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. Conclusion

4. Preview: A Next Step

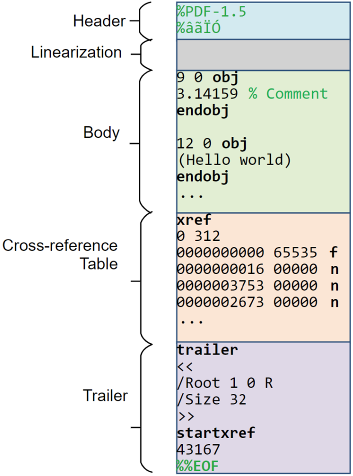
1. Pre-DOM (Pre Document Object Model)

2. Modeling Pre-DOM: Highlights

3. Conclusion

4. Preview: A Next Step

PDF Complexity?

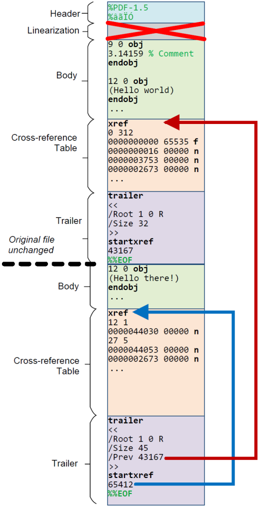


```
%PDF-1.5
%âãÏÓ

9 0 obj
3.14159 % Comment
endobj

12 0 obj
(Hello world)
endobj
...
xref
0 312
0000000000 65535 f
0000000016 00000 n
0000003753 00000 n
0000002673 00000 n
...
trailer
<<
/Root 1 0 R
/Size 32
>>
startxref
43167
%%EOF
```

"Original PDF"

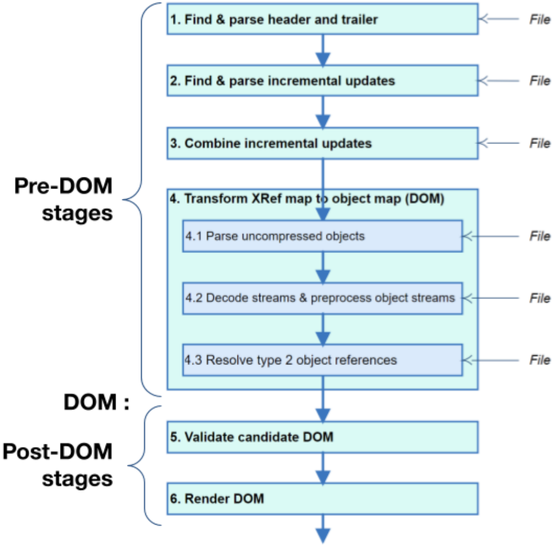


```
%PDF-1.5
%âãÏÓ
9 0 obj
3.14159 % Comment
endobj

12 0 obj
(Hello world)
endobj
...
xref
0 312
0000000000 65535 f
0000000016 00000 n
0000003753 00000 n
0000002673 00000 n
...
trailer
<<
/Root 1 0 R
/Size 32
>>
startxref
43167
%%EOF
12 0 obj
(Hello there!)
endobj
...
xref
12 1
0000044030 00000 n
27 5
0000044053 00000 n
0000002673 00000 n
...
trailer
<<
/Root 1 0 R
/Size 45
/Prev 43167
>>
startxref
65412
%%EOF
```

"Original PDF"
with incremental update

PDF Trust Chain: Pre-DOM and Post-DOM


1. Find & parse header and trailer
File
2. Find & parse incremental updates
File
3. Combine incremental updates
File
Pre-DOM stages
4. Transform XRef map to object map (DOM)
4.1 Parse uncompressed objects
File
4.2 Decode streams & preprocess object streams
File
4.3 Resolve type 2 object references
File
DOM :
Post-DOM stages
5. Validate candidate DOM
6. Render 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.

PDF, and Pre-DOM, Challenges

- Lack of formality in standard. Thus, implementations:
    - are more effort
    - over implement, under implement, wrongly implement
- No definition of acceptable, reasonable error recovery
- Less than ideal design that reflects 27 years of an evolving standard
- Pre-DOM processing
    - is where many parsing errors & recovery occur
    - is non-trivial
    - involves multiple interdependent features
    - involves multiple redundant features
        - schizophrenic if these features aren't mutually consistent

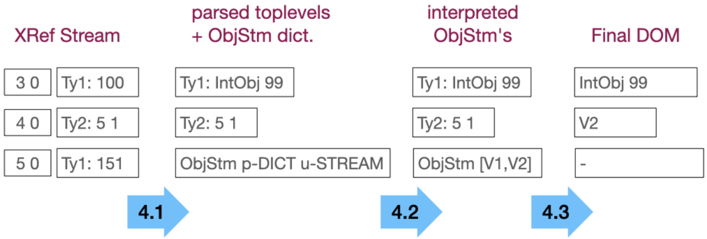
1. Pre-DOM (Pre Document Object Model)

2. Modeling Pre-DOM: Highlights

3. Conclusion

4. Preview: A Next Step

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



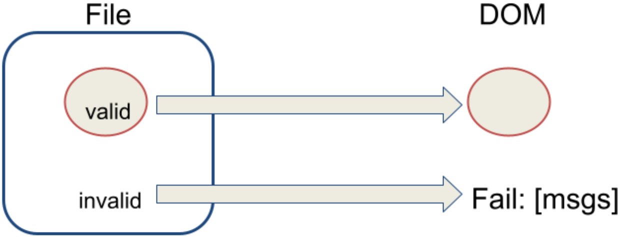
XRef Stream		parsed toplevels + ObjStm dict.	interpreted ObjStm's	Final DOM
3 0	Ty1: 100	Ty1: IntObj 99	Ty1: IntObj 99	IntObj 99
4 0	Ty2: 5 1	Ty2: 5 1	Ty2: 5 1	V2
5 0	Ty1: 151	ObjStm p-DICT u-STREAM	ObjStm [V1,V2]	-

```
        ...
100     3 0 obj 99 endobj
123     % object 4 is not here
151     5 0 obj
        <<
        /Type /ObjStm
        /Length 3 0 R    % indirect!
        /N 2             % 2 objects; (potentially indirect)
        /First 10        % offset to 1st object (potentially indirect)
        >>
```

Parser $\neq$ Validator

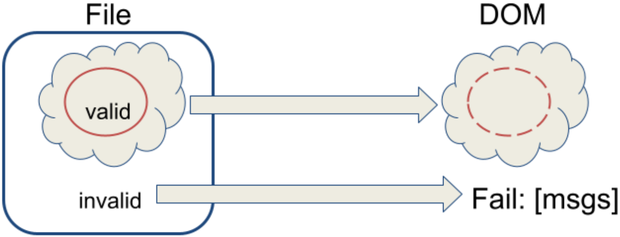
- A surprising source of mis-communication.
- . . .

A Validator (Parser $\neq$ Validator)



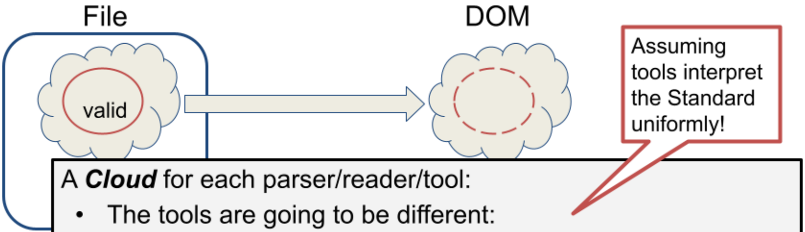
VALIDATOR: only valid PDFs can produce DOM (must Fail otherwise)

A Parser (Parser ≠ Validator)



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

A Very Accepting Parser (Parser ≠ Validator)



Assuming tools interpret the Standard uniformly!

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
```

1. Pre-DOM (Pre Document Object Model)

2. Modeling Pre-DOM: Highlights

3. Conclusion

4. Preview: A Next Step

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*.

1. Pre-DOM (Pre Document Object Model)

2. Modeling Pre-DOM: Highlights

3. Conclusion

4. Preview: A Next Step

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
```