Diving into the Portable Document Format

Toulouse Hacking Convention 2017

Guillaume Endignoux
@gendignoux

Friday 3$^{\text{rd}}$ March, 2017

Portable Document Format ?

PDF timeline:

- 1991-1993: inception and first release by Adobe[1]
- 2008: ISO specification released (PDF 1.7) ⇒ alternative readers: Evince, PDF.js, Chrome...
- Soon? ISO specification for PDF 2.0

[1] https://acrobat.adobe.com/us/en/why-adobe/about-adobe-pdf.html

Portable Document Format ?

PDF timeline:

- 1991-1993: inception and first release by Adobe[1]
- 2008: ISO specification released (PDF 1.7) ⇒ alternative readers: Evince, PDF.js, Chrome...
- Soon? ISO specification for PDF 2.0

Many features (not all portable):

- interactive forms
- encryption
- scripting: JavaScript, Flash
- multimedia: video, sound, 3D artwork
- ...

[1] https://acrobat.adobe.com/us/en/why-adobe/about-adobe-pdf.html

Portable Document Format ?

A commonly used format, but many security issues:

- 500+ reported vulnerabilities in Adobe Reader[2] (since 1999).
- Variations between implementations.
- Syntax facilitates polymorphism, e.g. PoC||GTFO (PDF+ZIP, PDF+JPEG...).
- SHA-1 collisions...

I worked on PDF validation: Caradoc[3] project started in 2015 (at ANSSI), paper & presentation at LangSec Workshop 2016[4].

[2] http://www.cvedetails.com

[3] https://github.com/ANSSI-FR/caradoc

[4] http://spw16.langsec.org/

Table of contents

1. Introduction to PDF syntax
2. Security problems: case studies
3. Caradoc: 2 years of PDF validation

Table of contents

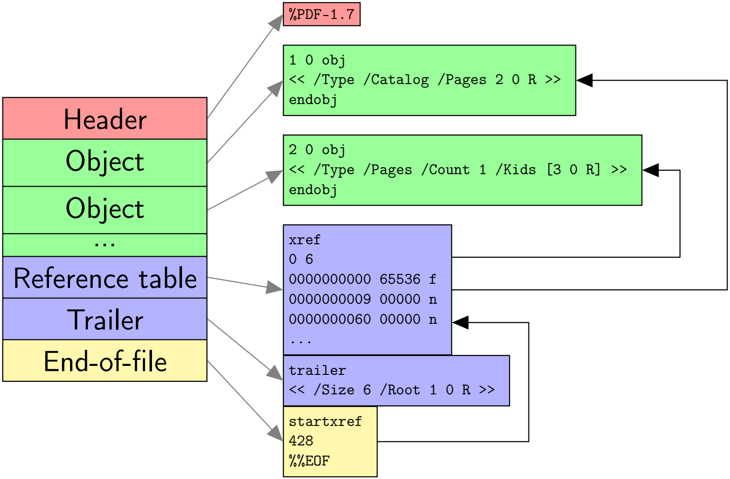
1. Introduction to PDF syntax
2. Security problems: case studies
3. Caradoc: 2 years of PDF validation

PDF syntax 101

A PDF document is made of objects. Textual format, similar to JSON but different syntax:

- `null`
- booleans: `true`, `false`
- numbers: `123`, `-4.56`
- strings: `(foo)`
- names: `/bar`
- arrays: `[1 2 3]`, `[(foo) /bar]`
- dictionaries: `<< /key (value) /foo 123 >>`
- references: `1 0 obj ... endobj` and `1 0 R`
- streams: `<< ... >> stream ... endstream`

Structure of a PDF file



Header

Object

Object

...

Reference table

Trailer

End-of-file

```
%PDF-1.7
```

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

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

```
xref
0 6
0000000000 65536 f
0000000009 00000 n
0000000060 00000 n
...
```

```
trailer
<< /Size 6 /Root 1 0 R >>
```

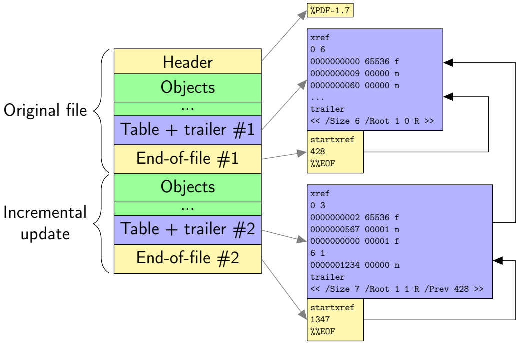
```
startxref
428
%%EOF
```

Organization of a simple PDF file.

Structure of a PDF file

More complex structures:

- incremental updates,
- object streams,
- linearization.



```
%PDF-1.7
```

```
xref
0 6
0000000000 65536 f
0000000009 00000 n
0000000060 00000 n
...
trailer
<< /Size 6 /Root 1 0 R >>
```

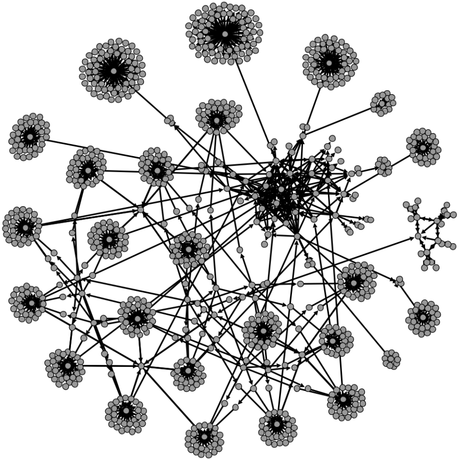
```
startxref
428
%%EOF
```

```
xref
0 3
0000000002 65536 f
0000000567 00001 n
0000000000 00001 f
6 1
0000001234 00000 n
trailer
<< /Size 7 /Root 1 1 R /Prev 428 >>
```

```
startxref
1347
%%EOF
```

Incremental update.

Logical structure of a PDF file



Document of 17 pages (about 1000 objects).

Graphical instructions

Vector graphics = low-level instructions, stored in a *stream*. Some examples:

- set font ABC in size 10: `/ABC 10 Tf`
- set blue color (RGB): `0 0 1 rg`
- draw text: `(Hello world) Tj`
- move to $(x, y) = (5, 10)$: `5 10 m`
- draw line to $(15, 20)$: `15 20 l`
- ...

I made a cheat sheet:
`https://github.com/gendx/pdf-cheat-sheets`

Draw your own PDF!

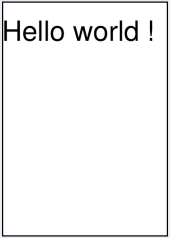
Creating reference tables/streams is error-prone and boring...

Python script to automate the process:
https://github.com/gendx/pdf-corpus

Source

```
template = contentstream
---
BT
0 700 Td
/F1 100 Tf
(Hello world !) Tj
ET
```

Resulting PDF



Hello world !

Table of contents

1. Introduction to PDF syntax
2. Security problems: case studies
3. Caradoc: 2 years of PDF validation

Security problems: case studies

Security problems arise from:

- unclear or ambiguous specification,
- complex or flawed designs in the standard,
- improper input checking by PDF readers.

Security problems: case studies

Security problems arise from:

- unclear or ambiguous specification,
- complex or flawed designs in the standard,
- improper input checking by PDF readers.

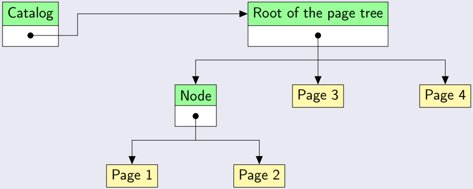
Some case studies:

- malicious graph structures,
- graphics instructions,
- home-made encryption.

Graph organization

The graph of objects is organized into sub-structures, especially trees.

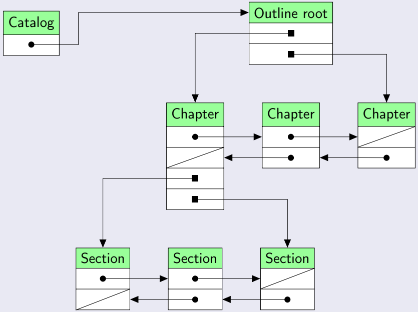
Page tree.



Graph organization

The table of contents uses doubly-linked lists.

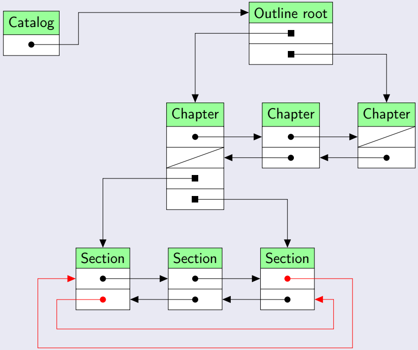
Table of contents.



Problematic structure

Some PDF readers loop forever with an invalid structure...

Invalid table of contents.



Problematic structure

This is a design flaw:

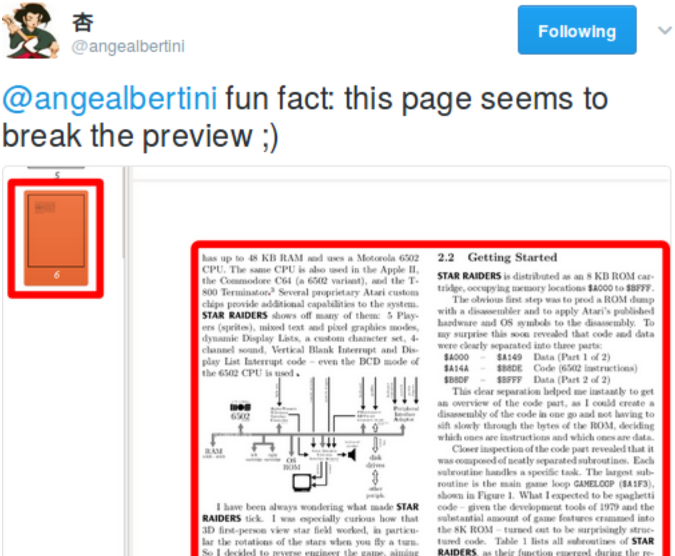
- Complex structures everywhere, but PDF readers do not check them...
- Simpler design: array of references to store pages?

Graphics instructions

Graphics instructions = core of the format ⇒ potential for many bugs!

Graphics instructions

Graphics instructions = core of the format ⇒ potential for many bugs!



杏
@angealbertini

@angealbertini fun fact: this page seems to break the preview ;)

has up to 48 KB RAM and uses a Motorola 6502 CPU. The same CPU is also used in the Apple II, the Commodore C64 (a 6502 variant), and the T-800 Terminator.[3] Several proprietary Atari custom chips provide additional capabilities to the system. **STAR RAIDERS** shows off many of them: 5 Players (sprites), mixed text and pixel graphics modes, dynamic Display Lists, a custom character set, 4-channel sound, Vertical Blank Interrupt and Display List Interrupt code – even the BCD mode of the 6502 CPU is used.

I have been always wondering what made **STAR RAIDERS** tick. I was especially curious how that 3D first-person view star field worked, in particular the rotations of the stars when you fly a turn. So I decided to reverse engineer the game, aiming

2.2 Getting Started

STAR RAIDERS is distributed as an 8 KB ROM cartridge, occupying memory locations `$A000` to `$BFFF`.

The obvious first step was to prod a ROM dump with a disassembler and to apply Atari's published hardware and OS symbols to the disassembly. To my surprise this soon revealed that code and data were clearly separated into three parts:

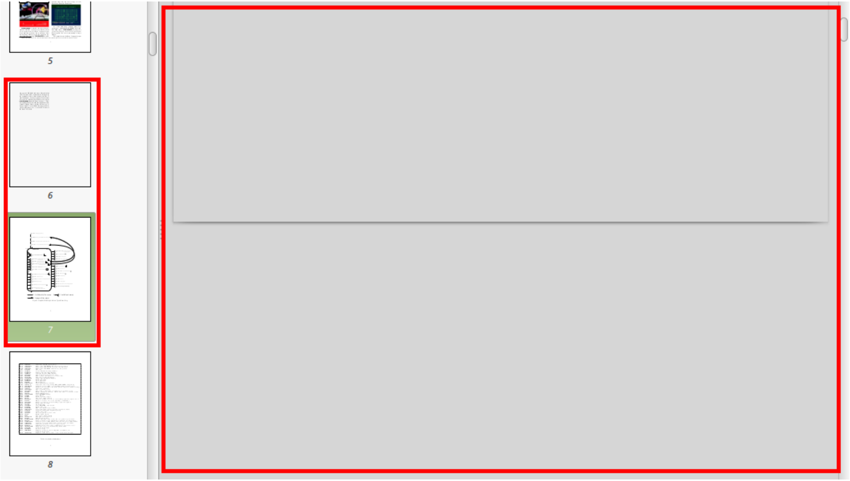
`$A000`	–	`$A149`	Data (Part 1 of 2)
`$A14A`	–	`$B8DE`	Code (6502 instructions)
`$B8DF`	–	`$BFFF`	Data (Part 2 of 2)

This clear separation helped me instantly to get an overview of the code part, as I could create a disassembly of the code in one go and not having to sift slowly through the bytes of the ROM, deciding which ones are instructions and which ones are data.

Closer inspection of the code part revealed that it was composed of neatly separated subroutines. Each subroutine handles a specific task. The largest subroutine is the main game loop `GAMELOOP ($A1F3)`, shown in Figure 1. What I expected to be spaghetti code – given the development tools of 1979 and the substantial amount of game features crammed into the 8K ROM – turned out to be surprisingly structured code. Table 1 lists all subroutines of **STAR RAIDERS**, as their function emerged during the re-

Graphics instructions

I tried to write a PDF optimizer, and found more weird bugs...



Graphics instructions

What is in the graphics interpreter?

A simple example:

- **Graphics state** = font, colors, translations, etc. (e.g. font modified by `setfont`, used by `drawtext`).
- **Graphics state stack**: `push` and `pop` operators to save & restore graphics state.

What if we `pop` too much (stack underflow)?

Graphics instructions

Example[5] for Evince: unbalanced pop seems to stop the interpreter.

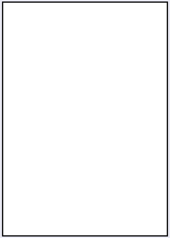
Pseudo-code: pop before

```
pop
setfont
drawtext (Hello world !)
```

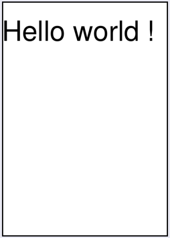
Pseudo-code: pop after

```
setfont
drawtext (Hello world !)
pop
```

PDF



PDF



Hello world !

[5] https://github.com/gendx/pdf-corpus/tree/master/corpus/contentstream/graphic-stack

Demonstration

Loop in the outline structure

https://github.com/ANSSI-FR/caradoc/blob/master/test_files/negative/outlines/cycle.pdf

Polymorphic file

https://github.com/ANSSI-FR/caradoc/blob/master/test_files/negative/polymorph/polymorph.pdf

Poc||GTFO 0x13

https://www.alchemistowl.org/pocorgtfo/pocorgtfo13.pdf

Demonstration

These problems may lead to several attacks:

- Attacks against the parser: denial of service, crash (or worse).
- Evasion techniques: variations PDF reader vs. malware detector.

Encryption

PDF encryption supported since v1.1.

Encryption

PDF encryption supported since v1.1.

Based on 2 passwords.

- **User** password P_u: decrypt and view content.
- **Owner** password P_o: unlock *permissions* (print, modify...) $\Rightarrow$ enforced only by compliant software (P_u is enough to decrypt).

Encryption

PDF encryption supported since v1.1.

Based on 2 passwords.

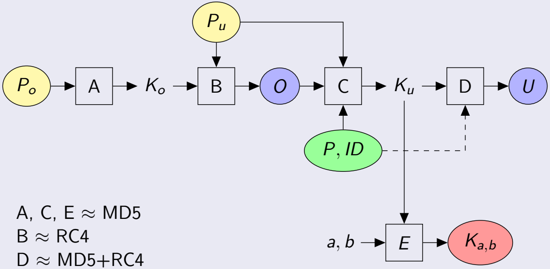
- **User** password P_u: decrypt and view content.
- **Owner** password P_o: unlock *permissions* (print, modify...) ⇒ enforced only by compliant software (P_u is enough to decrypt).

Security issues:

- **Partial encryption**: only *strings* and *streams* are encrypted, general document structure is leaked...
- **Ad-hoc key-derivation** from passwords & checksums (based on MD5+RC4).

Home-made encryption

Complex derivation of keys from passwords.



A, C, E ≈ MD5
B ≈ RC4
D ≈ MD5+RC4

🟡 password 🔵 checksum (in file) 🟢 salt (in file) 🔴 object key

Main problem: checksum O is deterministic function of passwords, no salt! ⇒ 33% collisions for 478 files crawled from Internet...

Table of contents

1. Introduction to PDF syntax
2. Security problems: case studies
3. Caradoc: 2 years of PDF validation

Caradoc validation

I worked on Caradoc, a PDF validator. Implementation in OCaml from the PDF specification[6].

Caradoc verifies the following:

- File syntax.
- Objects consistency (type checking).
- Graph (page tree...).
- Vector graphics instructions (syntax).

[6] https://www.adobe.com/devnet/pdf/pdf_reference.html

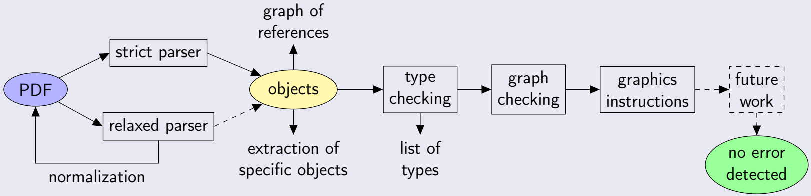
Caradoc validation

I worked on Caradoc, a PDF validator. Implementation in OCaml from the PDF specification[6].

Caradoc verifies the following:

- File syntax.
- Objects consistency (type checking).
- Graph (page tree...).
- Vector graphics instructions (syntax).

Validation workflow.



[6] https://www.adobe.com/devnet/pdf/pdf_reference.html

Syntax restriction

At **syntax** level, guarantee extraction of objects without ambiguity:

- Grammar formalization[7] (BNF).
- Structure restrictions (no updates, no *linearization*, etc.).
- Systematic rejection of "corrupted" files.

[7] `https://github.com/ANSSI-FR/caradoc/tree/master/doc/grammar`

Syntax restriction

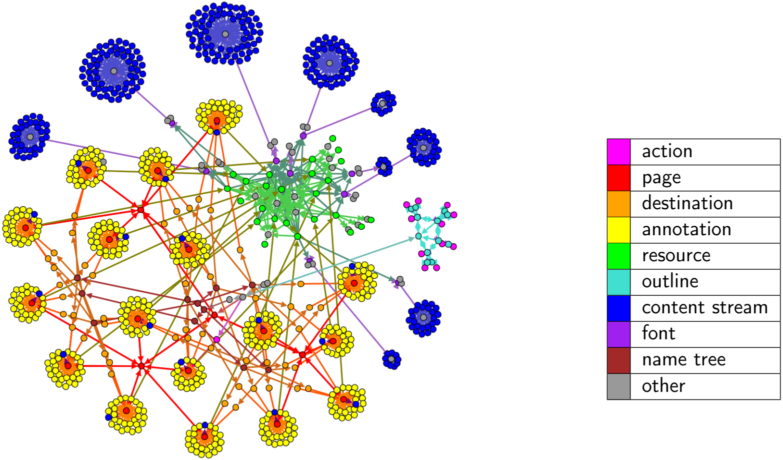
At **syntax** level, guarantee extraction of objects without ambiguity:

- Grammar formalization[7] (BNF).
- Structure restrictions (no updates, no *linearization*, etc.).
- Systematic rejection of “corrupted” files.

> *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, annex C.2

[7] `https://github.com/ANSSI-FR/caradoc/tree/master/doc/grammar`

Type checking



Types of a 17-page document.

Real-world files: lessons learned

Real-world evaluation: 10K files collected from random queries on a web search engine.

Real-world files: lessons learned

Real-world evaluation: 10K files collected from random queries on a web search engine.

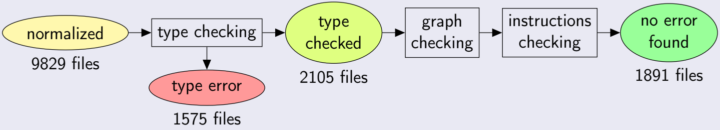
The strict parser rejects common features:

Feature	% of files
incremental updates	65%
object streams	37%
free objects	28%
encryption	5%

$\Rightarrow$ Workaround: **normalize** with relaxed parser first!

Real-world files: lessons learned

Validation after normalization.



Type-checker detected typos:

- `/Blackls1` instead of `/BlackIs1`,
- `/XObjcect` instead of `/XObject`.

We identified incorrect tree structures in the wild.

Caradoc: main commands

Some useful caradoc commands:

- Get stats

  ```
  $ caradoc stats file.pdf
  ```

- Validate

  ```
  $ caradoc stats --strict file.pdf
  ```

- Normalize

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

- Interactive console UI: explore objects, decode stream, search...

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

More on GitHub: https://github.com/ANSSI-FR/caradoc

Conclusion

- PDF is an old format (25+ years), not designed for simple parsing ⇒ error-prone.
- Producers make mistakes, readers try best-effort ⇒ compatibility bugs, security holes...
- We need cleaner, simpler and more robust file formats! ⇒ e.g. Protocol Buffers[8].

[8] `https://developers.google.com/protocol-buffers/`.

Conclusion

My PDF projects:

- Caradoc: `github.com/ANSSI-FR/caradoc`
- Cheat sheet: `github.com/gendx/pdf-cheat-sheets`
- PDF corpus: `github.com/gendx/pdf-corpus`

Some blog posts about PDF: `https://gendignoux.com/blog/`

Twitter: @gendignoux
GitHub: @gendx