

Strengthening Weak Links in the PDF Trust Chain

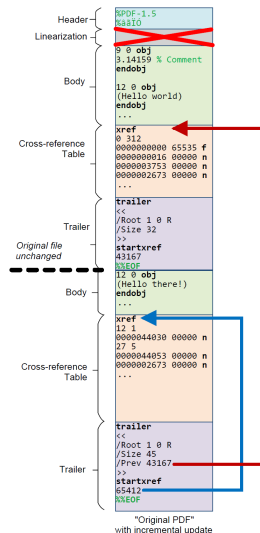
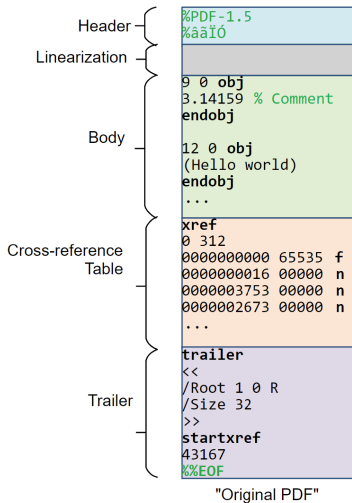
Mark Tullsen, William Harris, Peter Wyatt

May 26, 2022

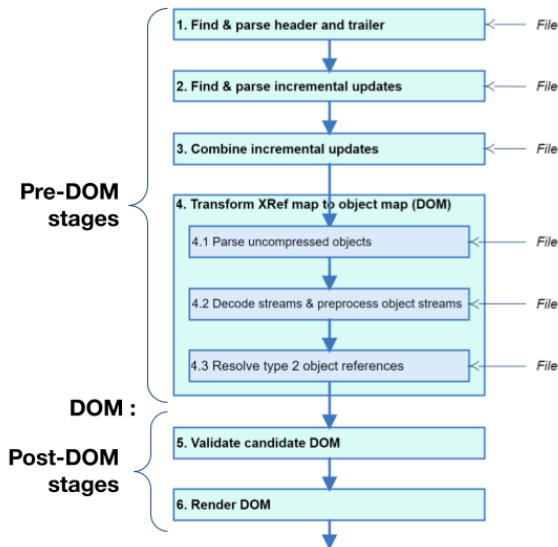
Outline

- 1 Pre-DOM (Pre Document Object Model)
- 2 Modeling Pre-DOM: Highlights
- 3 Conclusions
- 4 Preview: Parser as API

PDF Complexity?



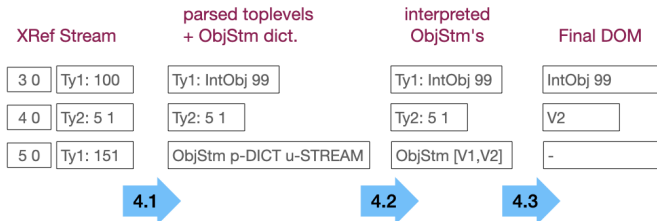
PDF Trust Chain: Pre-DOM and Post-DOM



Vulnerabilities Occurring Primarily Pre-DOM

- Schizophrenic files (different tools, different renderings)
- Polyglot files (file being in 2+ formats)
- Shadow attacks
 - i.e., attacker can
 - add "shadow content" that is PDF-signed,
 - after signing, can update-at-will (revealing shadow content)
 - without giving clear warnings to user.
 - possible because of ability to sign *dead objects* and *cavities*
- Multiple places for hidden/unused/malicious data in PDF
 - non-obvious places, unnoticed when "simply parsing"
 - e.g., shadow-attacks
 - dead bytes, dead objects, dead updates, dead linearization sections, etc.

Stage 4 Sub-Stages: Transform XRef Map to Object Map

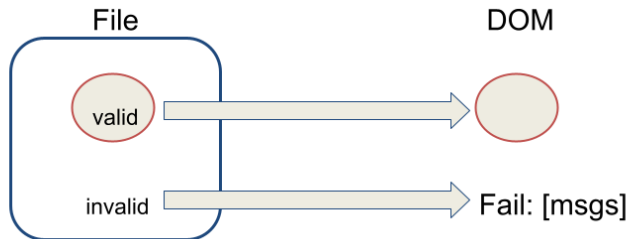


```
1      ...
2 100  3 0 obj 99 endobj
3 123  % object 4 is not here
4 151  5 0 obj
5      <<
6      /Type /ObjStm
7      /Length 3 0 R  % indirect!
8      /N 2           % 2 objects; (potentially indirect)
9      /First 10      % offset to 1st object (potentially indirect)
10     >
11     stream
12     4 0 6 100
13     V1 % PDF-Value here, 4 0 R, [fake comment]
14     V2 % PDF-Value here, 6 0 R, [fake comment]
15     endstream
16     endobj
17 409  7 0 obj ... endobj
18     ...
```

Parser \neq Validator

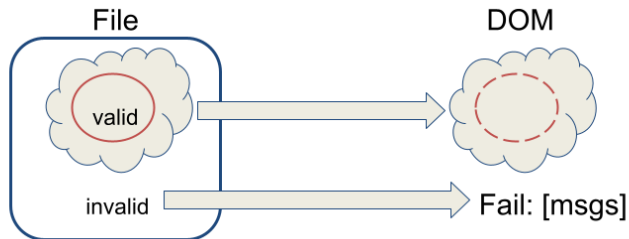
- A surprising source of mis-communication.
- ...

A Validator (Parser \neq Validator)



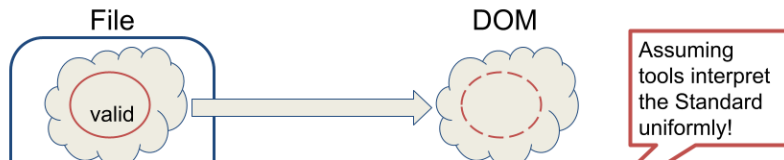
VALIDATOR: only valid PDFs can produce DOM (must Fail otherwise)

A Parser (Parser \neq Validator)



PARSER: efficiently, construct the correct DOM when a valid PDF

A Very Accepting Parser (Parser \neq Validator)



A **Cloud** for each parser/reader/tool:

- The tools are going to be different:
 - redundancies in format allow for different choices
 - tools allow “minor” errors
 - tool may traverse & evaluate implicit data structures differently.
- Goal for our “parser specification”:
 - Encompass any reasonable & correct cloud

Turning Parser into Validator

Parser specification is designed to be

- understandable: clear, pure Haskell
- phased, clearly terminating (get parallelizability for free)
- very lazy "Parser" (big input cloud)

We can extend spec into a validator, orthogonally, via "validate" constructs (turning on/off on with command-line flag). E.g.,

```
do
  (xrefRaw, xrefEndOff) ← pXrefRaw
validate $
  verifyXrefRaw xrefRaw -- ensures no duplicates
```

Accomplishments

- A specification for pre-DOM parsing/computation
 - Clarifies some subtle issues in PDF Standard
 - A growing list of PDF Association “issues” that we have contributed to creating [23,24,...,30]
- Cause *and* effect of
 - unique tool for displaying updates & cavities

Future

- Not accomplished yet
 - the less interesting/subtle parts specified/implemented
 - integrated with our primitive, daedalus-generated parsers to create a tool.
- Create a full pre-DOM tool that
 - supports further PDF features (hybrids, compression, ...)
 - add support for commonly allowed “exuberances”
 - add more “validate”s to get closer to a *validator*.

Implementation?

Tools & renderers rarely need (*demand*) the whole PDF

- reading?
- parsing??
- semantic checks???

Thus, this

```
parsePDF :: FileData → Maybe PDFAbstractSyntax
```

is not going to be used in practice!

One Solution ...

- For complex formats,
 - tools are "projections": rarely used parse/validate all.
 - may have alternate "parsing paths" we want to take
 - e.g., metadata, page 1, text-only
- Shotgun Parsers?
 - ... the deadliest of patterns: "Input data checking, handling interspersed with processing logic"
- I.e., we provide multiple parsers where the following is interspersed through code and the relation between these is **not specified**:

```
parseA :: Offset → IO A
parseB :: Offset → IO B
parseC :: Offset → IO C
validateA :: A → IO ()
validateB :: A → B → IO ()
```

Better Solution, Parser as API

We provide four inter-dependent calls (not *entry points*):

```
parseHdrTrlr :: FileData → IO HdrTrlr
parseUpdates :: HdrTrlr → IO [Updates]
createXRef    :: [Updates] → IO XRef
derefObjId    :: ObjId → XRef → IO PdfValue
```

(The returned types can be as abstract as we wish.)

Using this, we write abstractions on the above:

```
getInitialUpdate :: FileData → IO XRef
getRootValue      :: HdrTrailer → XRef → PdfValue
getPageTree      :: XRef → Tree PdfValue
```