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

Guillaume Endignoux[1] et Olivier Levillain[2]
guillaume.endignoux@m4x.org
olivier.levillain@ssi.gouv.fr

[1] EPFL
[2] ANSSI

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, etc.), et comporte de nombreuses fonctionnalités *intéressantes* : compression, chiffrement, gestion en version du contenu. Le format, décrit dans un standard ISO [5], 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 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 projet de logiciel libre développé initialement par Guillaume Endignoux dans le cadre d'un stage à l'ANSSI. Le logiciel est disponible sur GitHub.

Cet article présente dans un premier temps les motivations qui ont mené au développement de `caradoc` (section 1). Ensuite, la section 2 présente comment installer et utiliser simplement l'outil. Les apports potentiels de `caradoc` à la sécurité sont décrits dans la section 3. Des expérimentations réalisés sur des échantillons de fichiers PDF sont présentés dans la section 4. Enfin, la section 5 décrit les limitations actuelles de l'outil et présente des perspectives. Une version longue est disponible sur le site de la conférence[3].

1 État de l'art et motivation

La structure basique d'un fichier PDF est composée de :

- un en-tête, qui contient un marqueur du format et la version ;
- le corps du fichier, qui contient des *objets indirects* ;

[3] `https://www.sstic.org/2017/presentation/caradoc/`

- 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.

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 [5], annex C.2

Par le passé, plusieurs travaux ont été présentés sur PDF. Par exemple, les travaux de Raynal, Delugré et Aumaitre sur les origamis [6] décrivaient des moyens d'injecter des codes malveillants dans des fichiers PDF, en particulier via les champs `Action`.

Plus récemment, Albertini a présenté de nombreux exemples de fichiers PDF exotiques [1, 2]. 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 [7]. 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.

Notre objectif était de proposer des outils pour analyser de manière fiable et robuste le format PDF, en partant des structures de base. Comme ce format 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 [4]. 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.

2 Prise en main de `caradoc`

2.1 Installation de `caradoc`

Il existe deux manières d'installer l'outil sur votre système Linux : des paquets Debian pour l'architecture `amd64`[4], ou la compilation depuis le dépôt GitHub[5]. La version longue de l'article et le dépôt donnent plus de détails concernant l'installation.

2.2 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
Version : 1.6
Incremental updates : 0
Neither updates nor object streams nor free objects nor encryption
Object count : 158
Filter : FlateDecode -> 39 times
...
```

[4] `http://paperstreet.picty.org/~yeye/2017/conf-sstic-EndignouxL17/`
[5] `https://github.com/ANSSI-FR/caradoc`

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'ambiguïté 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.

```
$ 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*.

2.3 Interface interactive

Dans la version la plus récente, `caradoc` intègre une interface interactive en console, reposant sur la bibliothèque `ncurses` :

```
$ 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.

3 Applications de `caradoc` à la sécurité

3.1 Validation et nettoyage

La première application de `caradoc` est la validation et la normalisation de 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[6].

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 [3]. 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 mis en évidence des fautes d'orthographe dans certains PDF. En effet, les champs 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`.

[6] 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*.

3.2 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 [8] en ajoutant des objets non référencés ou en altérant les *xref tables*. De même, les outils développés par Didier Stevens[7], 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[8], et `pdfid` cherche des mots clés sans être un *parser* à part entière[9].

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 4.1). Cependant, ces analyses nécessiteront des développements supplémentaires pour se brancher sur la structure obtenue par `caradoc` (voir section 5).

4 Analyse d'un échantillon de fichiers PDF

Nous avons testé `caradoc` sur un jeu de fichiers représentatif : nous avons téléchargé 10 000 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 1 465 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

[7] `https://blog.didierstevens.com/programs/pdf-tools/`

[8] "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."

[9] "This tool is not a PDF parser, but it will scan a file to look for certain PDF keywords."

simplement *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 la liste des types supportés est donné par la commande `caradoc types`.

4.1 Chiffrement

Nous avons également étudié une fonctionnalité intéressante : le chiffrement. En effet, PDF intègre un mécanisme 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.). Notre jeu de données contient 478 fichiers 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 masquer le contenu.

Diverses clés de chiffrement sont dérivées depuis ces mots de passe, selon un mécanisme ad-hoc[10] 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.

5 Limitations et perspectives

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

[10] `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 dissimuler 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 *parser* avancé et notre algorithme de typage peuvent offrir plus de garanties quant à l'exhaustivité du contenu JavaScript extrait, par rapport à des *parsers* 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 TAR archive is a PDF ! *PoC or GTFO 0x06*, 2014.
3. 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.
4. 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.
5. 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`.
6. Frédéric Raynal, Guillaume Delugré, and Damien Aumaitre. Malicious origami in PDF. *Journal in computer virology*, 6(4) :289–315, 2010.
7. Charles Smutz and Angelos Stavrou. Malicious PDF detection using metadata and structural features. In *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, pages 239–248, 2012.
8. 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.