

```

38 2019
39
40 Copyright 2018 The pdfcpu Authors.
41
42 Licensed under the Apache License, Version 2.0 (the "License");
43 you may not use this file except in compliance with the License.
44 You may obtain a copy of the License at
45
46     http://www.apache.org/licenses/LICENSE-2.0
47
48 Unless required by applicable law or agreed to in writing, software
49 distributed under the License is distributed on an "AS IS" BASIS,
50 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
51 See the License for the specific language governing permissions and
52 limitations under the License.
53
54
55 package pdfcpu
56
57 import (
58     "bufio"
59     "bytes"
60     "io"
61     "log"
62     "math/rand"
63     "os"
64     "strings"
65     "time"
66
67     "github.com/pdfcpu/pdfcpu/pkg/api"
68     "github.com/pdfcpu/pdfcpu/pkg/font"
69     "github.com/pdfcpu/pdfcpu/pkg/glyphs"
70     "github.com/pdfcpu/pdfcpu/pkg/objects"
71     "github.com/pdfcpu/pdfcpu/pkg/output"
72     "github.com/pdfcpu/pdfcpu/pkg/reader"
73     "github.com/pdfcpu/pdfcpu/pkg/transform"
74     "github.com/pdfcpu/pdfcpu/pkg/units"
75     "github.com/pdfcpu/pdfcpu/pkg/visual"
76
77     "github.com/pkg/errors"
78     "github.com/sirupsen/logrus"
79
80     "golang.org/x/net/context"
81
82     "github.com/olekukonko/tablewriter"
83
84     "github.com/olekukonko/tablewriter/pkg/reader"
85     "github.com/olekukonko/tablewriter/pkg/writer"
86
87     "github.com/olekukonko/tablewriter/pkg/reader"
88     "github.com/olekukonko/tablewriter/pkg/writer"
89
90     "github.com/olekukonko/tablewriter/pkg/reader"
91     "github.com/olekukonko/tablewriter/pkg/writer"
92
93     "github.com/olekukonko/tablewriter/pkg/reader"
94     "github.com/olekukonko/tablewriter/pkg/writer"
95
96     "github.com/olekukonko/tablewriter/pkg/reader"
97     "github.com/olekukonko/tablewriter/pkg/writer"
98
99     "github.com/olekukonko/tablewriter/pkg/reader"
100    "github.com/olekukonko/tablewriter/pkg/writer"
101
102    "github.com/olekukonko/tablewriter/pkg/reader"
103    "github.com/olekukonko/tablewriter/pkg/writer"
104
105    "github.com/olekukonko/tablewriter/pkg/reader"
106    "github.com/olekukonko/tablewriter/pkg/writer"
107
108    "github.com/olekukonko/tablewriter/pkg/reader"
109    "github.com/olekukonko/tablewriter/pkg/writer"
110
111    "github.com/olekukonko/tablewriter/pkg/reader"
112    "github.com/olekukonko/tablewriter/pkg/writer"
113
114    "github.com/olekukonko/tablewriter/pkg/reader"
115    "github.com/olekukonko/tablewriter/pkg/writer"
116
117    "github.com/olekukonko/tablewriter/pkg/reader"
118    "github.com/olekukonko/tablewriter/pkg/writer"
119
120    "github.com/olekukonko/tablewriter/pkg/reader"
121    "github.com/olekukonko/tablewriter/pkg/writer"
122
123    "github.com/olekukonko/tablewriter/pkg/reader"
124    "github.com/olekukonko/tablewriter/pkg/writer"
125
126    "github.com/olekukonko/tablewriter/pkg/reader"
127    "github.com/olekukonko/tablewriter/pkg/writer"
128
129    "github.com/olekukonko/tablewriter/pkg/reader"
130    "github.com/olekukonko/tablewriter/pkg/writer"
131
132    "github.com/olekukonko/tablewriter/pkg/reader"
133    "github.com/olekukonko/tablewriter/pkg/writer"
134
135    "github.com/olekukonko/tablewriter/pkg/reader"
136    "github.com/olekukonko/tablewriter/pkg/writer"
137
138    "github.com/olekukonko/tablewriter/pkg/reader"
139    "github.com/olekukonko/tablewriter/pkg/writer"
140
141    "github.com/olekukonko/tablewriter/pkg/reader"
142    "github.com/olekukonko/tablewriter/pkg/writer"
143
144    "github.com/olekukonko/tablewriter/pkg/reader"
145    "github.com/olekukonko/tablewriter/pkg/writer"
146
147    "github.com/olekukonko/tablewriter/pkg/reader"
148    "github.com/olekukonko/tablewriter/pkg/writer"
149
150    "github.com/olekukonko/tablewriter/pkg/reader"
151    "github.com/olekukonko/tablewriter/pkg/writer"
152
153    "github.com/olekukonko/tablewriter/pkg/reader"
154    "github.com/olekukonko/tablewriter/pkg/writer"
155
156    "github.com/olekukonko/tablewriter/pkg/reader"
157    "github.com/olekukonko/tablewriter/pkg/writer"
158
159    "github.com/olekukonko/tablewriter/pkg/reader"
160    "github.com/olekukonko/tablewriter/pkg/writer"
161
162    "github.com/olekukonko/tablewriter/pkg/reader"
163    "github.com/olekukonko/tablewriter/pkg/writer"
164
165    "github.com/olekukonko/tablewriter/pkg/reader"
166    "github.com/olekukonko/tablewriter/pkg/writer"
167
168    "github.com/olekukonko/tablewriter/pkg/reader"
169    "github.com/olekukonko/tablewriter/pkg/writer"
170
171    "github.com/olekukonko/tablewriter/pkg/reader"
172    "github.com/olekukonko/tablewriter/pkg/writer"
173
174    "github.com/olekukonko/tablewriter/pkg/reader"
175    "github.com/olekukonko/tablewriter/pkg/writer"
176
177    "github.com/olekukonko/tablewriter/pkg/reader"
178    "github.com/olekukonko/tablewriter/pkg/writer"
179
180    "github.com/olekukonko/tablewriter/pkg/reader"
181    "github.com/olekukonko/tablewriter/pkg/writer"
182
183    "github.com/olekukonko/tablewriter/pkg/reader"
184    "github.com/olekukonko/tablewriter/pkg/writer"
185
186    "github.com/olekukonko/tablewriter/pkg/reader"
187    "github.com/olekukonko/tablewriter/pkg/writer"
188
189    "github.com/olekukonko/tablewriter/pkg/reader"
190    "github.com/olekukonko/tablewriter/pkg/writer"
191
192    "github.com/olekukonko/tablewriter/pkg/reader"
193    "github.com/olekukonko/tablewriter/pkg/writer"
194
195    "github.com/olekukonko/tablewriter/pkg/reader"
196    "github.com/olekukonko/tablewriter/pkg/writer"
197
198    "github.com/olekukonko/tablewriter/pkg/reader"
199    "github.com/olekukonko/tablewriter/pkg/writer"
200
201    "github.com/olekukonko/tablewriter/pkg/reader"
202    "github.com/olekukonko/tablewriter/pkg/writer"
203
204    "github.com/olekukonko/tablewriter/pkg/reader"
205    "github.com/olekukonko/tablewriter/pkg/writer"
206
207    "github.com/olekukonko/tablewriter/pkg/reader"
208    "github.com/olekukonko/tablewriter/pkg/writer"
209
210    "github.com/olekukonko/tablewriter/pkg/reader"
211    "github.com/olekukonko/tablewriter/pkg/writer"
212
213    "github.com/olekukonko/tablewriter/pkg/reader"
214    "github.com/olekukonko/tablewriter/pkg/writer"
215
216    "github.com/olekukonko/tablewriter/pkg/reader"
217    "github.com/olekukonko/tablewriter/pkg/writer"
218
219    "github.com/olekukonko/tablewriter/pkg/reader"
220    "github.com/olekukonko/tablewriter/pkg/writer"
221
222    "github.com/olekukonko/tablewriter/pkg/reader"
223    "github.com/olekukonko/tablewriter/pkg/writer"
224
225    "github.com/olekukonko/tablewriter/pkg/reader"
226    "github.com/olekukonko/tablewriter/pkg/writer"
227
228    "github.com/olekukonko/tablewriter/pkg/reader"
229    "github.com/olekukonko/tablewriter/pkg/writer"
230
231    "github.com/olekukonko/tablewriter/pkg/reader"
232    "github.com/olekukonko/tablewriter/pkg/writer"
233
234    "github.com/olekukonko/tablewriter/pkg/reader"
235    "github.com/olekukonko/tablewriter/pkg/writer"
236
237    "github.com/olekukonko/tablewriter/pkg/reader"
238    "github.com/olekukonko/tablewriter/pkg/writer"
239
240    "github.com/olekukonko/tablewriter/pkg/reader"
241    "github.com/olekukonko/tablewriter/pkg/writer"
242
243    "github.com/olekukonko/tablewriter/pkg/reader"
244    "github.com/olekukonko/tablewriter/pkg/writer"
245
246    "github.com/olekukonko/tablewriter/pkg/reader"
247    "github.com/olekukonko/tablewriter/pkg/writer"
248
249    "github.com/olekukonko/tablewriter/pkg/reader"
250    "github.com/olekukonko/tablewriter/pkg/writer"
251
252    "github.com/olekukonko/tablewriter/pkg/reader"
253    "github.com/olekukonko/tablewriter/pkg/writer"
254
255    "github.com/olekukonko/tablewriter/pkg/reader"
256    "github.com/olekukonko/tablewriter/pkg/writer"
257
258    "github.com/olekukonko/tablewriter/pkg/reader"
259    "github.com/olekukonko/tablewriter/pkg/writer"
260
261    "github.com/olekukonko/tablewriter/pkg/reader"
262    "github.com/olekukonko/tablewriter/pkg/writer"
263
264    "github.com/olekukonko/tablewriter/pkg/reader"
265    "github.com/olekukonko/tablewriter/pkg/writer"
266
267    "github.com/olekukonko/tablewriter/pkg/reader"
268    "github.com/olekukonko/tablewriter/pkg/writer"
269
270    "github.com/olekukonko/tablewriter/pkg/reader"
271    "github.com/olekukonko/tablewriter/pkg/writer"
272
273    "github.com/olekukonko/tablewriter/pkg/reader"
274    "github.com/olekukonko/tablewriter/pkg/writer"
275
276    "github.com/olekukonko/tablewriter/pkg/reader"
277    "github.com/olekukonko/tablewriter/pkg/writer"
278
279    "github.com/olekukonko/tablewriter/pkg/reader"
280    "github.com/olekukonko/tablewriter/pkg/writer"
281
282    "github.com/olekukonko/tablewriter/pkg/reader"
283    "github.com/olekukonko/tablewriter/pkg/writer"
284
285    "github.com/olekukonko/tablewriter/pkg/reader"
286    "github.com/olekukonko/tablewriter/pkg/writer"
287
288    "github.com/olekukonko/tablewriter/pkg/reader"
289    "github.com/olekukonko/tablewriter/pkg/writer"
290
291    "github.com/olekukonko/tablewriter/pkg/reader"
292    "github.com/olekukonko/tablewriter/pkg/writer"
293
294    "github.com/olekukonko/tablewriter/pkg/reader"
295    "github.com/olekukonko/tablewriter/pkg/writer"
296
297    "github.com/olekukonko/tablewriter/pkg/reader"
298    "github.com/olekukonko/tablewriter/pkg/writer"
299
300    "github.com/olekukonko/tablewriter/pkg/reader"
301    "github.com/olekukonko/tablewriter/pkg/writer"
302
303    "github.com/olekukonko/tablewriter/pkg/reader"
304    "github.com/olekukonko/tablewriter/pkg/writer"
305
306    "github.com/olekukonko/tablewriter/pkg/reader"
307    "github.com/olekukonko/tablewriter/pkg/writer"
308
309    "github.com/olekukonko/tablewriter/pkg/reader"
310    "github.com/olekukonko/tablewriter/pkg/writer"
311
312    "github.com/olekukonko/tablewriter/pkg/reader"
313    "github.com/olekukonko/tablewriter/pkg/writer"
314
315    "github.com/olekukonko/tablewriter/pkg/reader"
316    "github.com/olekukonko/tablewriter/pkg/writer"
317
318    "github.com/olekukonko/tablewriter/pkg/reader"
319    "github.com/olekukonko/tablewriter/pkg/writer"
320
321    "github.com/olekukonko/tablewriter/pkg/reader"
322    "github.com/olekukonko/tablewriter/pkg/writer"
323
324    "github.com/olekukonko/tablewriter/pkg/reader"
325    "github.com/olekukonko/tablewriter/pkg/writer"
326
327    "github.com/olekukonko/tablewriter/pkg/reader"
328    "github.com/olekukonko/tablewriter/pkg/writer"
329
330    "github.com/olekukonko/tablewriter/pkg/reader"
331    "github.com/olekukonko/tablewriter/pkg/writer"
332
333    "github.com/olekukonko/tablewriter
```

```

2300         offset, err := strconv.ParseInt(fields[0], 10, 64)
2301         if err != nil {
2302             return err
2303         }
2304         generation, err := strconv.Atoi(fields[1])
2305         if err != nil {
2306             return err
2307         }
2308         entryType := fields[2]
2309         if entryType == "in use" {
2310             return errors.New(objects.parseObjectTableEntry: corrupt ref subobject
2311             entry")
2312         }
2313         var xrefTableEntry *xrefTableEntry
2314         if entryType == "in" {
2315             // in use object
2316             log.Debug.Print("parseObjectTableEntry: Object %d is in use at offset%0d,
2317             generation%0d", objIndexNumber, offset, generation)
2318             if offset == 0 {
2319                 log.Info.Print("parseObjectTableEntry: Skip entry for in use object %d
2320                 with offset 0", objIndexNumber)
2321                 return nil
2322             }
2323             xrefTableEntry =
2324                 &xrefTableEntry{
2325                     xrefTableEntryType:
2326                         xrefTableEntryType{
2327                             From:      &file,
2328                             Offset:     &offset,
2329                             Generation: &generation}
2330                     } else {
2331                         // free object
2332                         log.Debug.Print("parseObjectTableEntry: Object %d is unused, next free is
2333                         object %d", objIndexNumber, offset, generation)
2334                         xrefTableEntry =
2335                             &xrefTableEntry{
2336                                 xrefTableEntryType:
2337                                     xrefTableEntryType{
2338                                         From:      &file,
2339                                         Offset:     &offset,
2340                                         Generation: &generation}
2341                             }
2342                     }
2343             log.Debug.Print("parseObjectTableEntry: Insert new xrefTable entry for Object %d",
2344             objIndexNumber)
2345             xrefTable.Table[objIndexNumber] = xrefTableEntry
2346             return nil
2347         }
2348         return errors.New(objects.parseObjectTableEntry: end")
2349     }
2350     return nil
2351 }

```

[illegible]

```

657         if destTable == null {
658             // return error: how? (perhaps: parameterInfoRef missing entry "Root"?).
659             return errors.New("parameterInfoRef missing entry \"Root\"")
660         }
661         xRefTable.Root = destTableRef
662         log.Debug.Printf("parameterInfoRef: Root object: %s\n", xRefTable.Root)
663     }
664     if xRefTable.Info == nil {
665         // infoRef := & indirectEntry("Info")
666         if infoRef := & indirectEntry("Info")
667         if infoRef != nil {
668             xRefTable.Info = infoRefInfo
669         }
670         log.Debug.Printf("parameterInfoRef: Info object: %s\n", xRefTable.Info)
671     }
672     if xRefTable.ID == nil {
673         idEntry := & indirectEntry("ID")
674         if idEntry == nil {
675             // xRefTable.ID = idEntry
676             log.Debug.Printf("parameterInfoRef: ID object: %s\n", xRefTable.ID)
677         } else if xRefTable.ID == nil {
678             // return errors.New("parameterInfoRef missing entry \"ID\"")
679             return errors.New("parameterInfoRef missing entry \"ID\"")
680         }
681     }
682     log.Debug.Printf("parameterInfoRef end")
683 }
684
685 func paramInfoRef(trailerDict Dict, ctx *Context) (xTable, error) {
686     log.Debug.Printf("parameterInfoRef begin")
687     xRefTable = ctx.XRefTable
688     err = paramTableRef(trailerDict, xRefTable)
689     if err == nil {
690         return nil, err
691     }
692     if err == trailerEntryArrayEmpty("AdditionalStreams") {
693         err = nil
694         log.Debug.Printf("parameterInfoRef: found AdditionalStreams: %s\n", err)
695         a := make([]interface{}, 0)
696         for i, value := range arr {
697             if infoRef := & value (indirectRef); a != nil {
698                 a = append(a, infoRef)
699             }
700         }
701         xRefTable.AdditionalStreams = a
702     }
703     offset = trailerDict.Prev()
704     if offset == nil {
705         log.Debug.Printf("parameterInfoRef: previous xref table section offset: %s\n", offset)
706     }
707     offsetRefStream = trailerDict.InfoEntry("RefStream")
708 }

```

```

9870 // else if
9871 // log.Read.Printf("line %d\n", len(line), line)
9872 }
9873 }
9874 trailerString, err := scanTrailer(s, trailerString)
9875 if err == nil {
9876     return nil, err
9877 }
9878 }
9879 // log.Read.Printf("processTrailer: trailerDict: %v\n",
9880 // len(trailerString), trailerString)
9881
9882 // o, err := parseObject(trailerString)
9883 // if err == nil {
9884 //     return nil, err
9885 // }
9886 // trailerDict, ok := o.Dict()
9887 // if !ok {
9888 //     return nil, errors.New("pdpic: processTrailer: corrupt trailer dict")
9889 // }
9890 // log.Read.Printf("processTrailer: trailerDict: %v\n", trailerDict)
9891 // return parseTrailerDict(trailerDict, ctx)
9892 }
9893
9894 // Parse sub section into corresponding number of sub table entries.
9895 func parseSubSection(s *bufio.Scanner, ctx *Content) (*uint64, error) {
9896     // log.Read.Printf("parseSubSection begin")
9897     // line, err := scanLine(s)
9898     // if err == nil {
9899         //     return nil, err
9900     // }
9901     // }
9902     // log.Read.Printf("parseSubSection: %v\n", line)
9903     // fields := strings.Fields(line)
9904     // }
9905     // }
9906     // }
9907     // }
9908     // }
9909     // }
9910     // }
9911     // }
9912     // }
9913     // }
9914     // }
9915     // }
9916     // }
9917     // }
9918     // }
9919     // }
9920     // }
9921     // }
9922     // }
9923     // }
9924     // }
9925     // }
9926     // }
9927     // }
9928     // }
9929     // }
9930     // }
9931     // }
9932     // }
9933     // }
9934     // }
9935     // }
9936     // }
9937     // }
9938     // }
9939     // }
9940     // }
9941     // }
9942     // }
9943     // }
9944     // }
9945     // }
9946     // }
9947     // }
9948     // }
9949     // }
9950     // }
9951     // }
9952     // }
9953     // }
9954     // }
9955     // }
9956     // }
9957     // }
9958     // }
9959     // }
9960     // }
9961     // }
9962     // }
9963     // }
9964     // }
9965     // }
9966     // }
9967     // }
9968     // }
9969     // }
9970     // }
9971     // }
9972     // }
9973     // }
9974     // }
9975     // }
9976     // }
9977     // }
9978     // }
9979     // }
9980     // }
9981     // }
9982     // }
9983     // }
9984     // }
9985     // }
9986     // }
9987     // }
9988     // }
9989     // }
9990     // }
9991     // }
9992     // }
9993     // }
9994     // }
9995     // }
9996     // }
9997     // }
9998     // }
9999     // }
10000     // }

```

```

1337 // to cxx.NewReader()
1338
1339 br, rdCount, err = reader.SeekHeader(rs)
1340 if err != nil {
1341     return err
1342 }
1343
1344 cxx.NewReader(rsin = br
1345 cxx.NewReader(rsout = rdCount + br
1346
1347 for offset := nil {
1348
1349     rd, err = newPositionHeader(rs, offset)
1350     if err != nil {
1351         return err
1352     }
1353 }
1354
1355 s = bufio.NewReader(r)
1356 s.Split(ScanLines)
1357
1358 line, err = s.ReadString()
1359 if err != nil {
1360     return err
1361 }
1362
1363 log.Printf("line: %s\n", line)
1364
1365 if strings.TrimSpace(line) == "ref" {
1366     log.Printf("buildIndexTableStartingAt: found seek section")
1367     if offset, err = parseSeekSections(s, cxx); err != nil {
1368         return err
1369     }
1370 } else {
1371     log.Printf("buildIndexTableStartingAt: found seek stream")
1372     log.Debug("using file streams + true")
1373     rd, err = newPositionHeader(rs, offset)
1374     if err != nil {
1375         return err
1376     }
1377     if offset, err = parseStream(rs, offset, cxx); err != nil {
1378         log.Printf("buildIndexTableStartingAt after 'huh', err")
1379         // fix file for corrupt invalid seek section.
1380         return hypothesisSection(s)
1381     }
1382     return hypothesisSection(s)
1383 }
1384
1385 log.Printf("buildIndexTableStartingAt: end")
1386
1387 return nil
1388 }
1389
1390 // Populate the cross reference table for this PDF file.
1391 // Note offset of first seek table entry.
1392 // Can be "ref" or indirect object reference or "is a obj"
1393 // Can be "is a obj" or indirect object reference or "is a obj"
1394 // and build up the seek table along the way.
1395 func readerTable(cxx *Context) (err error) {

```

```

550 // =====
551 log.Read.Print("Read: begin!")
552
553 ctx, err := NewContexts(), conf)
554 if err != nil {
555     return nil, err
556 }
557
558 if ctx.ReadOnly {
559     log.Info.Print("PDF Version 1.5 conforming reader")
560 } else {
561     log.Info.Print("PDF Version 1.4 conforming reader - no object streams &
562         references allowed")
563 }
564
565 // Populate shuffTable
566 if err = readShuffTable(ctx); err != nil {
567     return nil, errors.New("Read: shuffTable failed")
568 }
569
570 // Make all objects explicitly available (load into memory) in corresponding
571 // shuffTable entries.
572 // Also makes any involved object streams.
573 if err = dereferenceObjects(ctx, conf); err != nil {
574     return nil, err
575 }
576
577 // Some references write an invariant size into trailer.
578 // i.e. ctx.ShuffTable.Size < ctx.ctx.ShuffTable.TabSize()
579 // i.e. ctx.ShuffTable.Size < len(ctx.ShuffTable)
580
581 // =====
582 log.Read.Print("Read: end")
583
584 return ctx, nil
585
586 // =====
587
588 // ScanLine is a multi-function for a Scanner that returns each line of
589 // text, stripped of any trailing end-of-line marker. The returned line may
590 // be empty, may contain carriage returns, or may contain carriage returns followed
591 // by one or more line or one carriage return or one newline.
592 // If the returned line is empty, it will be returned even if it was an error.
593
594 func scanLines(data []byte, startIdx int) (string, bool) {
595     if startIdx > len(data) {
596         return "", true
597     }
598     if startIdx > len(data) - 1 {
599         return "", true
600     }
601     switch {
602     case startIdx >= 0 && index > 0:
603         if index < len(data) {
604             if index == index {
605                 // do nothing
606             }
607             return index, 1, data[startIdx:index], nil
608         }
609         // do nothing - break

```

```

245
246
247 // Print out the object's section and create corresponding table entry
248 func parseObjectTableSubSection(objIn Scanner, sHeader tableHeader, fields []string
249     error {
250     log.Read.Println("parseObjectTableSubSection: begin")
251
252     startObjNumber, err := stream.Atoi(fields[0])
253     if err == nil {
254         return err
255     }
256
257     objCount, err := stream.Atoi(fields[1])
258     if err == nil {
259         return err
260     }
261
262     log.Read.Println("detected err subSection, startObj=Obj length=ObjIn",
263         startObjNumber, objCount)
264
265     // Process all entries of this subsection into table entries
266     for i := 0; i < objCount; i++ {
267         if err := parseObjectTableEntry(s, sHeader, startObjNumber+i); err == nil {
268             return err
269         }
270     }
271
272     log.Read.Println("parseObjectTableSubSection: end")
273
274     return nil
275 }
276
277 // Parse compressed object
278 func parseCompressedObject(s string) (Object, error) {
279     log.Read.Println("parseCompressedObject: begin")
280
281     o, err := parseObject(s)
282     if err == nil {
283         return nil, err
284     }
285
286     d, ok := o.(Dict)
287     if !ok {
288         // return trivial Object: Integer, Array, etc.
289         log.Read.Println("compressedObject: end, any other than dict")
290         return o, nil
291     }
292
293     streamLength, streamLengthRef := d.length()
294     if streamLength == nil || streamLengthRef == nil {
295         // return dict
296         log.Read.Println("compressedObject: end, dict")
297         return d, nil
298     }
299
300     return nil, errors.New("pdfcpu: compressedObject: stream objects are not to be
301         stored in an object's stream")

```

```

347         }
348     }
349     } else {
350         ct.table(objectNumber) = <objectTableEntry
351     }
352     }
353     }
354     }
355     }
356     }
357     }
358     }
359     }
360     }
361     }
362     }
363     }
364     }
365     }
366     }
367     }
368     }
369     }
370     }
371     }
372     }
373     }
374     }
375     }
376     }
377     }
378     }
379     }
380     }
381     }
382     }
383     }
384     }
385     }
386     }
387     }
388     }
389     }
390     }
391     }
392     }
393     }
394     }
395     }
396     }
397     }
398     }
399     }
400     }
401     }
402     }
403     }
404     }
405     }
406     }
407     }
408     }
409     }
410     }
411     }
412     }
413     }
414     }
415     }
416     }
417     }
418     }
419     }
420     }
421     }
422     }
423     }
424     }
425     }
426     }
427     }
428     }
429     }
430     }
431     }
432     }
433     }
434     }
435     }
436     }
437     }
438     }
439     }
440     }
441     }
442     }
443     }
444     }
445     }
446     }
447     }
448     }
449     }
450     }
451     }
452     }
453     }
454     }
455     }
456     }
457     }
458     }
459     }
460     }
461     }
462     }
463     }
464     }
465     }
466     }
467     }
468     }
469     }
470     }
471     }
472     }
473     }
474     }
475     }
476     }
477     }
478     }
479     }
480     }
481     }
482     }
483     }
484     }
485     }
486     }
487     }
488     }
489     }
490     }
491     }
492     }
493     }
494     }
495     }
496     }
497     }
498     }
499     }
500     }
501     }
502     }
503     }
504     }
505     }
506     }
507     }
508     }
509     }
510     }
511     }
512     }
513     }
514     }
515     }
516     }
517     }
518     }
519     }
520     }
521     }
522     }
523     }
524     }
525     }
526     }
527     }
528     }
529     }
530     }
531     }
532     }
533     }
534     }
535     }
536     }
537     }
538     }
539     }
540     }
541     }
542     }
543     }
544     }
545     }
546     }
547     }
548     }
549     }
550     }
551     }
552     }
553     }
554     }
555     }
556     }
557     }
558     }
559     }
560     }
561     }
562     }
563     }
564     }
565     }
566     }
567     }
568     }
569     }
570     }
571     }
572     }
573     }
574     }
575     }
576     }
577     }
578     }
579     }
580     }
581     }
582     }
583     }
584     }
585     }
586     }
587     }
588     }
589     }
590     }
591     }
592     }
593     }
594     }
595     }
596     }
597     }
598     }
599     }
600     }
601     }
602     }
603     }
604     }
605     }
606     }
607     }
608     }
609     }
610     }
611     }
612     }
613     }
614     }
615     }
616     }
617     }
618     }
619     }
620     }
621     }
622     }
623     }
624     }
625     }
626     }
627     }
628     }
629     }
630     }
631     }
632     }
633     }
634     }
635     }
636     }
637     }
638     }
639     }
640     }
641     }
642     }
643     }
644     }
645     }
646     }
647     }
648     }
649     }
650     }
651     }
652     }
653     }
654     }
655     }
656     }
657     }
658     }
659     }
660     }
661     }
662     }
663     }
664     }
665     }
666     }
667     }
668     }
669     }
670     }
671     }
672     }
673     }
674     }
675     }
676     }
677     }
678     }
679     }
680     }
681     }
682     }
683     }
684     }
685     }
686     }
687     }
688     }
689     }
690     }
691     }
692     }
693     }
694     }
695     }
696     }
697     }
698     }
699     }
700     }
701     }
702     }
703     }
704     }
705     }
706     }
707     }
708     }
709     }
710     }
711     }
712     }
713     }
714     }
715     }
716     }
717     }
718     }
719     }
720     }
721     }
722     }
723     }
724     }
725     }
726     }
727     }
728     }
729     }
730     }
731     }
732     }
733     }
734     }
735     }
736     }
737     }
738     }
739     }
740     }
741     }
742     }
743     }
744     }
745     }
746     }
747     }
748     }
749     }
750     }
751     }
752     }
753     }
754     }
755     }
756     }
757     }
758     }
759     }
760     }
761     }
762     }
763     }
764     }
765     }
766     }
767     }
768     }
769     }
770     }
771     }
772     }
773     }
774     }
775     }
776     }
777     }
778     }
779     }
780     }
781     }
782     }
783     }
784     }
785     }
786     }
787     }
788     }
789     }
790     }
791     }
792     }
793     }
794     }
795     }
796     }
797     }
798     }
799     }
800     }
801     }
802     }
803     }
804     }
805     }
806     }
807     }
808     }
809     }
810     }
811     }
812     }
813     }
814     }
815     }
816     }
817     }
818     }
819     }
820     }
821     }
822     }
823     }
824     }
825     }
826     }
827     }
828     }
829     }
830     }
831     }
832     }
833     }
834     }
835     }
836     }
837     }
838     }
839     }
840     }
841     }
842     }
843     }
844     }
845     }
846     }
847     }
848     }
849     }
850     }
851     }
852     }
853     }
854     }
855     }
856     }
857     }
858     }
859     }
860     }
861     }
862     }
863     }
864     }
865     }
866     }
867     }
868     }
869     }
870     }
871     }
872     }
873     }
874     }
875     }
876     }
877     }
878     }
879     }
880     }
881     }
882     }
883     }
884     }
885     }
886     }
887     }
888     }
889     }
890     }
891     }
892     }
893     }
894     }
895     }
896     }
897     }
898     }
899     }
900     }
901     }
902     }
903     }
904     }
905     }
906     }
907     }
908     }
909     }
910     }
911     }
912     }
913     }
914     }
915     }
916     }
917     }
918     }
919     }
920     }
921     }
922     }
923     }
924     }
925     }
926     }
927     }
928     }
929     }
930     }
931     }
932     }
933     }
934     }
935     }
936     }
937     }
938     }
939     }
940     }
941     }
942     }
943     }
944     }
945     }
946     }
947     }
948     }
949     }
950     }
951     }
952     }
953     }
954     }
955     }
956     }
957     }
958     }
959     }
960     }
961     }
962     }
963     }
964     }
965     }
966     }
967     }
968     }
969     }
970     }
971     }
972     }
973     }
974     }
975     }
976     }
977     }
978     }
979     }
980     }
981     }
982     }
983     }
984     }
985     }
986     }
987     }
988     }
989     }
990     }
991     }
992     }
993     }
994     }
995     }
996     }
997     }
998     }
999     }
1000    }

```

```

210 if offsetHexStream == nil {
211     // no cross reference stream.
212 }
213 if !ctx.readHex(0x00, Version(0) < v14 || !ctx.readHybrid(
214     return nil, errors.Errorf("parse(allocator): PCL: a constant reader
215 found incompatible version %v, %v(leaveVersionString())
216
217 log.Debug.Println("parse(allocator) end")
218 return offset, nil
219 }
220
221 // This file is using cross reference streams.
222
223 if !ctx.readHybrid(
224     ctx.readHybrid < true
225     ctx.readShadingStream < true
226
227 // Is constant readers process hidden objects contained
228 // in %shd% before continuing to process any previous %shd%Section.
229 // Previous %shd%Section is expected to have free entries for hidden entries.
230 // No reason in %shd%Section only.
231 if !ctx.readHex(0x00,
232     err = parseHiddenShdStream(offsetHexStream, ctx); err == nil {
233         return nil, errors.Errorf("
234
235 log.Debug.Println("parse(allocator) end")
236
237 return offset, nil
238 }
239
240 func scanIndexMap(*shdInfo.Scanner) (string, error) {
241     if !ctx.readHex(0x00,
242         if s.err() == nil {
243             return "", s.Err()
244         }
245         return "", errors.New("pdfcpu: scanIndexMap: returning nothing
246     return s.Text(), nil
247 }
248
249 func scanIndex(*shdInfo.Scanner) (string, error) {
250     for i := 0; i <= i; i++ {
251         if !ctx.readHex(0x00,
252             if !ctx.readHex(0x00,
253                 return "", s.Err()
254             if !ctx.readHex(0x00,
255                 break
256             return "", s.Err()
257         }
258     }

```

```

960         fields = strings.Fields(line)
961     }
962 }
963
964 log.Debug.Println("parseObjectSections: All subsections read")
965
966 // Return a PdfHeader struct, "trailer"
967 func (m *Metadata) parseTrailer() (*PdfHeader, error) {
968     if strings.HasPrefix(m, "trailer") {
969         return nil, errors.Errorf("trailersection missing trailer dict, line = '%s'",
970             m)
971     }
972 }
973
974 // Parse the trailer dictionary
975 func (m *Metadata) parseTrailerDict() (*PdfHeader, error) {
976     log.Debug.Println("parseObjectSections: parsing trailer dict.")
977
978     v, err := m.parseObjectSection(m, 1, line)
979     if err != nil {
980         return nil, err
981     }
982 }
983
984 // Get version from first line of file.
985 // Beginning with PDF 1.4, the Version entry in the document's catalog dictionary
986 // is optional. The PDF 1.4 specification states that the version is "1.3.5, 'File
987 // Trailer'". This means that the version is optional in the trailer dictionary.
988 // If the version is not present in the trailer dictionary, the version is assumed to
989 // be the PDF 1.4 version from header to footer.
990 // The header version is used as the first line of the file.
991 // Note that the version is optional in the trailer dictionary (see 1.3.5, 'File
992 // Trailer').
993 func headerVersion(v *PdfHeader) (vVersion, nCount int, err error) {
994     m := new(Metadata)
995     m.parseObjectSection(m, 0, line)
996     return vVersion, nCount, err
997 }
998
999 func (m *Metadata) parseVersion() (*PdfHeader, error) {
1000     var errPdfHeaderVersion = errors.New("pdfdoc: headerVersion: corrupt pdf stream =
1001     header version available")
1002
1003     // Get the version of the file which holds the version of this document.
1004     // If we call this the header version.
1005     if err := m.parseObjectSection(m, 0, line); err != nil {
1006         return nil, &errPdfHeaderVersion
1007     }
1008
1009     buf := make([]byte, 20)
1010     s, err := m.readHeader(buf)
1011     if err != nil {
1012         return nil, &errPdfHeaderVersion
1013     }
1014
1015     s = strings.Trim(s, " ")
1016     prefix := "Header"
1017
1018     if len(s) < len(prefix) || !strings.HasPrefix(s, prefix) {
1019         return nil, &errPdfHeaderVersion
1020     }
1021
1022     pdfVersion, err := PDFVersionFromLine(prefix: len(prefix)+1)
1023     if err != nil {
1024         return nil, errors.Wrap(err, "headerVersion: unknown PDF Header Version")
1025     }
1026
1027     v = &PdfHeader{
1028         s: strings.TrimPrefix(s, prefix),
1029     }
1030
1031     // Detect the used end which should be 1 (end), 0 (end) or 2 chars (endOfCatalog,
1032     // endOfCatalogAndTrailer)

```

```

1197 //log.Read.Println("readFile: begin")
1198
1199 offset, err = offSetLastHeaderSection(ctx)
1200 if err == nil {
1201     return
1202 }
1203
1204 err = buildNewFileStartingAt(ctx, offset)
1205 if err != io.EOF {
1206     return errors.Wrapferr, "readFile: failed: unexpected eof"}
1207
1208 if err == nil {
1209     return
1210 }
1211
1212 // log list of free objects from the "free list".
1213 //log.Read.Println("freelist: %v", cty.FreeObjects)
1214
1215 // Ensure valid freelist of objects.
1216 err = cty.EnsureValidFreelist()
1217 if err == nil {
1218     return
1219 }
1220
1221 log.Read.Println("readFile: end")
1222
1223 return
1224 }
1225
1226 func growBuf(buf []byte, size int, rd io.Reader) ([]byte, error) {
1227
1228     o := make([]byte, size)
1229
1230     n, err := rd.Read(o)
1231     if err == nil {
1232         return nil, err
1233     }
1234
1235     //log.Read.Println("growBufBy: Read %d byteslen", n)
1236
1237     return append(buf[:], o...), nil
1238 }
1239
1240 func maxStreamOffset(line string, streamLen int) (off int) {
1241
1242     off = streamLen - len("stream")
1243
1244     // Skip optional blanks.
1245     // TODO Should we skip optional whitespace instead?
1246     for i, lineOff := 0; off < i; i++ {
1247         if !isSpac(line[i]) {
1248             if line[off] == '\n' {
1249                 return
1250             }
1251         }
1252     }
1253
1254     // Skip till eof.

```

```

310         return index + 1, data[index], nil
311     }
312     // debug - debug
313     return index + 1, data[index], nil
314 }
315 case index > 0:
316     // We have a full carriage return terminated line.
317     return index + 1, data[index], nil
318 }
319 case index > 0:
320     // We have a full newline-terminated line.
321     return index + 1, data[index], nil
322 }
323 }
324 // If we're at EOF, we have a final, non-terminated line. Return it.
325 if atEOF {
326     return len(data), data, nil
327 }
328 // Request more data.
329 return 0, nil, nil
330 }
331
332 // bufio.NewReader(rs is ReaderSeeker, offset int64) (*bufio.Reader, error)
333 func (rs ReaderSeeker) NewReader(offset int64) (*bufio.Reader, error) {
334     if rs == rs.Seek(offset, io.SeekStart), err == nil {
335         return nil, err
336     }
337     log.Read.Print("bufio.NewReader: positioned to offset: %d\n", offset)
338     return bufio.NewReader(rs), nil
339 }
340
341 // Get the file offset of the last R/WSection.
342 // Get the file and search backwards for the first occurrence of startchar
343 // (offset)
344 func (lastRWSec *RWSSection)(ctx *Context) (*int64, error) {
345     rs := ctx.Read.Rs
346
347     var {
348         prevBuf, worked bool //byte
349         bufSize int64 = 512
350         offset int64
351     }
352     for i := 1; offset > 0; i = {
353         off, err := rs.Seek(-index(i), io.SeekEnd)
354         if, err == nil {
355             return lastRWSec, errors.New("previous can't find last r/w sec")
356         }
357         log.Read.Print("Scanning for offsetLastR/WSection starting at %d\n", off)
358         curBuf := make([]byte, bufSize)
359     }
360 }

```

```

340 // @ts-ignore
341 // If we call obj instanceof an object stream we have fun, but also instanceofStream()
342 func parObj instanceofStream() {
343     logDecompressPrint("parObj instanceofStream: decoding had objects: %s", objDec.Count)
344     decodedContent := decodedContent(objDec.FirstObjStream)
345     obj := strings.Fields(string(decodedContent))
346     if len(obj) % 2 != 0 {
347         return errors.New("pdcfg: parObj instanceofStream: corrupt object stream dict")
348     }
349     // e.g., 10 8 11 25 = 2 Objects: 10 @ offset 0, #1 @ offset 25
350     var objArray Array
351     var offsetInt int
352     for i := 0; i < len(obj); i += 2 {
353         offset, err := strconv.Atoi(obj[i+1])
354         if err != nil {
355             return err
356         }
357         offset += objDec.FirstObjOffset
358         if i % 2 == 1 {
359             dst := strings(decodedContent[offset:])
360             logDecompressPrint("parObj instanceofStream: objstring = %s\n", dst)
361             o, err := compressObject(dst)
362             if err != nil {
363                 return err
364             }
365             logDecompressPrint("parObj instanceofStream: [%d] = obj %s\n", i/2+1, obj[i])
366             objArray = append(objArray, o)
367         } else {
368             if i == len(obj) - 1 {
369                 dst := strings(decodedContent[offset:])
370                 logDecompressPrint("parObj instanceofStream: objstring = %s\n", dst)
371                 o, err := compressObject(dst)
372                 if err != nil {
373                     return err
374                 }
375                 logDecompressPrint("parObj instanceofStream: [%d] = obj %s\n", i/2+1, obj[i])
376                 objArray = append(objArray, o)
377             }
378             offset = offset + objDec.FirstObjOffset
379         }
380     }
381     return objArray
382 }

```

[illegible]

```

780         return s1.nil
781     }
782     func isDict(s string) (bool, error) {
783         ok, err := parseDict(s)
784         if err == nil {
785             return false, err
786         }
787         ok, err = o.Dict(s)
788         return ok, err
789     }
790     func scanHeader(s bufio.Scanner, line string) (string, error) {
791         var buf bytes.Buffer
792         var err error
793         var i32 int32
794         var i32s int32
795         buf.WriteString(fmt.Sprintf("line: %s\n", line))
796         // Scan for dict start tag "\n".
797         for {
798             s := strings.Index(line, "\n")
799             if i32 >= 0 {
800                 break
801             }
802             line, err = scanLine(s)
803             buf.WriteString(fmt.Sprintf("line: %s\n", line))
804             if err == nil {
805                 return "", err
806             }
807         }
808     }
809     line := line[s:]
810     buf.WriteString(line)
811     buf.WriteString("\n")
812     buf.WriteString(fmt.Sprintf("scanner dictbuf after start tag: %s\n", line))
813     // Scan for dict tag "\n" but account for inner dicts.
814     line := line[2:]
815     for {
816         if len(line) == 0 {
817             line, err = scanLine(s)
818             if err == nil {
819                 return "", err
820             }
821             buf.WriteString(line)
822             buf.WriteString("\n")
823             buf.WriteString(fmt.Sprintf("scanner dictbuf next line: %s\n", line))
824         }
825         s := strings.Index(line, "\n")
826         if i32 <= 0 {
827             s := strings.Index(line, "\n")
828             if i32 >= 0 {

```

```

6200         if s[i] == 'mha':
6201             colCount = 1
6202         elif s[i] == 'oanh':
6203             colCount = 1
6204         if s[i] == 'mha':
6205             colCount = 2
6206         elif:
6207             return nil, 0, errorCorruptHeader
6208
6209     Log.Debug.Printf("headerVersion: end, found header version: %s", pdfVersion)
6210
6211     return pdfVersion, colCount, nil
6212
6213 // popadefinitions is a hack for displaying correct section.
6214 // It populates the offsettable by reading in all indirect objects line by line
6215 // and works on the assumption of a single xref section - meaning no incremental
6216 // updates have been made.
6217 func ParseDefinitionTable(ctt *Content) error {
6218     err := parse
6219     if err != nil {
6220         if err == FrequentGeneration {
6221             cttable.AddEntry(
6222                 Free:    true,
6223                 Offset: 0,
6224                 Generation: 0)
6225         }
6226         rs := ct.NewReader()
6227         colCount := ct.NewReader(colCount)
6228         var off, offset index
6229         rs, err := newMultiIndexReader(rs, offset)
6230         if err == nil {
6231             return err
6232         }
6233     }
6234     s := bufio.NewReaderString(rs)
6235     s.SplitScanLines()
6236     bo := []byte{
6237         'w', 'r',
6238         'withnbody', 'bool',
6239         'withnheader', 'bool',
6240         'withn trailer', 'bool',
6241     }
6242     for {
6243         line, err := scanLineFrom(s)
6244         if err == nil {
6245             break
6246         }
6247         if withnheader {
6248             offset += int64(len(line) * colCount)
6249         }
6250         if withnbody {
6251             bo = append(bo, ' ')
6252         }
6253         if strings.Index(line, "startref")
6254             if i > 0 {

```

[illegible]

```

363 // https://stackoverflow.com/questions/4040404/using-std-weak-map
374
375         .. err = r.Read(cursor)
376         if err == nil {
377             return nil, err
378         }
379     }
380
381     workBuf = curBuf
382     if preBuf == nil {
383         workBuf = append(curBuf, preBuf...)
384     }
385
386     j := strings.LastIndex(string(workBuf), "startref")
387     if j == -1 {
388         preBuf = curBuf
389         continue
390     }
391
392     p = workBuf[j+1len(string(workBuf)):]
393     posOff := strings.Index(string(p), "MEOF")
394     if posOff == -1 {
395         return nil, errors.New("pfcpu: no matching MEOF for startref")
396     }
397
398     p = p[posOff:]
399     offset, err := strconv.ParseInt(strings.TrimSpace(string(p)), 10, 64)
400     if err == nil {
401         return nil, errors.New("pfcpu: corrupted last ref section")
402     }
403 }
404
405 log.Read.Printf("Offset last xrefsection: %d\n", offset)
406
407 return bufOffset, nil
408 }
409
410 // Read next subsection entry and generate corresponding ref table entry.
411 func (parser *ParserTableEntry) xrefInfoScanner, xrefInfo *xrefInfo, objectNumber int) {
412     log.Read.Printf("parser:tableEntry: begin")
413
414     line, err = scanner()
415     if err == nil {
416         return nil, err
417     }
418
419     if xrefInfo.Exists(objectNumber) {
420         log.Read.Printf("parser:tableEntry: end - Skip entry %d - already assigned", objectNumber)
421         return nil, nil
422     }
423
424     fields = strings.Split(line)
425     if len(fields) == 1 {
426         log.Read.Printf("parser:tableEntry: end - Skip entry %d - already assigned", objectNumber)
427         return errors.New("pfcpu: parser:tableEntry: corrupt ref subsection")
428     }
429 }
430
431 }
432
433 }

```

```

380 // @ts-ignore
390 std::ostringstream objArray
391
392     log.Read_Printf("paramsJsonObjectStream")
393
394     return nil
395
396 // for each object embedded in this std::Stream create the corresponding std table
397 // that shall be present in the stream
398 // @ts-ignore
399     extractJsonObjectFromStream(buf, lByte, &stdStreamTable, &ts
400         ) error {
401
402         log.Read_Printf("extractJsonObjectFromStream begin")
403
404         // Note:
405         // * A value of zero for an element in the m array indicates that the corresponding
406         //   element shall not be present in the stream
407         // * The default value shall be zero, if there is one.
408         // * If an element is zero, the type field shall not be present, and shall
409         //   default to type: 0
410
411         int m = std.m[0]
412         int n = std.n[0]
413         int k = std.k[0]
414
415         xrefEntryLen = "1" * 12 + "3"
416
417         extractEntryLen = "extractJsonObjectFromStream: begin xrefEntryLen = "
418             + xrefEntryLen
419
420         // @ts-ignore
421         if len(buf) > xrefEntryLen > 0 {
422             return errors.New("pfcgo: extractJsonObjectFromStream: corrupt
423                 refstream")
424         }
425
426         objCount = len(std.objs)
427         log.Read_Printf("extractJsonObjectFromStream: objCount %d bufLen %d",
428             objCount, len(buf))
429
430         log.Read_Printf("extractJsonObjectFromStream: len(buf) %d
431             xrefEntryLen %s", len(buf), xrefEntryLen)
432         // Sometimes there is an additional xref entry not accounted for by "index".
433         // This entry contains a pointer and do not treat this as an error
434         // @ts-ignore
435         return errors.New("pfcgo: extractJsonObjectFromStream: corrupt
436             refstream")
437     }
438
439     j = 0
440
441     // bufio.Reader interprets the content of buf as an int64.
442     // therefore we func(buf).i() instead of
443     // buf.i()
444     for i := range buf {
445         i := int64(b)
446
447         return
448     }
449 }

```

```

645 log.ReadPrintln("parseHeader: Insert new shafeable entry for Object %d\n",
646 objId);
647
648 ctx.Table[objId] = &entry
649
650 // Use new shafeableStream object to store a context
651 prevOffset = &entry.prevOffset
652
653 log.ReadPrintln("parseHeader: end")
654
655 return prevOffset, nil
656
657 // =====
658 // Parse an shafeableStream as a typical PDF file.
659 func parseShafeableStream(offset uint64, ctx context) error {
660
661     log.ReadPrintln("parseShafeableStream: begin")
662
663     rd, err := newPositionalReader(ctx, offset)
664     if err != nil {
665         return err
666     }
667
668     // =====
669     // parse shafeableStream's offset, etc
670     if err != nil {
671         return err
672     }
673
674     log.ReadPrintln("parseShafeableStream: end")
675
676     return nil
677
678 // =====
679 // Parse trailer dict and return any offset of a previous shafe section.
680 func parseTrailerInfo(dict shafeableShafeTable) error {
681
682     log.ReadPrintln("parseTrailerInfo: begin")
683
684     if _, found = dict.FindEntry("Root"); found {
685         // =====
686         if encryptObj := dict["Encrypt"]; encryptObj != nil {
687             if encryptObj["Name"] != nil {
688                 shafeableDecrypt = decryptObj["Name"]
689                 log.ReadPrintln("parseTrailerInfo: Encrypt object %s\n",
690 shafeableDecrypt)
691             }
692         }
693     }
694
695     // =====
696     if shafeable.Size == nil {
697         size = d.Size()
698     }
699     if size != nil {
700         return errors.New("pdfproc: parseTrailerInfo: missing entry \"%s\"")
701     }
702
703     // =====
704     // Not reliable
705     // Because after all read in,
706     shafeable.Size = size
707
708     // =====
709     if shafeable.Root == nil {
710         rootObjId = d.IndirectRefEntry("Root")
711     }
712 }

```

```

607 // Check for error
608 if k == 0 {
609     // Check for err
610     ok, err = bufio.ReadString()
611     if err != nil || !ok {
612         return buf.String(), nil
613     }
614 } else {
615     k--
616 }
617 line := line[j+1:]
618 continue
619 // No op
620 line, err = scanner(s)
621 if err == nil {
622     return "", err
623 }
624 buf.WriteString(line)
625 buf.WriteString(" ")
626 log.ReadPrintf("scanner failed disclim next line: %s\n", line)
627 } else {
628     // Yes ok
629     if j < 0 {
630         s = string.Index(line, "\n")
631         j++
632         line = line[j:]
633     } else {
634         // No op
635         line = line[j+1:]
636     }
637 }
638 // Yes ok
639 if j < 0 {
640     // Handle ok
641     line = line[j+1:]
642 } else {
643     // Handle no
644     if k == 0 {
645         // Check for dict
646         ok, err = bufio.ReadString()
647         if err == nil || ok {
648             return buf.String(), nil
649         }
650     } else {
651         k--
652     }
653     line = line[j+1:]
654 }
655 }
656 }
657 }
658 }
659 }
660 }
661 func processTrailer(ctx context.Context, s bufio.Scanner, line string) (*int64, error) {
662     var trailerString string
663     if line == "trailer" {
664         trailerString = line[j:]
665         log.ReadPrintf("processTrailer: trailer leftover: %s\n", trailerString)

```

```

9780 // Scan trailer
9781     off = processTrailer(ctx, s, string(bb))
9782     return err
9783 }
9784
9785 // Ignore all until "trailer".
9786 func strings.Index(line, "trailer")
9787 if i >= 0 {
9788     bb := append(bb, line...)
9789     while(trailer != line...)
9790         continue
9791 }
9792 l := strings.Index(line, "\r\n")
9793 if i >= 0 {
9794     offset += int64(int(l)line - eolCount)
9795     withinStrat = true
9796     continue
9797 }
9798 if !withinObj {
9799     l := strings.Index(line, "obj")
9800     if i >= 0 {
9801         withinObj = true
9802         off += offset
9803         bb = append(bb, line[i:l]...)
9804     }
9805     offset += int64(int(l)line - eolCount)
9806     withinObj = true
9807 }
9808
9809 // Within obj
9810 offset += int64(int(l)line - eolCount)
9811 bb = append(bb, s)
9812 bb = append(bb, line[:l])
9813 l := strings.Index(line, "endobj")
9814 if i >= 0 {
9815     objMr, generation, err = parseObjectAttributes(bl)
9816     if err == nil {
9817         return err
9818     }
9819 }
9820 if off == 0 {
9821     ctx.table[objMr] = &objTableEntry{
9822         offset: false,
9823         offset: objMr,
9824         generation: generation}
9825     bb = nil
9826     withinObj = false
9827 }
9828 }
9829 return nil
9830 }
9831
9832 // Build an instance by reading from object's or array's stream
9833 func buildFromStartAtStartAtCtx *Context, offset uint64 error {
9834     log.Debug.Printf("buildFromStartAtStartAtCtx: begin")
9835 }

```

```

1307 //
1308 // line in string(buf)
1309 //
1310 // endline = string::index(line, "\nendline")
1311 // streamline = string::index(line, "stream")
1312 //
1313 // if endline > 0 && (streamline < 0 || streamline > endline) {
1314 //   // No stream marker in buf detected.
1315 //   break;
1316 // }
1317 //
1318 // For very rare cases where "stream" also occurs within buf dict
1319 // we need to find the last "stream" marker before a position and marker.
1320 // For streamline > 0, we need to find the last occurrence of dict(line, streamline)
1321 // lastStreamMarker(streamline, endline, line)
1322 //
1323 //
1324 //
1325 //
1326 //
1327 //
1328 // log.Debug.Printf("buffer: endline=%d streamline=%d\n", endline, streamline)
1329 //
1330 // if streamline > 0 {
1331 //   // streamOffset = the offset where the actual stream data begins.
1332 //   // In fact, it is right after the call after "stream"
1333 //   //
1334 //   //
1335 //   //
1336 //   //
1337 //   //
1338 //   //
1339 //   //
1340 //   //
1341 //   //
1342 //   //
1343 //   //
1344 //   //
1345 //   //
1346 //   //
1347 //   //
1348 //   //
1349 //   //
1350 //   //
1351 //   //
1352 //   //
1353 //   //
1354 //   //
1355 //   //
1356 //   //
1357 //   //
1358 //   //
1359 //   //
1360 //   //
1361 //   //
1362 //   //
1363 //   //
1364 //   //
1365 //   //
1366 //   //
1367 //   //
1368 //   //
1369 //   //
1370 //   //
1371 //   //
1372 //   //
1373 //   //
1374 //   //
1375 //   //
1376 //   //
1377 //   //
1378 //   //
1379 //   //
1380 //   //
1381 //   //
1382 //   //
1383 //   //
1384 //   //
1385 //   //
1386 //   //
1387 //   //
1388 //   //
1389 //   //
1390 //   //
1391 //   //
1392 //   //
1393 //   //
1394 //   //
1395 //   //
1396 //   //
1397 //   //
1398 //   //
1399 //   //
1400 //   //
1401 //   //
1402 //   //
1403 //   //
1404 //   //
1405 //   //
1406 //   //
1407 //   //
1408 //   //
1409 //   //
1410 //   //
1411 //   //
1412 //   //
1413 //   //
1414 //   //
1415 //   //
1416 //   //
1417 //   //
1418 //   //
1419 //   //
1420 //   //
1421 //   //
1422 //   //
1423 //   //
1424 //   //
1425 //   //
1426 //   //
1427 //   //
1428 //   //
1429 //   //
1430 //   //
1431 //   //
1432 //   //
1433 //   //
1434 //   //
1435 //   //
1436 //   //
1437 //   //
1438 //   //
1439 //   //
1440 //   //
1441 //   //
1442 //   //
1443 //   //
1444 //   //
1445 //   //
1446 //   //
1447 //   //
1448 //   //
1449 //   //
1450 //   //
1451 //   //
1452 //   //
1453 //   //
1454 //   //
1455 //   //
1456 //   //
1457 //   //
1458 //   //
1459 //   //
1460 //   //
1461 //   //
1462 //   //
1463 //   //
1464 //   //
1465 //   //
1466 //   //
1467 //   //
1468 //   //
1469 //   //
1470 //   //
1471 //   //
1472 //   //
1473 //   //
1474 //   //
1475 //   //
1476 //   //
1477 //   //
1478 //   //
1479 //   //
1480 //   //
1481 //   //
1482 //   //
1483 //   //
1484 //   //
1485 //   //
1486 //   //
1487 //   //
1488 //   //
1489 //   //
1490 //   //
1491 //   //
1492 //   //
1493 //   //
1494 //   //
1495 //   //
1496 //   //
1497 //   //
1498 //   //
1499 //   //
1500 //   //
1501 //   //
1502 //   //
1503 //   //
1504 //   //
1505 //   //
1506 //   //
1507 //   //
1508 //   //
1509 //   //
1510 //   //
1511 //   //
1512 //   //
1513 //   //
1514 //   //
1515 //   //
1516 //   //
1517 //   //
1518 //   //
1519 //   //
1520 //   //
1521 //   //
1522 //   //
1523 //   //
1524 //   //
1525 //   //
1526 //   //
1527 //   //
1528 //   //
1529 //   //
1530 //   //
1531 //   //
1532 //   //
1533 //   //
1534 //   //
1535 //   //
1536 //   //
1537 //   //
1538 //   //
1539 //   //
1540 //   //
1541 //   //
1542 //   //
1543 //   //
1544 //   //
1545 //   //
1546 //   //
1547 //   //
1548 //   //
1549 //   //
1550 //   //
1551 //   //
1552 //   //
1553 //   //
1554 //   //
1555 //   //
1556 //   //
1557 //   //
1558 //   //
1559 //   //
1560 //   //
1561 //   //
1562 //   //
1563 //   //
1564 //   //
1565 //   //
1566 //   //
1567 //   //
1568 //   //
1569 //   //
1570 //   //
1571 //   //
1572 //   //
1573 //   //
1574 //   //
1575 //   //
1576 //   //
1577 //   //
1578 //   //
1579 //   //
1580 //   //
1581 //   //
1582 //   //
1583 //   //
1584 //   //
1585 //   //
1586 //   //
1587 //   //
1588 //   //
1589 //   //
1590 //   //
1591 //   //
1592 //   //
1593 //   //
1594 //   //
1595 //   //
1596 //   //
1597 //   //
1598 //   //
1599 //   //
1600 //   //
1601 //   //
1602 //   //
1603 //   //
1604 //   //
1605 //   //
1606 //   //
1607 //   //
1608 //   //
1609 //   //
1610 //   //
1611 //   //
1612 //   //
1613 //   //
1614 //   //
1615 //   //
1616 //   //
1617 //   //
1618 //   //
1619 //   //
1620 //   //
1621 //   //
1622 //   //
1623 //   //
1624 //   //
1625 //   //
1626 //   //
1627 //   //
1628 //   //
1629 //   //
1630 //   //
1631 //   //
1632 //   //
1633 //   //
1634 //   //
1635 //   //
1636 //   //
1637 //   //
1638 //   //
1639 //   //
1640 //   //
1641 //   //
1642 //   //
1643 //   //
1644 //   //
1645 //   //
1646 //   //
1647 //   //
1648 //   //
1649 //   //
1650 //   //
1651 //   //
1652 //   //
1653 //   //
1654 //   //
1655 //   //
1656 //   //
1657 //   //
1658 //   //
1659 //   //
1660 //   //
1661 //   //
1662 //   //
1663 //   //
1664 //   //
1665 //   //
1666 //   //
1667 //   //
1668 //   //
1669 //   //
1670 //   //
1671 //   //
1672 //   //
1673 //   //
1674 //   //
1675 //   //
1676 //   //
1677 //   //
1678 //   //
1679 //   //
1680 //   //
1681 //   //
1682 //   //
1683 //   //
1684 //   //
1685 //   //
1686 //   //
1687 //   //
1688 //   //
1689 //   //
1690 //   //
1691 //   //
1692 //   //
1693 //   //
1694 //   //
1695 //   //
1696 //   //
1697 //   //
1698 //   //
1699 //   //
1700 //   //
1701 //   //
1702 //   //
1703 //   //
1704 //   //
1705 //   //
1706 //   //
1707 //   //
1708 //   //
1709 //   //
1710 //   //
1711 //   //
1712 //   //
1713 //   //
1714 //   //
1715 //   //
1716 //   //
1717 //   //
1718 //   //
1719 //   //
1720 //   //
1721 //   //
1722 //   //
1723 //   //
1724 //   //
1725 //   //
1726 //   //
1727 //   //
1728 //   //
1729 //   //
1730 //   //
1731 //   //
1732 //   //
1733 //   //
1734 //   //
1735 //   //
1736 //   //
1737 //   //
1738 //   //
1739 //   //
1740 //   //
1741 //   //
1742 //   //
1743 //   //
1744 //   //
1745 //   //
1746 //   //
1747 //   //
1748 //   //
1749 //   //
1750 //   //
1751 //   //
1752 //   //
1753 //   //
1754 //   //
1755 //   //
1756 //   //
1757 //   //
1758 //   //
1759 //   //
1760 //   //
1761 //   //
1762 //   //
1763 //   //
1764 //   //
1765 //   //
1766 //   //
1767 //   //
1768 //   //
1769 //   //
1770 //   //
1771 //   //
1772 //   //
1773 //   //
1774 //   //
1775 //   //
1776 //   //
1777 //   //
1778 //   //
1779 //   //
1780 //   //
1781 //   //
1782 //   //
1783 //   //
1784 //   //
1785 //   //
1786 //   //
1787 //   //
1788 //   //
1789 //   //
1790 //   //
1791 //   //
1792 //   //
1793 //   //
1794 //   //
1795 //   //
1796 //   //
1797 //   //
1798 //   //
1799 //   //
1800 //   //
1
```



```

1570         }
1571         return false
1572     }
1573
1574     // We found the last zero (end1) just after end of dict only whitespace,
1575     ok = strings.TrimSpace(end1) == ">"
1576
1577     // Log Read.Printf("keyWords=string(HeaderDict)+NDICT: end: %s", ok)
1578
1579     return ok
1580 }
1581
1582 func buildFilterPipeline(ctx *Context, filterArray, decodeParamsArr Array, decodeParams
1583 *dict) (*Pfilter, error) {
1584     var filterPipeline []Pfilter
1585
1586     for i, f := range filterArray {
1587
1588         filterName, ok := f.(Name)
1589         if !ok {
1590             corrupt := true
1591             return nil, errors.New("pfilter: buildFilterPipeline: filterArray elements
1592 corrupt")
1593         }
1594         if decodeParams == nil || decodeParamsArr[i] == nil {
1595             filterPipeline = append(filterPipeline, PFilter{Name:
1596 filterName, decodeParams: nil})
1597             continue
1598         }
1599         dict, ok := decodeParamsArr[i].(Dict)
1600         if !ok {
1601             corrupt := true
1602             return nil, errors.New("pfilter: buildFilterPipeline: dictHeader")
1603         }
1604         if !ok {
1605             return nil, errors.Errorf("buildFilterPipeline: corrupt Dict: %s",
1606 dict)
1607         }
1608         if err := deferenceDicts(dict, indirectObjectNumberValue()) {
1609             if err != nil {
1610                 return nil, err
1611             }
1612             dict = d
1613         }
1614     }
1615
1616     filterPipeline = append(filterPipeline, PFilter{Name: filterName.String(),
1617 decodeParams: dict})
1618 }
1619
1620 return filterPipeline, nil
1621 }
1622
1623 // Decode the filter pipeline associated with this stream dict:
1624 func pfilterPipeline(ctx *Context, dict Dict) (*Pfilter, error) {
1625     log.Read.Printf("pfilterPipeline: begin")
1626
1627     var err error
1628
1629     o, found := dict.Find("Filter")
1630     if !found {
1631         return nil, errors.New("stream is not compressed")
1632     }
1633 }

```

[illegible]

```

1130 // Save the saveDecodedContentContext to ctx.content, id, saveStreams, objKey, goenv int, decoded
1131 // err error()
1132
1133 // Log.Read.Print("saveDecodedContentContext: begin decode\n"), decode)
1134
1135 // If the "identity" crypt filter is used we do not need to decode.
1136 if ctx.will nil or ctx.filterKey == nil {
1137     if ctx.filterPolicyName == 1 do do.FilterPolicy(1).Name == "Crypt" {
1138         return nil
1139     }
1140 }
1141
1142 // Special case: If the length of the encoded data is 0, we do not need to decode anything.
1143 if ctx.len(ReadRaw) == 0 {
1144     id.Content = id.Raw
1145     return nil
1146 }
1147
1148 // id gets created after WebStream parsing.
1149 // WebStreams are not encrypted.
1150 if ctx.will == 0 or ctx.filterKey == nil {
1151     id.Raw, err = decryptReadRaw(id.Raw, objKey, goenv, ctx.KeyKey, ctx.AESStreams,
1152 ctx.Ex)
1153     if err == nil {
1154         return err
1155     }
1156     id = len(id.Raw)
1157     id.StreamLength = id
1158 }
1159
1160 // If decode
1161     return nil
1162 }
1163
1164 // Actual decoding of content stream.
1165 err = decodeStream(id)
1166 if err == filter.StreamSupportFilter {
1167     err = nil
1168 }
1169
1170 if err == nil {
1171     return err
1172 }
1173
1174 // Log.Read.Print("saveDecodedContentContext: end")
1175
1176 return nil
1177
1178 // Decode compressed objectTableEntry
1179 func decodeCompressedObjectTableEntry (ctx *Context, objTable *Table, objectNumber int, entry
1180 *ObjectEntry) error {
1181     // Log.Read.Print("decodeCompressedObjectTableEntry: compressed object id at %d\n"),
1182     objectNumber, entry.ObjectStream, entry.ObjectStreamLen)
1183
1184     // Message streamName entry at reference object stream.
1185     objStreamLenTableEntry, obj = objTable.FindEntry(objectNumber)
1186     if obj !=

```

```

2037         if m == nil {
2038             return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = missing array entry m, objId: %v", objId)
2039         }
2040         if len(m) == 2 {
2041             if len(a) == 4 {
2042                 return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry m, needs length 2 or 4", objId)
2043             }
2044             offset, ok = a[0].(Integer)
2045             if !ok {
2046                 return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry m, needs Integer values", objId)
2047             }
2048             offset4 := Int64(offset.Value())
2049             ctx.OffsetPrincipalsTable = offsets4
2050             if len(a) == 4 {
2051                 if !
2052                     return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry m, needs Integer values", objId)
2053             }
2054             ctx.OffsetOverluminTable = offsets4
2055         }
2056     }
2057     return nil
2058 }
2059
2060 func LoadBinaryStream(ctx *Context, s *StreamReader, objId, genNr int) error {
2061     var err error
2062
2063     // Load stream's content and store data into offsetable entry
2064     if err = LoadInfiniteStream(ctx, s, objId, genNr, nil); err != nil {
2065         return errors.Wrapf(err, "dereferencingContext: problem dereferencing stream %d", objId)
2066     }
2067     ctx.Read.BinaryFileSize += s.GetSizeLength()
2068
2069     // Decode stream's content
2070     err = saveDecodedStreamContent(ctx, s, objId, genNr, ctx.DecodedAllStreams)
2071     return err
2072 }
2073
2074 func updateLinearizationDict(ctx *Context, o Object) {
2075     switch o := o.(type) {
2076     case StreamDict:
2077         ctx.Read.BinaryFileSize += o.GetSizeLength()
2078     }
2079 }

```

```

2350 }
2351 // Create a mutable version of the file (since it's in the catalog
2352 // and record this as rootVersion (as opposed to headerVersion).
2353 std::unique_ptr<RootVersion> xRefTable = RootTable.error()
2354 // Log the RootVersion
2355 log.LogPrintln("IdentifyRootVersion: begin")
2356 // Log the version from the RootTable
2357 RootVersion* rtr = xRefTable->ParseRootVersion()
2358 if (rtr == nil)
2359     return error
2360 if rootVersionStr == nil {
2361     return nil
2362 }
2363 // Validate version and save corresponding constant to xRefTable.
2364 rootVersion, err := PDPVersion(rootVersionStr)
2365 if err != nil
2366     return errors.Wrap(error, "IdentifyRootVersion: unknown PDP Root version:
2367 %v", rootVersionStr)
2368 xRefTable.RootVersion = rootVersion
2369 // Since it's a header version we can override by a version entry in the
2370 // catalog.
2371 if xRefTable.HeaderVersion < v1 {
2372     log.LogPrintln("IdentifyRootVersion: PDP version is %s - will ignore root
2373 version %s",
2374         xRefTable.HeaderVersion, rootVersionStr)
2375     xRefTable.HeaderVersion = rootVersionStr
2376 log.LogPrintln("IdentifyRootVersion: end")
2377 return nil
2378 }
2379 // Parse all Objects including stream content from file and save to the corresponding
2380 // headerFields.
2381 // Parse all Objects including object stream and linearization dicts.
2382 std::unique_ptr<Catalog> cxtx = Catalog.Configure()
2383 log.LogPrintln("DeserializeCatalog: begin")
2384 xRefTable = cxtx.XRefTable
2385 // Note for unencrypted files.
2386 // Mandatory provide users to open & display file.
2387 // Access may be restricted (Open access available).
2388 // Optionally provide comments in order to gain unrestricted access.
2389 if err := xRefTable.ParseCatalog()
2390 
```

```

2487 d, err := differenceCdc(c1x, ifObjectNumber.Value())
2488 if err != nil {
2489     return err
2490 }
2491 log.Read.Printf("%s\n", d)
2492
2493 // We need to decrypt this file in order to read it.
2494 return setupEncryptionKey(c1x, d)
2495
2496

```

```

3420 // return nil, nil
3421 // 1426
3422 // 1426
3423 // compressed stream.
3424 // 1428
3425 var filterPipeline []Pffilter
3426 // 1431
3427 if indirOf, ok := o.Directref(ctx); ok {
3428     // 1433
3429     o, err = derefResourceDirect(ctx, indirOf.ObjectNumber.Value())
3430     // 1435
3431     if err != nil {
3432         return nil, err
3433     }
3434 // 1437
3435 // 1437
3436 // 1437
3437 // 1437
3438 // 1437
3439 // 1437
3440 // 1437
3441 // 1437
3442 // 1437
3443 // 1437
3444 // 1437
3445 // 1437
3446 // 1437
3447 // 1437
3448 // 1437
3449 // 1437
3450 // 1437
3451 // 1437
3452 // 1437
3453 // 1437
3454 // 1437
3455 // 1437
3456 // 1437
3457 // 1437
3458 // 1437
3459 // 1437
3460 // 1437
3461 // 1437
3462 // 1437
3463 // 1437
3464 // 1437
3465 // 1437
3466 // 1437
3467 // 1437
3468 // 1437
3469 // 1437
3470 // 1437
3471 // 1437
3472 // 1437
3473 // 1437
3474 // 1437
3475 // 1437
3476 // 1437
3477 // 1437
3478 // 1437
3479 // 1437
3480 // 1437
3481 // 1437
3482 // 1437
3483 // 1437
3484 // 1437
3485 // 1437
3486 // 1437
3487 // 1437
3488 // 1437
3489 // 1437
3490 // 1437
3491 // 1437
3492 // 1437
3493 // 1437
3494 // 1437
3495 // 1437
3496 // 1437
3497 // 1437
3498 // 1437
3499 // 1437
3500 // 1437
3501 // 1437
3502 // 1437
3503 // 1437
3504 // 1437
3505 // 1437
3506 // 1437
3507 // 1437
3508 // 1437
3509 // 1437
3510 // 1437
3511 // 1437
3512 // 1437
3513 // 1437
3514 // 1437
3515 // 1437
3516 // 1437
3517 // 1437
3518 // 1437
3519 // 1437
3520 // 1437
3521 // 1437
3522 // 1437
3523 // 1437
3524 // 1437
3525 // 1437
3526 // 1437
3527 // 1437
3528 // 1437
3529 // 1437
3530 // 1437
3531 // 1437
3532 // 1437
3533 // 1437
3534 // 1437
3535 // 1437
3536 // 1437
3537 // 1437
3538 // 1437
3539 // 1437
3540 // 1437
3541 // 1437
3542 // 1437
3543 // 1437
3544 // 1437
3545 // 1437
3546 // 1437
3547 // 1437
3548 // 1437
3549 // 1437
3550 // 1437
3551 // 1437
3552 // 1437
3553 // 1437
3554 // 1437
3555 // 1437
3556 // 1437
3557 // 1437
3558 // 1437
3559 // 1437
3560 // 1437
3561 // 1437
3562 // 1437
3563 // 1437
3564 // 1437
3565 // 1437
3566 // 1437
3567 // 1437
3568 // 1437
3569 // 1437
3570 // 1437
3571 // 1437
3572 // 1437
3573 // 1437
3574 // 1437
3575 // 1437
3576 // 1437
3577 // 1437
3578 // 1437
3579 // 1437
3580 // 1437
3581 // 1437
3582 // 1437
3583 // 1437
3584 // 1437
3585 // 1437
3586 // 1437
3587 // 1437
3588 // 1437
3589 // 1437
3590 // 1437
3591 // 1437
3592 // 1437
3593 // 1437
3594 // 1437
3595 // 1437
3596 // 1437
3597 // 1437
3598 // 1437
3599 // 1437
3600 // 1437
3601 // 1437
3602 // 1437
3603 // 1437
3604 // 1437
3605 // 1437
3606 // 1437
3607 // 1437
3608 // 1437
3609 // 1437
3610 // 1437
3611 // 1437
3612 // 1437
3613 // 1437
3614 // 1437
3615 // 1437
3616 // 1437
3617 // 1437
3618 // 1437
3619 // 1437
3620 // 1437
3621 // 1437
3622 // 1437
3623 // 1437
3624 // 1437
3625 // 1437
3626 // 1437
3627 // 1437
3628 // 1437
3629 // 1437
3630 // 1437
3631 // 1437
3632 // 1437
3633 // 1437
3634 // 1437
3635 // 1437
3636 // 1437
3637 // 1437
3638 // 1437
3639 // 1437
3640 // 1437
3641 // 1437
3642 // 1437
3643 // 1437
3644 // 1437
3645 // 1437
3646 // 1437
3647 // 1437
3648 // 1437
3649 // 1437
3650 // 1437
3651 // 1437
3652 // 1437
3653 // 1437
3654 // 1437
3655 // 1437
3656 // 1437
3657 // 1437
3658 // 1437
3659 // 1437
3660 // 1437
3661 // 1437
3662 // 1437
3663 // 1437
3664 // 1437
3665 // 1437
3666 // 1437
3667 // 1437
3668 // 1437
3669 // 1437
3670 // 1437
3671 // 1437
3672 // 1437
3673 // 1437
3674 // 1437
3675 // 1437
3676 // 1437
3677 // 1437
3678 // 1437
3679 // 1437
3680 // 1437
3681 // 1437
3682 // 1437
3683 // 1437
3684 // 1437
3685 // 1437
3686 // 1437
3687 // 1437
3688 // 1437
3689 // 1437
3690 // 1437
3691 // 1437
3692 // 1437
3693 // 1437
3694 // 1437
3695 // 1437
3696 // 1437
3697 // 1437
3698 // 1437
3699 // 1437
3700 // 1437
3701 // 1437
3702 // 1437
3703 // 1437
3704 // 1437
3705 // 1437
3706 // 1437
3707 // 1437
3708 // 1437
3709 // 1437
3710 // 1437
3711 // 1437
3712 // 1437
3713 // 1437
3714 // 1437
3715 // 1437
3716 // 1437
3717 // 1437
3718 // 1437
3719 // 1437
3720 // 1437
3721 // 1437
3722 // 1437
3723 // 1437
3724 // 1437
3725 // 1437
3726 // 1437
3727 // 1437
3728 // 1437
3729 // 1437
3730 // 1437
3731 // 1437
3732 // 1437
3733 // 1437
3734 // 1437
3735 // 1437
3736 // 1437
3737 // 1437
3738 // 1437
3739 // 1437
3740 // 1437
3741 // 1437
3742 // 1437
3743 // 1437
3744 // 1437
3745 // 1437
3746 // 1437
3747 // 1437
3748 // 1437
3749 // 1437
3750 // 1437
3751 // 1437
3752 // 1437
3753 // 1437
3754 // 1437
3755 // 1437
3756 // 1437
3757 // 1437
3758 // 1437
3759 // 1437
3760 // 1437
3761 // 1437
3762 // 1437
3763 // 1437
3764 // 1437
3765 // 1437
3766 // 1437
3767 // 1437
3768 // 1437
3769 // 1437
3770 // 1437
3771 // 1437
3772 // 1437
3773 // 1437
3774 // 1437
3775 // 1437
3776 // 1437
3777 // 1437
3778 // 1437
3779 // 1437
3780 // 1437
3781 // 1437
3782 // 1437
3783 // 1437
3784 // 1437
3785 // 1437
3786 // 1437
3787 // 1437
3788 // 1437
3789 // 1437
3790 // 1437
```

```

3540 // (ct, &v)
3541 if err == nil {
3542     return nil, err
3543 }
3544 return StringLiteral(string(bb)), nil
3545 }
3546
3547 // default:
3548 return o, nil
3549 }
3550 }
3551
3552 func dereferenceObject(ctx *Context, objectNumber int) (Object, error) {
3553     entry, ok := ctx.Find(objectNumber)
3554     if !ok {
3555         return nil, errors.New("p4cpu: dereferenceObject: unregistered object")
3556     }
3557     if entry.Compressed {
3558         err := decompressHeaderTableEntry(ctx, &HeaderTable, objectNumber, entry)
3559         if err == nil {
3560             return nil, err
3561         }
3562     }
3563     if entry.Object == nil {
3564         log.Bad.Printf("dereferenceObject: dereferencing object %d\n", objectNumber)
3565         o, err := ParseObject(ctx, entry.Offset, objectNumber, entry.Generation)
3566         if err == nil {
3567             return nil, errors.Wrap(err, "dereferenceObject: problem dereferencing object %d", objectNumber)
3568         }
3569     }
3570     if o == nil {
3571         return nil, errors.New("p4cpu: dereferenceObject: object is nil")
3572     }
3573     entry.Object = o
3574 }
3575 return entry.Object, nil
3576 }
3577
3578 func dereferenceInteger(ctx *Context, objectNumber int) (Integer, error) {
3579     o, err := dereferenceObject(ctx, objectNumber)
3580     if err == nil, err {
3581         return nil, err
3582     }
3583     i, ok := o.(Integer)
3584     if !ok {
3585         return nil, errors.New("p4cpu: dereferenceInteger: corrupt integer")
3586     }
3587 }

```

```

1573 // Do not access object's decompressHeader property: problem dereferencing object
1574 streamMd, no ref table entry", entry.objectsStream);
1575
1576 //1574
1577 //1575
1578 // Object of class entry has no objectStream
1579 sd, o = objectStream(objectStreamMd, Object(objectStreamMd))
1580
1581 //1577
1582 if !ok {
1583     return errors.Errorf("decompressHeaderTableEntry: problem dereferencing objectStreamMd, no object stream", entry.objectStream)
1584 }
1585 //1579
1586
1587 // Get indexed object from ObjectStream
1588 o, err = sd.IndexedObjectFrom(objectStreamMd)
1589 if err != nil {
1590     return errors.Errorf("decompressHeaderTableEntry: problem dereferencing object stream Md", entry.objectStream)
1591 }
1592 //1585
1593 // Save object to theHeaderEntry.
1594
1595 g := &
1596     entry.Object = o
1597     entry.Compression = HG
1598     entry.Expression = false
1599 //1592
1600 //1593
1601 // Log entry's decompressHeaderTableEntry: end, Db Md[md]:%v\n",
1602     entry.objectsStream, entry.objectStreamMd, o)
1603 //1595
1604 return nil
1605 //1596
1606 //1597
1607 // Log interesting stream content.
1608 func logStreamContent(i int) {
1609     //1600
1610     //1601
1611     switch o := o.(type) {
1612     //1602
1613     case StreamMd:
1614         //1603
1615         if o.Content == nil {
1616             //1604
1617             log.Printf("logStream: no stream content")
1618         }
1619         //1605
1620         if o.IsSpkgContent {
1621             //1606
1622             //Log_Read_Print("Content: %s\n", StreamMd.Content)
1623         }
1624         //1611
1625     case ObjectStreamMd:
1626         //1612
1627         if o.Content == nil {
1628             //1613
1629             log.Printf("logStream: no object stream content")
1630         }
1631         //1614
1632         if o.IsSpkgContent {
1633             //1615
1634             log.Printf("logStream: objectStream content = %v\n", o.Content)
1635         }
1636         //1617
1637         if o.IsObjEntry == nil {
1638             //1618
1639             log.Printf("logStream: no object stream obj entry")
1640         }
1641         //1619
1642         if o.IsObjEntry == nil {
1643             //1620
1644             log.Printf("logStream: objectStream objEntry = %v\n", o.ObjEntry)
1645         }
1646         //1621
1647     }
1648 }
1649 //1608
1650 //1609
1651 //1610
1652 //1611
1653 //1612
1654 //1613
1655 //1614
1656 //1615
1657 //1616
1658 //1617
1659 //1618
1660 //1619
1661 //1620
1662 //1621
1663 //1622
1664 //1623
1665 //1624
1666 //1625
1667 //1626
1668 //1627
1669 //1628
1670 //1629
1671 //1630
1672 //1631
1673 //1632
1674 //1633
1675 //1634
1676 //1635
1677 //1636
1678 //1637
1679 //1638
1680 //1639
1681 //1640
1682 //1641
1683 //1642
1684 //1643
1685 //1644
1686 //1645
1687 //1646
1688 //1647
1689 //1648
1690 //1649
1691 //1650
1692 //1651
1693 //1652
1694 //1653
1695 //1654
1696 //1655
1697 //1656
1698 //1657
1699 //1658
1700 //1659
1701 //1660
1702 //1661
1703 //1662
1704 //1663
1705 //1664
1706 //1665
1707 //1666
1708 //1667
1709 //1668
1710 //1669
1711 //1670
1712 //1671
1713 //1672
1714 //1673
1715 //1674
1716 //1675
1717 //1676
1718 //1677
1719 //1678
1720 //1679
1721 //1680
1722 //1681
1723 //1682
1724 //1683
1725 //1684
1726 //1685
1727 //1686
1728 //1687
1729 //1688
1730 //1689
1731 //1690
1732 //1691
1733 //1692
1734 //1693
1735 //1694
1736 //1695
1737 //1696
1738 //1697
1739 //1698
1740 //1699
1741 //1700
1742 //1701
1743 //1702
1744 //1703
1745 //1704
1746 //1705
1747 //1706
1748 //1707
1749 //1708
1750 //1709
1751 //1710
1752 //1711
1753 //1712
1754 //1713
1755 //1714
1756 //1715
1757 //1716
1758 //1717
1759 //1718
1760 //1719
1761 //1720
1762 //1721
1763 //1722
1764 //1723
1765 //1724
1766 //1725
1767 //1726
1768 //1727
1769 //1728
1770 //1729
1771 //1730
1772 //1731
1773 //1732
1774 //1733
1775 //1734
1776 //1735
1777 //1736
1778 //1737
1779 //1738
1780 //1739
1781 //1740
1782 //1741
1783 //1742
1784 //1743
1785 //1744
1786 //1745
1787 //1746
1788 //1747
1789 //1748
1790 //1749
1791 //1750
1792 //1751
1793 //1752
1794 //1753
1795 //1754
1796 //1755
1797 //1756
1798 //1757
1799 //1758
1800 //1759
1801 //1760
1802 //1761
1803 //1762
1804 //1763
1805 //1764
1806 //1765
1807 //1766
1808 //1767
1809 //1768
1810 //1769
1811 //1770
1812 //1771
1813 //1772
1814 //1773
1815 //1774
1816 //1775
1817 //1776
1818 //1777
1819 //1778
1820 //1779
1821 //1780
1822 //1781
1823 //1782
1824 //1783
1825 //1784
1826 //1785
1827 //1786
1828 //1787
1829 //1788
1830 //1789
1831 //1790
1832 //1791
1833 //1792
1834 //1793
1835 //1794
1836 //1795
1837 //1796
1838 //1797
1839 //1798
1840 //1799
1841 //1800
1842 //1801
1843 //1802
1844 //1803
1845 //1804
1846 //1805
1847 //1806
1848 //1807
1849 //1808
1850 //1809
1851 //1810
1852 //1811
1853 //1812
1854 //1813
1855 //1814
1856 //1815
1857 //1816
1858 //1817
1859 //1818
1860 //1819
1861 //1820
1862 //1821
1863 //1822
1864 //1823
1865 //1824
1866 //1825
1867 //1826
1868 //1827
1869 //1828
1870 //1829
1871 //1830
1872 //1831
1873 //1832
1874 //1833
1875 //1834
1876 //1835
1877 //1836
1878 //1837
1879 //1838
1880 //1839
1881 //1840
1882 //1841
1883 //1842
1884 //1843
1885 //1844
1886 //1845
1887 //1846
1888 //1847
1889 //1848
1890 //1849
1891 //1850
1892 //1851
1893 //1852
1894 //1853
1895 //1854
1896 //1855
1897 //1856
1898 //1857
1899 //1858
1900 //1859
1901 //1860
1902 //1861
1903 //1862
1904 //1863
1905 //1864
1906 //1865
1907 //1866
1908 //1867
1909 //1868
1910 //1869
1911 //1870
1912 //1871
1913 //1872
1914 //1873
1915 //1874
1916 //1875
1917 //1876
1918 //1877
1919 //1878
1920 //1879
1921 //1880
1922 //1881
1923 //1882
1924 //1883
1925 //1884
1926 //1885
1927 //1886
1928 //1887
1929 //1888
1930 //1889
1931 //1890
1932 //1891
1933 //1892
1934 //1893
1935 //1894
1936 //1895
1937 //1896
1938 //1897
1939 //1898
1940 //1899
1941 //1900
1942 //1901
1943 //1902
1944 //1903
1945 //1904
1946 //1905
1947 //1906
1948 //1907
1949 //1908
1950 //1909
1951 //1910
1952 //1911
1953 //1912
1954 //1913
1955 //1914
1956 //1915
1957 //1916
1958 //1917
1959 //1918
1960 //1919
1961 //1920
1962 //1921
1963 //1922
1964 //1923
1965 //1924
1966 //
```

```

2052 case obj instanceof java.lang.Object && obj instanceof java.lang.String {
2053     case Read.BinaryTableSize + w, Stream.Length
2054         ctx.Read.BinaryTableSize += w, Stream.Length
2055     case WriteStreamAnd:
2056         ctx.Write.BinaryTableSize += w, Stream.Length
2057 }
2058 }
2059 }
2060 }
2061 }
2062 }
2063 func dereferenceObject(ctx *Context, objR int) error {
2064     // objR is the object id
2065     objRTable := ctx.ObjRTable
2066     objRTableSize := len(objRTable.Table)
2067     // objR is the object id
2068     log.Read.Printf("dereferenceObject: begin, dereferenceObject %d\n", objR)
2069     entry := objRTable.Table[objR]
2070     if entry == nil {
2071         // entry is free
2072         log.Read.Printf("free object %d\n", objR)
2073         return nil
2074     }
2075     if entry.Compressed {
2076         if err := decompressObjTableEntry(objRTable, objR, entry)
2077             if err == nil {
2078                 return err
2079             }
2080     }
2081     //log.Read.Printf("dereferenceObject: decompressed entry,
2082     //dereferenceObjTableEntry, objRTable, entry, objR)
2083     return nil
2084 }
2085 }
2086 }
2087 }
2088 }
2089 }
2090 }
2091 }
2092 }
2093 }
2094 }
2095 }
2096 }
2097 }
2098 }
2099 }
2100 }
2101 }
2102 }
2103 }
2104 }
2105 }
2106 }
2107 }
2108 }
2109 }
2110 }
2111 }
2112 }
2113 }
2114 }
2115 }
2116 }
2117 }
2118 }
2119 }
2120 }
2121 }
2122 }
2123 }
2124 }
2125 }
2126 }
2127 }
2128 }
2129 }
2130 }
2131 }
2132 }
2133 }
2134 }
2135 }
2136 }
2137 }
2138 }
2139 }
2140 }
2141 }
2142 }
2143 }
2144 }
2145 }
2146 }
2147 }
2148 }
2149 }
2150 }
2151 }
2152 }
2153 }
2154 }
2155 }
2156 }
2157 }
2158 }
2159 }
2160 }
2161 }
2162 }
2163 }
2164 }
2165 }
2166 }
2167 }
2168 }
2169 }
2170 }
2171 }
2172 }
2173 }
2174 }
2175 }
2176 }
2177 }
2178 }
2179 }
2180 }
2181 }
2182 }
2183 }
2184 }
2185 }
2186 }
2187 }
2188 }
2189 }
2190 }
2191 }
2192 }
2193 }
2194 }
2195 }
2196 }
2197 }
2198 }
2199 }
2200 }
2201 }
2202 }
2203 }
2204 }
2205 }
2206 }
2207 }
2208 }
2209 }
2210 }
2211 }
2212 }
2213 }
2214 }
2215 }
2216 }
2217 }
2218 }
2219 }
2220 }
2221 }
2222 }
2223 }
2224 }
2225 }
2226 }
2227 }
2228 }
2229 }
2230 }
2231 }
2232 }
2233 }
2234 }
2235 }
2236 }
2237 }
2238 }
2239 }
2240 }
2241 }
2242 }
2243 }
2244 }
2245 }
2246 }
2247 }
2248 }
2249 }
2250 }
2251 }
2252 }
2253 }
2254 }
2255 }
2256 }
2257 }
2258 }
2259 }
2260 }
2261 }
2262 }
2263 }
2264 }
2265 }
2266 }
2267 }
2268 }
2269 }
2270 }
2271 }
2272 }
2273 }
2274 }
2275 }
2276 }
2277 }
2278 }
2279 }
2280 }
2281 }
2282 }
2283 }
2284 }
2285 }
2286 }
2287 }
2288 }
2289 }
2290 }
2291 }
2292 }
2293 }
2294 }
2295 }
2296 }
2297 }
2298 }
2299 }
2300 }
2301 }
2302 }
2303 }
2304 }
2305 }
2306 }
2307 }
2308 }
2309 }
2310 }
2311 }
2312 }
2313 }
2314 }
2315 }
2316 }
2317 }
2318 }
2319 }
2320 }
2321 }
2322 }
2323 }
2324 }
2325 }
2326 }
2327 }
2328 }
2329 }
2330 }
2331 }
2332 }
2333 }
2334 }
2335 }
2336 }
2337 }
2338 }
2339 }
2340 }
2341 }
2342 }
2343 }
2344 }
2345 }
2346 }
2347 }
2348 }
2349 }
2350 }
2351 }
2352 }
2353 }
2354 }
2355 }
2356 }
2357 }
2358 }
2359 }
2360 }
2361 }
2362 }
2363 }
2364 }
2365 }
2366 }
2367 }
2368 }
2369 }
2370 }
2371 }
2372 }
2373 }
2374 }
2375 }
2376 }
2377 }
2378 }
2379 }
2380 }
2381 }
2382 }
2383 }
2384 }
2385 }
2386 }
2387 }
2388 }
2389 }
2390 }
2391 }
2392 }
2393 }
2394 }
2395 }
2396 }
2397 }
2398 }
2399 }
2400 }
2401 }
2402 }
2403 }
2404 }
2405 }
2406 }
2407 }
2408 }
2409 }
2410 }
2411 }
2412 }
2413 }
2414 }
2415 }
2416 }
2417 }
2418 }
2419 }
2420 }
2421 }
2422 }
2423 }
2424 }
2425 }
2426 }
2427 }
2428 }
2429 }
2430 }
2431 }
2432 }
2433 }
2434 }
2435 }
2436 }
2437 }
2438 }
2439 }
2440 }
2441 }
2442 }
2443 }
2444 }
2445 }
2446 }
2447 }
2448 }
2449 }
2450 }
2451 }
2452 }
2453 }
2454 }
2455 }
2456 }
2457 }
2458 }
2459 }
2460 }
2461 }
2462 }
2463 }
2464 }
2465 }
2466 }
2467 }
2468 }
2469 }
2470 }
2471 }
2472 }
2473 }
2474 }
2475 }
2476 }
2477 }
2478 }
2479 }
2480 }
2481 }
2482 }
2483 }
2484 }
2485 }
2486 }
2487 }
2488 }
2489 }
2490 }
2491 }
2492 }
2493 }
2494 }
2495 }
2496 }
2497 }
2498 }
2499 }
2500 }
2501 }
2502 }
2503 }
2504 }
2505 }
2506 }
2507 }
2508 }
2509 }
2510 }
2511 }
2512 }
2513 }
2514 }
2515 }
2516 }
2517 }
2518 }
2519 }
2520 }
2521 }
2522 }
2523 }
2524 }
2525 }
2526 }
2527 }
2528 }
2529 }
2530 }
2531 }
2532 }
2533 }
2534 }
2535 }
2536 }
2537 }
2538 }
2539 }
2540 }
2541 }
2542 }
2543 }
2544 }
2545 }
2546 }
2547 }
2548 }
2549 }
2550 }
2551 }
2552 }
2553 }
2554 }
2555 }
2556 }
2557 }
2558 }
2559 }
2560 }
2561 }
2562 }
2563 }
2564 }
2565 }
2566 }
2567 }
2568 }
2569 }
2570 }
2571 }
2572 }
2573 }
2574 }
2575 }
2576 }
2577 }
2578 }
2579 }
2580 }
2581 }
2582 }
2583 }
2584 }
2585 }
2586 }
2587 }
2588 }
2589 }
2590 }
2591 }
2592 }
2593 }
2594 }
2595 }
2596 }
2597 }
2598 }
2599 }
2600 }
2601 }
2602 }
2603 }
2604 }
2605 }
2606 }
2607 }
2608 }
2609 }
2610 }
2611 }
2612 }
2613 }
2614 }
2615 }
2616 }
2617 }
2618 }
2619 }
2620 }
2621 }
2622 }
2623 }
2624 }
2625 }
2626 }
2627 }
2628 }
2629 }
2630 }
2631 }
2632 }
2633 }
2634 }
2635 }
2636 }
2637 }
2638 }
2639 }
2640 }
2641 }
2642 }
2643 }
2644 }
2645 }
2646 }
2647 }
2648 }
2649 }
2650 }
2651 }
2652 }
2653 }
2654 }
2655 }
2656 }
2657 }
2658 }
2659 }
2660 }
2661 }
2662 }
2663 }
2664 }
2665 }
2666 }
2667 }
2668 }
2669 }
2670 }
2671 }
2672 }
2673 }
2674 }
2675 }
2676 }
2677 }
2678 }
2679 }
2680 }
2681 }
2682 }
2683 }
2684 }
2685 }
2686 }
2687 }
2688 }
2689 }
2690 }
2691 }
2692 }
2693 }
2694 }
2695 }
2696 }
2697 }
```

```

2120         return err
2121     }
2122     //fmt.Println("pw authenticated")
2123
2124     // Prepare decrypted entry object.
2125     err = decodeObjectObject(ctxt)
2126     if err != nil {
2127         return err
2128     }
2129
2130     // For each shellEntryEntry object either by parsing from file or pass
2131     // a decrypted object.
2132     err = decodeObjectObject(ctxt)
2133     if err != nil {
2134         return err
2135     }
2136
2137     // Identify an optional Version entry in the root object/catalog.
2138     err = decodeObjectObject(ctxt)
2139     if err != nil {
2140         return err
2141     }
2142
2143     log.Root.Println("referenceCatalogTable: end")
2144
2145     return nil
2146 }
2147
2148 func handleEncryptedFile(ctxt *Context) error {
2149     err := ctxt.Cmd == DECRYPT || ctxt.Cmd == SETPASSWORDS ||
2150         return errors.New("pfcpu: this file is not encrypted")
2151 }
2152
2153 if ctxt.Cmd == DECRYPT {
2154     return nil
2155 }
2156
2157 // Encrypt subcommand found.
2158
2159 if ctxt.SubCmd == "" {
2160     return errors.New("pfcpu: please provide owner password and optional user
2161 password")
2162 }
2163
2164 return nil
2165 }
2166
2167 func lshBytes(ctxt *Context) (id []byte, err error) {
2168     if ctxt.ID == nil {
2169         return nil, errors.New("pfcpu: missing ID entry")
2170     }
2171
2172     N1, ok := ctxt.ID[0].(shellEntry)
2173     if ok {
2174         id, err = N1.Bytes()
2175         if err != nil {
2176             return nil, err
2177         }
2178     }
2179 }

```

```

1452 // If we have a stream object, found = dict.Fmt(DecompParam)
1453 if found != 0 {
1454     decodeParamArr, ok = decodeParam.(Array)
1455     if !ok {
1456         return nil, errors.New("pdcip: pdfFilterPipeline: expected decompParam
1457         array context")
1458     }
1459 }
1460
1461 // /var.PdfDict("decompParam: ba1u", decodeParamArr)
1462
1463 filterPipeline, err = buildFilterPipeline(ctx, filterArray, decodeParamArr,
1464     decodeParam)
1465 if err != nil {
1466     log.Read.Print("pdfFilterPipeline: err")
1467     return filterPipeline, err
1468 }
1469
1470 func streamDictForObj(object *Context, d Dict, objKey, streamIn int, streamFset
1471     *FileSet, ba1u *Context) (Stream, error) {
1472     streamLength, streamLengthF = d.Length()
1473     if streamLength == 0 {
1474         return sd, errors.New("pdcip: streamDictForObj: stream object without
1475         streamLength")
1476     }
1477     filterPipeline, err = pdfFilterPipeline(ctx, d)
1478     if err == nil {
1479         return sd, err
1480     }
1481     streamOffset = offset
1482
1483     // We have a stream object
1484     sd = NewStream(streamDict, streamOffset, streamLength, streamLengthF, filterPipeline
1485         *Context)
1486     log.Read.Print("streamDictForObj: end, streamObject %s\n", objKey)
1487     return sd, nil
1488 }
1489
1490 func dictCtx *Context, d Dict, objKey, err, endId, streamIn int) (d Dict, err
1491     error) {
1492     if ctx.EncKey == nil {
1493         ctx.EncKey = decryptPdfDict(d, objKey, ctx, ctx.EncKey, ctx.AES4Strings,
1494             ctx.B)
1495         if err == nil {
1496             return nil, err
1497         }
1498     }
1499
1500     if endId == 0 || (streamIn < 0 || streamIn > endId) {
1501         log.Read.Print("dict: end, %s\n", objKey)
1502         d = d
1503     }
1504 }

```

[illegible]

```

130 // @see https://github.com/ericniebler/psutil/blob/master/psutil/_psutil_linux.c
131         log.Read.PrintIn("logStream: no objectsReady to copy")
132     }
133 }
134
135 // Decode all object streams to contained objects are ready to be used.
136 void decodeObjectStreams(ctxs &ctxs) error {
137     // @see
138     // @entry "ctxs" intentionally left out.
139     // No object stream collection validation necessary.
140 }
141
142 log.Read.PrintIn("decodeObjectStreams: begin")
143
144 // Get sorted slice of object numbers.
145 void keyList()
146 for k = range cts.Read.ObjectStreams {
147     keys = append(keys, k)
148 }
149 sort.Int(keys)
150
151 for _ , objectNumber = range keys {
152     // @see ObjectReadyIndex.
153     entry = cts.Read.Index.Table(objectNumber)
154     if entry == nil {
155         return errors.Errorf("decodeObjectStreams: missing entry for objectNumber %d",
156             objectNumber)
157     }
158     log.Read.PrintIn("decodeObjectStreams: parsing object stream for objectNumber %d",
159         objectNumber)
160 }
161 // Parse object stream from file.
162 o, err = ParseObjectStream(entry.Offset, objectNumber, entry.Generation)
163 if err != nil || o == nil {
164     return errors.New("pdpcc: decodeObjectStreams: corrupt object stream")
165 }
166
167 // Ensure streamObject
168 sd, ok = p.(StreamObject)
169 if !ok {
170     return errors.New("pdpcc: decodeObjectStreams: corrupt object stream")
171 }
172
173 // Load decoded stream content to stableTable.
174 if err = loadDecodedStreamContent(ctxs, sd); err != nil {
175     return errors.Wrapf(err, "decodeObjectStreams: problem dereferencing object stream %d",
176         objectNumber)
177 }
178
179 // Save decoded stream content to stableTable.
180 if err = saveDecodedStreamContent(ctxs, sd, objectNumber, entry.Generation,
181     true); err != nil {
182     return err
183 }
184 }

```

```

2342 // err = err + ParseObject(err, 'entry.offset, objNr, entry.generation)
2343 if err == nil {
2344     return errors.Wrap(err, "dereferencedObject: problem dereferencing object id")
2345 }
2346 }
2347
2348 entry.Object = o
2349
2350 // // Linearization objects are validated and removed for stats only.
2351 err = handleLinearizationAndPurge(ctxt, o, objNr)
2352 if err == nil {
2353     return err
2354 }
2355 // // handle stream dict's
2356
2357 if err := o.ForEachStreamDict() ok {
2358     // Handle stream dict's
2359     // If err != nil, then the referencedObject object stream should already be
2360     // dereferenced at objId, objNr
2361     }
2362     if err := o.ForEachStreamDict() ok {
2363         // Handle stream dict's
2364         // If err != nil, then the referencedObject xref stream should already be
2365         // dereferenced at objId, objNr
2366         }
2367     if sd, ok := o.GetStreamDict() ok {
2368         // Handle stream dict's
2369         // If err != nil, then the referencedObject xref stream should already be
2370         // dereferenced at objId, objNr
2371         }
2372         err = loadStreamDict(ctxt, objId, objNr, entry.generation)
2373         if err == nil {
2374             return err
2375         }
2376     }
2377     entry.Object = sd
2378 }
2379
2380 log.Root().Print("dereferencedObject: and objId of %v\n", objNr,
2381     objNr, entry.Object)
2382
2383 // // logStream (entry.Object)
2384 logStream(entry.Object)
2385
2386 return nil
2387 }
2388
2389 func processBlockCounts(defTable *XRefTable, D Dict) {
2390     for i := range o {
2391         match ok := xEqTable(
2392             case IndexDict:
2393                 entry, ok := defTable.LookupTableEntry(defTable(i))
2394                 if ok {
2395                     entry.BlockCount++
2396                 }
2397             case Dict:
2398                 processBlockCounts(defTable, o)
2399             case Array:
2400                 processBlockCounts(defTable, o)
2401         }
2402     }
2403 }

```

```

2137 //
2138 } else {
2139     id, ok := ctx.ID().ID().StringLiteral()
2140     if !ok {
2141         return nil, errors.New("pdpoc: ID must contain hex literals or string
2142         literals");
2143     }
2144     id, err = Unescape(id.Value());
2145     if err != nil {
2146         return nil, err
2147     }
2148 }
2149
2150 //
2151 return id, nil
2152
2153 //
2154 func needsOwnerAndNamespace(cnd CommandNode) bool {
2155     return cnd == CHANGEIDOP || cnd == CHANGEUSER || cnd == SETPERMISSIONS
2156 }
2157
2158 //
2159 func handlePermissions(ctx *Context) error {
2160     //
2161     // AE255S Validate permissions
2162     ok, err = validatePermissions(ctx)
2163     if err != nil {
2164         return err
2165     }
2166
2167     if !ok {
2168         return errors.New("pdpoc: corrupted permissions after upw ok")
2169     }
2170
2171     // Double check existing permissions for pdpoc processing.
2172     if !hasWritePermissions(ctx.Cmd, ctx.Dst) {
2173         return errors.New("pdpoc: insufficient access permissions")
2174     }
2175
2176     return nil
2177 }
2178
2179 //
2180 func setupEncryptionKey(ctx *Context, d Dict) (err error) {
2181     //
2182     ctx.t, err = supportGetEncryption(ctx, d)
2183     if err != nil {
2184         return err
2185     }
2186
2187     ctx.t.ID, err = idbytes(ctx)
2188     if err != nil {
2189         return err
2190     }
2191
2192     var ok bool
2193
2194     // //test:Printf("poc: cks1: upw: cks1: %s", ctx.OwnerNs, ctx.IDString)
2195
2196     // Validate the owner password sha_permissions/master_password
2197     ok, err = validateOwnerPassword(ctx)
2198 }

```

[illegible][illegible]

```

1968 // Use the object stream directly for object stream reads.
1969 // If !sd, !isObject()
1970     return errors.New("pdcps: decodeObjectStream: corrupt object stream")
1971 }
1972
1973 // We have an object stream.
1974 // If !sd, err = objectStreamDict(svd)
1975 // If err == nil
1976     return errors.Wrap(err, "decodeObjectStream: problem dereferencing
1977 object stream svd", objectStream)
1978
1979 log.Read.Println("decodeObjectStream: decoding object stream SvId",
1980 objectStream)
1981
1982 // Have all objects of this object stream and save them to
1983 // ObjectStreamDict dictionary.
1984 // If err = readObjectStreamDict(svd) err == nil {
1985     return errors.Wrap(err, "decodeObjectStream: problem decoding object
1986 stream SvId", objectStream)
1987 }
1988
1989 // If ssvd.ObjectArray == nil {
1990     return errors.Wrap(err, "decodeObjectStream: objArray should be set")
1991 }
1992
1993 log.Read.Println("decodeObjectStream: decoded object stream SvId",
1994 objectStream)
1995
1996 // Save object stream dict to ssvdEntryDict.
1997 entry.Object = ssvd
1998
1999 log.Read.Println("decodeObjectStream: end")
2000
2001 return nil
2002
2003 func handleLinearizationPanicDict(cxt *Context, obj Object, objStr int) error {
2004     // If !sd, linearized
2005     // // linearization dict already processed.
2006     return nil
2007 }
2008
2009 // handle Linearization panic dict.
2010 // If d == c == obj (dict) obj == d, if !linearizationPanicDict(c) {
2011     handle.Linearization = true
2012     c.linearizationPanicDict[objStr] = true
2013     log.Read.Println("handleLinearizationPanicDict: identified linearizationObj
2014 SvId", c)
2015
2016     a := d.ArrayEntry("sv")
2017
2018 }

```

```

2020:
2021:
2022: func processArrayByCounts(x:Iterable, xObjTable, a Array) {
2023:   for _ in range a {
2024:     switch o in a.CType() {
2025:     case IndirectType:
2026:       entry, ok = xObjTable.FindTableIndirect(o)
2027:       if ok {
2028:         entry.RefCount++
2029:       }
2030:     case CInt:
2031:       processRefCounts(xObjTable, o)
2032:     case Array:
2033:       processRefCounts(xObjTable, o)
2034:     }
2035:   }
2036: }
2037:
2038: func processRefCounts(xObjTable *XRefTable, o Object) {
2039:   switch o in o.CType() {
2040:   case CInt:
2041:     processIndirectCounts(xObjTable, o)
2042:   case StringInd:
2043:     processIndirectCounts(xObjTable, o.Dict)
2044:   case Array:
2045:     processArrayByCounts(xObjTable, o)
2046:   }
2047: }
2048:
2049: // Performance analysis: counts including unmanaged objects from object streams.
2050: func debugRefCounts(cctx *Context) error {
2051:   log.Root.Println("debugRefCounts: begin")
2052:   xRefTable = cctx.XRefTable
2053:   // Use sorted list of object numbers.
2054:   // TODO: Skip sorting for performance gain.
2055:   keys := List
2056:   for k in xRefTable.Table {
2057:     keys = append(keys, k)
2058:   }
2059:   sort.Ints(keys)
2060:
2061:   for _, objNr := range keys {
2062:     err = deferenciateObject(cctx, objNr)
2063:     if err != nil {
2064:       return err
2065:     }
2066:   }
2067:
2068:   for _, objNr := range keys {
2069:     entry = xRefTable.Table[objNr]
2070:     if entry.ref != entry.compressed {
2071:       continue
2072:     }
2073:     processRefCounts(xRefTable, entry.obj.Seal)
2074:   }

```

```

2530 //
2531 if err == nil {
2532     return err
2533 }
2534 // If the owner password does not match we generally move on if the user password
2535 // errors
2536 // unless we need to limit on a correct owner password due to the specific
2537 // mount in use (password)
2538 if !ok {
2539     return errors.New("password: incorrect password")
2540 }
2541 return errors.New("password: please provide the master password with 'opw'")
2542 }
2543 //
2544 // Generally the user password (which is also regarded as the master password or
2545 // pre-password)
2546 // is sufficient for moving on. A password change is an exception since it
2547 // requires the master password.
2548 if ok {
2549     return handlePermissions(ctx, Cmd)
2550 }
2551 ok, err = validatePermissions(ctx)
2552 if err == nil {
2553     return err
2554 }
2555 // If ok {
2556     return errors.New("password: corrupted permissions after opw ok")
2557 }
2558 return nil
2559 }
2560 //
2561 // Validate the user password (opw, document open password,
2562 // pre-password)
2563 ok, err = validatePermissions(ctx)
2564 if err == nil {
2565     return err
2566 }
2567 // If ok {
2568     return errors.New("password: please provide the correct password")
2569 }
2570 //
2571 // If not, print("type ok: %t\n", ok)
2572 //
2573 //
2574 return handlePermissions(ctx)
2575 }
2576 //
2577 func checkForEncryption(c *Context) error {
2578     //
2579     //
2580     //
2581     //
2582     //
2583     //
2584     //
2585     //
2586     //
2587     //
2588     //
2589     //
2590     //
2591     //
2592     //
2593     //
2594     //
2595     //
2596     //
2597     //
2598     //
2599     //
2600     //
2601     //
2602     //
2603     //
2604     //
2605     //
2606     //
2607     //
2608     //
2609     //
2610     //
2611     //
2612     //
2613     //
2614     //
2615     //
2616     //
2617     //
2618     //
2619     //
2620     //
2621     //
2622     //
2623     //
2624     //
2625     //
2626     //
2627     //
2628     //
2629     //
2630     //
2631     //
2632     //
2633     //
2634     //
2635     //
2636     //
2637     //
2638     //
2639     //
2640     //
2641     //
2642     //
2643     //
2644     //
2645     //
2646     //
2647     //
2648     //
2649     //
2650     //
2651     //
2652     //
2653     //
2654     //
2655     //
2656     //
2657     //
2658     //
2659     //
2660     //
2661     //
2662     //
2663     //
2664     //
2665     //
2666     //
2667     //
2668     //
2669     //
2670     //
2671     //
2672     //
2673     //
2674     //
2675     //
2676     //
2677     //
2678     //
2679     //
2680     //
2681     //
2682     //
2683     //
2684     //
2685     //
2686     //
2687     //
2688     //
2689     //
2690     //
2691     //
2692     //
2693     //
2694     //
2695     //
2696     //
2697     //
2698     //
2699     //
2700     //
2701     //
2702     //
2703     //
2704     //
2705     //
2706     //
2707     //
2708     //
2709     //
2710     //
2711     //
2712     //
2713     //
2714     //
2715     //
2716     //
2717     //
2718     //
2719     //
2720     //
2721     //
2722     //
2723     //
2724     //
2725     //
2726     //
2727     //
2728     //
2729     //
2730     //
2731     //
2732     //
2733     //
2734     //
2735     //
2736     //
2737     //
2738     //
2739     //
2740     //
2741     //
2742     //
2743     //
2744     //
2745     //
2746     //
2747     //
2748     //
2749     //
2750     //
2751     //
2752     //
2753     //
2754     //
2755     //
2756     //
2757     //
2758     //
2759     //
2760     //
2761     //
2762     //
2763     //
2764     //
2765     //
2766     //
2767     //
2768     //
2769     //
2770     //
2771     //
2772     //
2773     //
2774     //
2775     //
2776     //
2777     //
2778     //
2779     //
2780     //
2781     //
2782     //
2783     //
2784     //
2785     //
2786     //
2787     //
2788     //
2789     //
2790     //
2791     //
2792     //
2793     //
2794     //
2795     //
2796     //
2797     //
2798     //
2799     //
2800     //
2801     //
2802     //
2803     //
2804     //
2805     //
2806     //
2807     //
2808     //
2809     //
2810     //
2811     //
2812     //
2813     //
2814     //
2815     //
2816     //
2817     //
2818     //
2819     //
2820     //
2821     //
2822     //
2823     //
2824     //
2825     //
2826     //
2827     //
2828     //
2829     //
2830     //
2831     //
2832     //
2833     //
2834     //
2835     //
2836     //
2837     //
2838     //
2839     //
2840     //
2841     //
2842     //
2843     //
2844     //
2845     //
2846     //
2847     //
2848     //
2849     //
2850     //
2851     //
2852     //
2853     //
2854     //
2855     //
2856     //
2857     //
2858     //
2859     //
2860     //
2861     //
2862     //
2863     //
2864     //
2865     //
2866     //
2867     //
2868     //
2869     //
2870     //
2871     //
2872     //
2873     //
2874     //
2875     //
2876     //
2877     //
2878     //
2879     //
2880     //
2881     //
2882     //
2883     //
2884     //
2885     //
2886     //
2887     //
2888     //
2889     //
2890     //
2891     //
2892     //
2893     //
2894     //
2895     //
2896     //
2897     //
2898     //
2899     //
2900     //
2901     //
2902     //
2903     //
2904     //
2905     //
2906     //
2907     //
2908     //
2909     //
2910     //
2911     //
2912     //
2913     //
2914     //
2915     //
2916     //
2917     //
2918     //
2919     //
2920     //
2921     //
2922     //
2923     //
2924     //
2925     //
2926     //
2927     //
2928     //
2929     //
2930     //
2931     //
2932     //
2933     //
2934     //
2935     //
2936     //
2937     //
2938     //
2939     //
2940     //
2941     //
2942     //
2943     //
2944     //
2945     //
2946     //
2947     //
2948     //
2949     //
2950     //
2951     //
2952     //
2953     //
2954     //
2955     //
2956     //
2957     //
2958     //
2959     //
2960     //
2961     //
2962     //
2963     //
2964     //
2965     //
2966     //
2967     //
2968     //
2969     //
2970     //
2971     //
2972     //
2973     //
2974     //
2975     //
2976     //
2977     //
2978     //
2979     //
2980     //
2981     //
2982     //
2983     //
2984     //
2985     //
2986     //
2987     //
2988     //
2989     //
2990     //
2991     //
2992     //
2993     //
2994     //
2995     //
2996     //
2997     //
2998     //
2999     //
3000     //
3001     //
3002     //
3003     //
3004     //
3005     //
3006     //
3007     //
3008     //
3009     //
3010     //
3011     //
3012     //
3013     //
3014     //
3015     //
3016     //
3017     //
3018     //
3019     //
3020     //
3021     //
3022     //
3023     //
3024     //
3025     //
3026     //
3027     //
3028     //
3029     //
3030     //
3031     //
3032     //
3033     //
3034     //
3035     //
3036     //
3037     //
3038     //
3039     //
3040     //
3041     //
3042     //
3043     //
3044     //
3045     //
3046     //
3047     //
3048     //
3049     //
3050     //
3051     //
3052     //
3053     //
3054     //
3055     //
3056     //
3057     //
3058     //
3059     //
3060     //
3061     //
3062     //
3063     //
3064     //
3065     //
3066     //
3067     //
3068     //
3069     //
3070     //
3071     //
3072     //
3073     //
3074     //
3075     //
3076     //
3077     //
3078     //
3079     //
3080     //
3081     //
3082     //
3083     //
3084     //
3085     //
3086     //
3087     //
3088     //
3089     //
30
```