose problème à certains outils [4].

Le problème avait déjà été présenté en 2014 par Bogk et Schopl, ayant voulu écrire un *parser* PDF en Coq pour en prouver certaines propriétés [6]. Pour pouvoir prouver la correction de leur *parser*, il leur fallut ajouter des contraintes empêchant les cycles (mais dans un contexte différent que l'arbre des pages).

`caradoc` permet également de détecter les objets non référencés (qui permettent par exemple de stocker de manière cachée de l'information).

De manière anecdotique, notre *parser* strict a permis de mettre en évidence des fautes d'orthographe dans certains PDF. En effet, les champs

4. Des exemples de fichiers mal formés sont disponibles dans le dépôt GitHub (répertoire `test_files/negative`). Certains de ces fichiers déclenchent des erreurs dans des lecteurs courants et ont fait l'objet de remontées de *bugs*.

inconnus sont généralement ignorés (ou corrigés silencieusement) par la plupart des lecteurs, mais notre approche stricte permet de les mettre au jour : `/Blackls1` à la place de `/BlackIs1`, `/XObjcect` à la place de `/XObject`.

4.2 Analyses de plus haut niveau

Grâce à l'extraction fiable des objets, des analyses complémentaires sur le contenu deviennent possibles. Il est intéressant de constater que la majorité des travaux sur PDF s'intéressent aux documents une fois interprétés. Cela signifie que certaines techniques d'évasion sont possibles si l'interprétation bas-niveau de la structure n'est pas faite de manière fiable. Par exemple, PDFRate est facilement contournable [11] en ajoutant des objets non référencés ou en altérant les *xref tables*. De même, les outils développés par Didier Stevens [5], ne sont pas très robustes : d'après l'auteur, `pdf-parser.py` est un *parser quick and dirty* qui se contente de faire le boulot [6], et `pdfid` cherche des mots clés sans être un *parser* à part entière [7].

À l'inverse, le *parser* de `caradoc` détecte les incohérences et les ambiguïtés, décompresse les flux et les déchiffre lorsque le mot de passe lui est fourni (voir section 5.1). Cependant, ces analyses nécessiteront des développements supplémentaires pour se brancher sur la structure obtenue par `caradoc` (voir section 6).

5 Analyse d'un échantillon de fichiers PDF

Nous avons testé `caradoc` sur un jeu de fichiers représentatifs : nous avons téléchargé 10000 fichiers PDF distincts via des recherches aléatoires sur un moteur de recherche. Les analyses de `caradoc` sur ce jeu de fichiers permettent de donner une idée du paysage des fichiers PDF communs.

Nous avons d'abord obtenu quelques statistiques générales sur la validation, avec ou sans normalisation. En utilisant la validation stricte directement, seulement 1465 fichiers sont parsés, dont 548 sont correctement typés. Cela vient du fait que le *parser* strict rejette de nombreuses constructions courantes mais complexes (mises-à-jour incrémentales, linéarisation, chiffrement). Finalement, 457 fichiers sont également validés au niveau du graphe des objets (absence de cycles dans les arbres) et des instructions graphiques.

En appliquant la normalisation avant de valider, beaucoup plus de fichiers passent les vérifications de `caradoc`. Ainsi, 9829 fichiers sont normalisés, après quoi 2105 fichiers sont correctement typés et 1891 passent tous les tests. Cependant, de nombreux fichiers non-validés sont en fait simplement

5. `https://blog.didierstevens.com/programs/pdf-tools/`

6. « The code of the parser is quick-and-dirty, I'm not recommending this as text book case for PDF parsers, but it gets the job done. »

7. « This tool is not a PDF parser, but it will scan a file to look for certain PDF keywords »

partiellement typés, et seulement 1575 fichiers déclenchent une erreur de typage. En effet, la spécification définit un très grand nombre de types, ce qui représente un travail conséquent à transcrire dans `caradoc`, notamment à cause des formulations parfois ambiguës de la spécification. Malgré tout, de nouveaux types sont régulièrement ajoutés à `caradoc` et vous pouvez obtenir la listes des types grâce à la commande suivante.

```
$ caradoc types
```

5.1 Chiffrement

Nous avons également étudié une fonctionnalité intéressante : le chiffrement. En effet, PDF intègre un méchanisme de chiffrement ad-hoc, à partir de deux mots de passe. Le mot de passe *utilisateur* permet de déchiffrer le contenu et visualiser le fichier, tandis que le mot de passe *propriétaire* permet de débloquer des *permissions* (impression, modification, copie, etc.). Parmi notre jeu de fichiers, 478 sont chiffrés, mais avec un mot de passe utilisateur vide. En pratique les lecteurs PDF testent le mot de passe vide et ouvrent le fichier de façon transparente le cas échéant. Ce type de fichier sert à bloquer des permissions ou à obfusquer le contenu.

Diverses clés de chiffrement sont dérivées depuis ces mots de passe, selon un mécanisme ad-hoc [8] utilisant essentiellement les algorithmes MD5 et RC4. Le fichier contient également un identifiant unique qui complète les mots de passe dans la dérivation des clés, ainsi que des sommes de contrôle pour vérifier que l'utilisateur a entré le bon mot de passe. Or, il se trouve que la somme de contrôle `/O` (pour *owner*) est dérivée uniquement des deux mots de passe mais pas de l'identifiant unique ! En d'autres termes, il s'agit d'un haché non salé des deux mots de passe (avec une fonction de hachage ad-hoc).

Il est possible d'extraire ce champ `/O` avec `caradoc`, via la commande `caradoc stats`. Nous avons donc pu vérifier si plusieurs fichiers possèdent le même champ `/O`, ce qui correspond à des mots de passe identiques. En effet, parmi les 478 fichiers chiffrés de notre échantillon, 157 fichiers partagent leur somme `/O` avec au moins un autre fichier, soit au total 33 % qui sont en collision ! Dans ce contexte, il va de soi qu'un attaquant pourrait utiliser des techniques plus poussées (*rainbow tables*) pour retrouver les mots de passe, ce qui affaiblit ce système de chiffrement.

6 Limitations et perspectives

Bien que déjà bien établi, le projet `caradoc` comporte encore quelques limitations que nous prévoyons de corriger à moyen terme.

8. `https://gendignoux.com/blog/2016/11/02/pdf-encryption.html`

Tout d'abord, certains objets (les *flux*) peuvent être compressés ou encodés avec une dizaine de filtres, tels que Deflate, JPEG ou encore un simple encodage hexadécimal. `caradoc` implémente déjà quatre algorithmes (ainsi que le déchiffrement) pour décoder ces objets. Nous prévoyons de compléter ce panel, notamment car certains fichiers malveillants utilisent des filtres exotiques pour obfusquer leur contenu.

Par ailleurs, nous envisageons d'ajouter un module spécifique pour extraire le code JavaScript éventuellement présent. En effet, de nombreuses vulnérabilités découvertes dans Adobe Reader concernent son moteur d'interprétation JavaScript, et un tel module faciliterait donc les analyses de plus haut niveau. Notre parseur avancé et notre algorithme de typage peuvent offrir plus de garanties quant à l'exhaustivité du contenu JavaScript extrait, par rapport à des parseurs minimalistes (du type "grep Javascript") qui passent à côté de structures complexes comme les objets compressés. Enfin, nous prévoyons d'affiner les expérimentations sur de plus gros volumes de fichiers et sur d'autres échantillons.

Références

[1] Ange Albertini. Polyglottes binaires et implications. In *SSTIC*, 2013.

[2] Ange Albertini. This OS is also a PDF. *PoC or GTFO 0x02*, 2013.

[3] Ange Albertini. This PDF is a JPEG ; or, this proof of concept is a picture of cats. *PoC or GTFO 0x03*, 2014.

[4] Ange Albertini. This TAR archive is a PDF ! *PoC or GTFO 0x06*, 2014.

[5] Ange Albertini. Abusing file formats ; or, Corkami, the Novella. *PoC or GTFO 0x07*, 2015.

[6] Andreas Bogk and Marco Schopl. The Pitfalls of Protocol Design : Attempting to Write a Formally Verified PDF Parser. In *Security and Privacy Workshops (SPW), 2014 IEEE*, pages 198–203. IEEE, 2014.

[7] Guillaume Endignoux, Olivier Levillain, and Jean-Yves Migeon. Caradoc : a pragmatic approach to PDF parsing and validation. In *37. IEEE Security and Privacy Workshops, SPW (LangSec) 2016, San Jose, CA, USA*, pages 126–139, May 2016.

[8] ISO. Document management—Portable document format—Part 1 : PDF 1.7. ISO 32000–1 :2008, International Organization for Standardization, Geneva, Switzerland, 2008. `http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/PDF32000_2008.pdf`.

[9] Frédéric Raynal, Guillaume Delugré, and Damien Aumaitre. Malicious origami in PDF. *Journal in computer virology*, 6(4) :289–315, 2010.

[10] Charles Smutz and Angelos Stavrou. Malicious PDF detection using metadata and structural features. In *28th Annual Computer Security Ap-*

plications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012, pages 239–248, 2012.

[11] Nedim Srndic and Pavel Laskov. Practical evasion of a learning-based classifier : A case study. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 197–211, 2014.

A Exemple de fichier PDF complet

A.1 Contenu du fichier

```
%PDF-1.5

1 0 obj <<
  /Type /Catalog
  /Pages 2 0 R
>> endobj

2 0 obj <<
  /Type /Pages
  /Count 1
  /Kids [3 0 R]
>> endobj

3 0 obj <<
  /Type /Page
  /MediaBox [0 0 700 200]
  /Resources <<
    /Font <<
      /F1 5 0 R
    >>
  >>
  /Parent 2 0 R
  /Contents 4 0 R
>> endobj

4 0 obj <<
  /Length 35
>> stream
BT /F1 100 Tf (Hello world !) Tj ET
endstream
endobj

5 0 obj <<
  /Type /Font
  /Subtype /Type1
  /BaseFont /Helvetica
  /Name /F1
>> endobj

6 0 obj <<
  /Author (Guillaume Endignoux)
>> endobj

xref
0 7
0000000000 65535 f
```

```
0000000010 00000 n
0000000064 00000 n
0000000128 00000 n
0000000281 00000 n
0000000369 00000 n
0000000458 00000 n
trailer
<<
  /Size 7
  /Root 1 0 R
  /Info 6 0 R
>>
startxref
512
%%EOF
```

A.2 Explications

Le fichier `hello.pdf` [9] correspond au format basique décrit à la figure 1. Il contient les éléments suivants :

- Ligne 1 : l'en-tête indique qu'il s'agit d'un fichier PDF en version 1.5.
- Lignes 3 à 6 : l'objet numéro 1 est le catalogue. Il annonce que l'objet 2 est la racine de l'arbre des pages.
- Lignes 8 à 12 : l'objet numéro 2 est la racine de l'arbre des pages. Ici, l'arbre se résume à une feuille unique, l'objet numéro 3.
- Lignes 14 à 24 : l'objet numéro 3 est l'unique page du document. La ligne 16 indique les dimentsions de la page. Les lignes 17 à 21 indiquent que la page utilise une police définie dans l'objet 5 et qui peut être utilisée sous le nom `/F1`. La ligne 23 indique que le contenu graphique de la page est défini par l'objet 4
- Lignes 26 à 31 : l'objet numéro 4 contient une séquence de commandes graphiques, stockées à l'intérieur d'un flux (*stream*). La ligne 29 contient trois commandes. En particulier, `/F1 100 Tf` permet l'utilisation de la police `/F1` avec une taille de 100 ; `(Hello world !) Tj` affiche le texte `Hello world !`.
- Lignes 33 à 38 : l'objet numéro 5 décrit une police. Il s'agit en l'occurrence de la police `Helvetica`, une des polices standard définies par le format. Ainsi, il n'est pas nécessaire d'intégrer la police à l'intérieur du PDF.
- Lignes 40 à 42 : l'objet numéro 6 contient les méta-données du document, ici le nom de l'auteur.
- Lignes 44 à 52 : la table de références croisées donne la position des objets dans le fichier. La ligne 45 annonce une table à 7 objets, en partant de l'objet numéro zero. Ensuite, pour chaque objet, la première colonne contient l'offset depuis le début du fichier de l'objet ; la seconde colonne contient le numéro de génération et la troisième

9. `https://github.com/ANSSI-FR/caradoc/raw/master/test_files/positive/hello/hello.pdf`

indique si l'objet est utilisé (`n`) ou non (`f`).

— Lignes 53 à 58 : le *trailer* indique qu'il existe 7 objets (en comptant l'objet numéro zero, un objet inutilisé obligatoire). Il précise que le catalogue est l'objet numéro 1 et que les méta-données sont situées dans l'objet numéro 6.

— Lignes 59 and 61 : le marqueur de fin de fichier donne la position absolue de la table de références croisées dans le fichier.

A.3 Graphe associé

La figure 2 illustre les relations entre les objets du fichier. Elle a été produite à l'aide de l'option `--dot` de la commande `extract`.

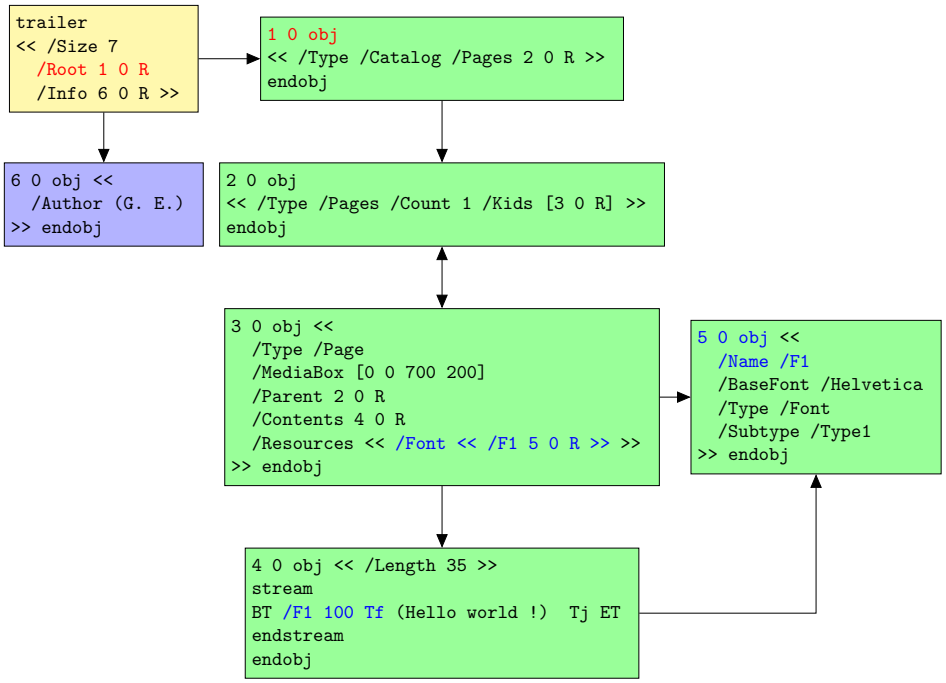

FIGURE 2 – Graphe décrivant le fichier simple.