GIAC
CERTIFICATIONS

Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

Interested in learning more?

Check out the list of upcoming events offering
"Automating Information Security with Python (Security 573)"
at http://www.giac.org/registration/gpyc

PDF Metadata Extraction with Python

Author: Christopher A. Plaisance, nxl4@protonmail.com
Advisor: Rob VandenBrink
Accepted: 5 February 2019

Abstract

This paper explores techniques for programmatically extracting metadata from PDF files using Python. It begins by detailing the internal structure of PDF documents, focusing on the internal system of indirect references and objects within the PDF binary, the document information dictionary metadata type, and the XMP metadata type contained in the file's metadata streams. Next, the paper explores the most common means of accessing PDF metadata with Python, the high-level `PyPDF` and `PyPDF2` libraries. This examination discovers deficiencies in the methodologies used by these modules, making them inappropriate for use in digital forensics investigations. An alternative low-level technique of carving the PDF binary directly with Python, using the re module from the standard library is described, and found to accurately and completely extract all of the pertinent metadata from the PDF file with a degree of completeness suitable for digital forensics use cases. These low-level techniques are built into a stand-alone open source Linux utility, `pdf-metadata`, which is discussed in the paper's final section.

1. Introduction

Since the introduction of the standard in 1993 (Adobe Systems Incorporated, 2001, p. xix), Adobe's Portable Document Format (PDF) has become one of the most widely adopted and commonly used file formats for creating and storing documents. Contrasted against plain text document formats, the PDF file type is intrinsically rich in metadata artifacts, which can be valuable to recover during a digital forensic investigation. While there are a number of ways to extract these metadata from a PDF file, these techniques typically have two features in common: they are manual processes which do not natively lend themselves to automation, and they do not encompass all of the metadata artifacts which can be hidden within the file's binary structure.

For example, in Larry Pesce's paper on *Document Metadata* (Pesce, 2008), he provides instructions on extracting PDF metadata using either the Adobe Acrobat viewer's Document Properties window, or the Linux strings command line operation (pp. 11–15). In the case of using a graphical tool like Adobe Acrobat, the forensic analyst is immediately faced with a challenge to automation, as this is a process which is *designed* to be manual. Moreover, the metadata presented in a graphic user interface (GUI) tool may not be exhaustive of the artifacts contained within the file's binary structure. The command-line interface (CLI) option has the benefit of delving more directly into the file's internal data structures, yielding a more complete picture of the file's metadata; however, the unstructured format of the data returned by the command does not lend itself either to analysis or automation.

In his book, *Violent Python*, T.J. O'Connor (O'Connor, 2013) presents a Python-based approach to the challenge that has the possibilities for both exhausting the file's metadata structures and lending itself to automation. However, O'Connor's utility presented in the book (pp. 93–95), only extracts one of the two metadata structures concealed within PDF files, and is written as a utility for use on a single file, without the native flexibility to be used on large collections of PDFs. Moreover, although the library O'Connor uses for his script, PyPDF, is capable of interfacing with both metadata structures (as is its Python 3 port, PyPDF2), it does not handle these artifacts with the rigor needed for a digital forensics investigation. As will be discussed in detail below, the metadata

extraction capabilities of this library do not deal appropriately with PDFs containing multiple metadata artifacts of either type, or with artifacts whose elements do not exactly match certain predetermined dictionaries.

This gap in the forensic analyst's toolkit needs to be closed, and Python lends itself easily to engineer a solution. In this paper, we will examine the different metadata structures that are embedded within PDF files: looking in detail at both the Document Information Dictionary and the Extensible Metadata Platform. Once these structural elements have been examined, we will address the ways in which these metadata can be accessed with Python. This treatment will include both higher-level techniques which involve the use of PDF-specific libraries, as well as lower-level techniques for carving the binary directly. These techniques of metadata extraction will also be discussed within the context of automation, to address situations where a forensic analyst may be faced with thousands of files to analyze. And, all of these strains are tied together in an open source Python utility, `pdf-metadata.py`, which facilitates the bulk extraction of metadata from PDF documents. The presented solution aims to exhaust all available metadata structures contained within the PDF, provide functionalities to decrypt the metadata of encrypted PDFs, and facilitate the batch extraction of any number of PDF files in sequence.

2. PDF Metadata Structures

2.1. What is a PDF?

There are three primary entity types that are included under the PDF aegis: the PDF *file format*, PDF *documents*, and PDF *files* (Adobe Systems Incorporated, 2001, p. 9). At its core, PDF is a *file format* which is designed to reliably and consistently represent documents in a way that is agnostic to both the software, hardware, and operating system of the document's creator as well as those of the viewer or printer. In this vein, a PDF *document* is composed of objects which define the appearance of pages, using the PDF syntax. Within this context, a PDF *file* can then be defined as the self-contained binary container of all the objects and structural elements which constitute the PDF document. The PDF language was based on, and remains similar to, the PostScript page description

Figure 1: Linux NTFS File System Metadata.

language; however, the similarities between the two does not result in straightforward conversion (Adobe Systems Incorporated, 2001, p. 21; Adobe Systems Incorporated, 1999, p. 9). Since 2008, the PDF format has been standardized as an open format, under ISO 32000 (International Organization for Standardization, 2017).

2.2. What is Metadata?

At its essence, *metadata* itself is near universally defined as "data about data" (Gill, 2008, p. 20; Pittman and Shaver, 2010, p. 232; Sammons, 2012, p. 72; Plaisance, 2016, p. 48). The metadata of digital objects can generally be divided into two categories: file system metadata, and application metadata (Gilliland, 2008, p. 10; Pittman and Shaver, 2010, p. 232; Sammons, 2012, p. 72; Plaisance, 2016, p. 49). File system metadata include those data elements which are *extrinsic* to the file itself, and *do not* participate in the byte-sequence that constitutes the file's binary structure. Conversely, application metadata includes those elements which are *intrinsic* to the file, and *do* participate in the binary's byte-sequence. In distinguishing between the two metadata types, the difference comes down to what *primary* source is being interrogated to retrieve the metadata.

File system metadata is retrieved by interrogating the *file system* itself, typically through a command line shell. For example, the `stat` command in most UNIX and Unix-like shells will display all of the available file system metadata belonging to a given file (see Figure 1).[1] While the specific file system metadata elements available are *file system*

---

[1] All of the examples given in sections 2 through 3 demonstrate the forensic techniques on the PDF file for my article, *Methods of Web Philology*, cited here as Plaisance 2016.

Figure 2: Windows FAT32 file system metadata.

and operating system specific (i.e. the file system metadata available in Windows for a file on a NTFS formatted disk will differ from the file system metadata available in Linux for the same file stored on a FAT32 formatted disk (for a Windows NTFS example, see Figure 2, which retrieves file system metadata using PowerShell's `Get-Item` and `Format-List` commands).

2.3. Accessing PDF Application Metadata

From a layman's perspective, the question of accessing the application metadata of a PDF file may seem as trivial as accessing the Properties window of one's chosen PDF viewer application (see Figure 3). Indeed, this is undoubtedly the most common technique for accessing a PDF file's application metadata. However, as it will become readily apparent as individual files are analyzed during the course of this paper, the metadata presented through this technique are *far* from exhaustive, making this completely unsuitable for any serious forensic investigation.
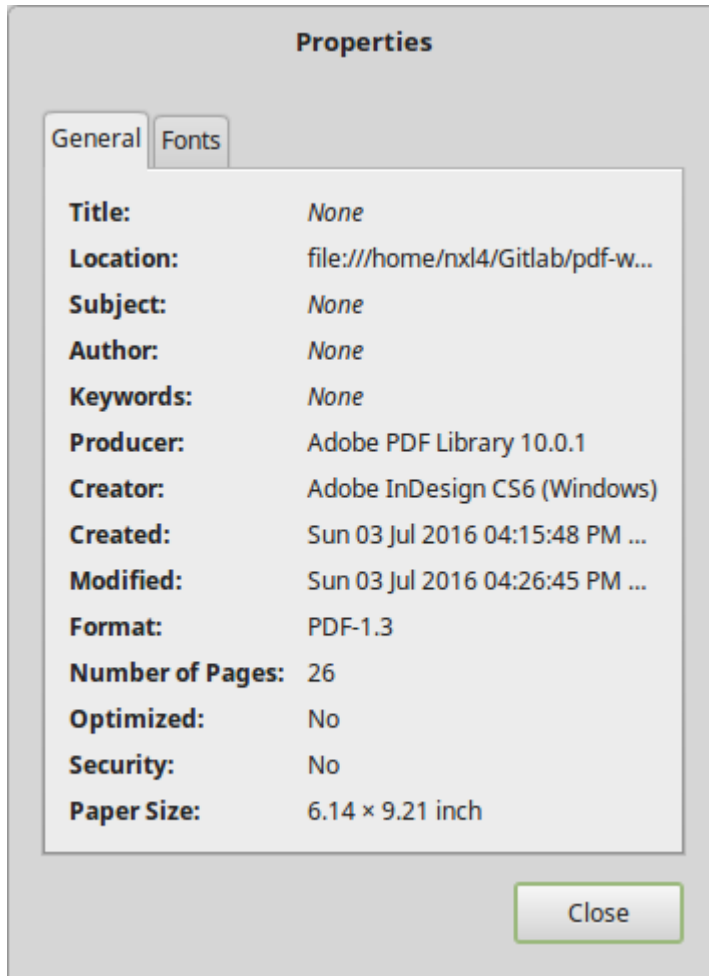
Figure 3: PDF Metadata Visible in Properties Window of Xreader

In order to understand how we can programmatically extract application metadata from a PDF file in a way that is suitable for forensics work, the first step is to investigate the general structure of the PDF file format. PDF files are typically large, and can contain thousands of constituent objects, which makes imperative for a forensic investigator to know what objects within a PDF file contain application metadata and how to locate these objects within a given PDF file. In addressing the first question, Adobe's documentation proves a useful starting point, identifying two different ways in which application metadata are stored within a PDF file, as either a *document information dictionary* object, or a *metadata stream* object (Adobe Systems Incorporated, 2008, p. 548).

Within the binary structure of a PDF file, the document information dictionary is a special class of a general type, the *indirect object*. All indirect objects are labeled with a two-part identifier, consisting of an *object number* and a *generation number*. The indirect

```
nxl4@wintermute ~/Gitlab/pdf-work                    _   □   ✕

File  Edit  View  Search  Terminal  Help
 35 0000000938 00000 n^M
 36 trailer^M
 37 <</Size 1210/Root 1180 0 R/Info 280 0 R/ID[<8890810FBE3C914495
    CCB63388839EE3><B9E36CF293C5B640A9FBFB2D215AE830>]/Prev 192792
    /XRefStm 2133>>^M
 38 startxref^M
 39 0^M
 40 %%EOF^M
                                            40,1              0%
```

Figure 4: Document Information Dictionary Reference in PDF File

object is thus defined by a structure consisting of its object and generation number, followed by two bracketing keywords—`obj` and `endobj`—with the object itself occurring between the brackets (Adobe Systems Incorporated, 2001, p. 39). A minimal example of this can be given as:

```
11 0 obj
     (Hello world)
endobj
```
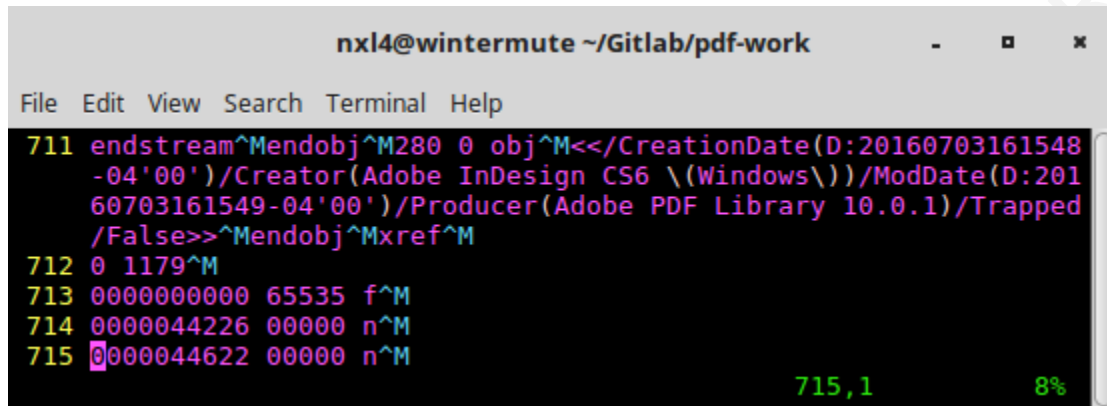
While this structural description of the indirect object *does* give us a pattern that could be used to programmatically locate objects within a PDF file, it does *not* tell us anything about what kind of object is being located. To address this exact concern, and to enable applications to parse PDF documents, each indirect object contains an *indirect reference* which—for many types of indirect objects, including document information dictionaries—will identify the object type. The indirect reference will contain the object and generation number which identify the specific object followed by the keyword `R` and preceded by the keyword (always beginning with a forward-slash) identifying the object type (Adobe Systems Incorporated, 2001, p. 40). For example, the above minimal example could be identified as:

```
/Hello 11 0 R
```

Figure 5: Document Information Dictionary Object in PDF File.

This combination of the indirect object and its reference provides the key to locating any object those referential keyword is known.

2.4. Document Information Dictionary

The keyword for the document information dictionary reference is `/Info`, and this reference is typically contained within a special object called the `trailer`—which is itself a collection of keywords bracketed between the `trailer` and end-of-file marker, `%%EOF` (Adobe Systems Incorporated, 2001, pp.67–78). On an *ad hoc* basis, by opening the PDF file in question with Vim, and using the text editor's search function to locate the `/Info` reference (see Figure 4). This reference, in this case `/Info 280 0 R`, can be used to search in Vim for the corresponding object by replacing the `R` keyword with an `obj` keyword, `280 0 obj` (see Figure 5). The object that this query will return will contain an additional pair of << and >> brackets within the `obj` and `endobj` brackets. Within those inner angle brackets will be an array of key value pairs. A minimal example of this can be given as:

```
11 0 obj
    <<    /Title (Document Title)
          /Author (Document Author)
    >>
endobj
```

Figure 6: XMP Metadata Reference in PDF File.

There are a total of nine approved keywords detailed for use with document information dictionary objects in the documentation (Adobe Systems Incorporated, 2001, p. 576):

Key Name	Data Type	Value Description
/Title	Text String	Title of the Document
/Author	Text String	Author of the Document
/Subject	Text String	Subject of the Document
/Keywords	Text String	Keywords Associated with the Document
/Creator	Text String	Application Used to Create the Original Document (Pre-PDF Conversion)
/Producer	Text String	Application Used to Convert Original (Pre-PDF) Document to PDF
/CreationDate	Date	Date and Time of Document's Creation
/ModDate	Date	Date and Time of Document's Last Modification
/Trapped	Name Object	Indicated if Document Had Been Modified to Include Trapping Information

2.5. Metadata Streams

The keyword for the metadata streams reference is /Metadata, with this reference generally being located in the document catalog—another special object which consists of an angle bracketed array of referential keywords (Adobe Systems Incorporated, 2001, p. 578). As with the document information dictionary reference, this reference can be located on an *ad hoc* basis by querying the /Metadata reference with the PDF file open in Vim (see Figure 6). In this case, the reference /Metadata 279 0 R can be used to locate the

Figure 7: XMP Metadata Object in PDF File.

corresponding metadata streams object, by searching the same open Vim document for `279 0 obj` (see Figure 7).

The resulting object is represented in a subset of the Extensible Markup Language (XML) framework called the Extensible Metadata Platform (XMP), which provides a highly customizable and adaptable structure into which metadata can be added to a PDF file (Adobe Systems Incorporated, 2001, p. 578). The XMP framework is too mutable and complex to provide a minimal example; however, the full format specification can be found in Adobe Systems Incorporated 2012, 2014, and 2016.

3. Metadata Extraction with PyPDF2

While it has been shown now that it is possible to locate both types of application metadata contained in a PDF file by first locating the reference for the respective keywords, and then locating the objects those object and generation numbers match the previously located references. However, this is a highly manual and time-consuming process that

```
In [1]:  import pprint
         import PyPDF2
```

Figure 8: PyPDF2 Module Import.

leaves plenty of room for human error, and does not scale. What is required, then, is an automated utility that can replicate the above manual parsing and extraction techniques. To begin, we will explore techniques for extracting these metadata types using the most common Python modules, PyPDF and PyPDF2.[2]

3.1. Extracting Document Information Dictionary Metadata

To extract the Document Information Dictionary metadata from a given PDF file, with the PyPDF2 technique, the first step will be to import two libraries: the PyPDF2 module which will facilitate the interface with the PDF file, and the pprint module will beautify the output dictionary when printed to the CLI (see Figure 8). Following the module imports, the next step is to define a function that will call two of the module's core methods.

```
In [2]:  def get_doc_info(file_name):
             pp = pprint.PrettyPrinter(indent=4)
             pdf_file = PyPDF2.PdfFileReader(file_name, 'rb')
             doc_info = pdf_file.getDocumentInfo()
             pp.pprint(doc_info)
```

Figure 9: PyPDF2 Function to Extract PDF's Document Information Dictionary.

First, the PdfFileReader() method accepts the PDF file path as an argument, which "initializes a PdfFileReader object" (Fenniak, 2016c). This object should be variablized, for use with subsequent methods. Second, the getDocumentInfo() method can then be used on the newly initialized PdfFileReader object. This creates a new DocumentInformation object, which is structured as a Python dictionary. These constituent dictionary will contain a number of key:value pairs, with the value returning as a TextStringObject if PyPDF2 is able to decode the string's encoding, or returning as

---

[2] At the time the afore referenced works on doing PDF forensics with Python were written, Python 2 was the standard, making PyPDF the module of choice. However, as Python 3 is now the de facto standard (with Python 2's deprecation on this horizon), PyPDF's Python 3 successor, PyPDF2 has superseded it in terms of utility and popularity. In terms of functionality, however, the two modules are all but identical.

```
In [3]:  get_doc_info("methods_of_web_philology.pdf")

         {   '/CreationDate': "D:20160703161548-04'00'",
             '/Creator': 'Adobe InDesign CS6 (Windows)',
             '/ModDate': "D:20160703162645-04'00'",
             '/Producer': 'Adobe PDF Library 10.0.1',
             '/Trapped': '/False'}
```

Figure 10: Example Extraction of Document Information Dictionary.

```
In [6]:  def get_xmp_info(file_name):
             pp = pprint.PrettyPrinter(indent=4)
             pdf_file = PyPDF2.PdfFileReader(file_name, 'rb')
             pdf_xmp = pdf_file.getXmpMetadata()
             xmp_methods = ['custom_properties', 'dc_contributor',
                            'dc_coverage', 'dc_creator',
                            'dc_date', 'dc_description',
                            'dc_format', 'dc_identifier',
                            'dc_language', 'dc_publisher',
                            'dc_relation', 'dc_rights',
                            'dc_source', 'dc_subject',
                            'dc_title', 'dc_type',
                            'pdf_keywords', 'pdf_pdfversion',
                            'pdf_producer', 'xmp_createDate',
                            'xmp_creatorTool', 'xmp_metadataDate',
                            'xmp_modifyDate', 'xmpmm_documentId',
                            'xmpmm_instanceId']
             xmp_dict = {}
             for i in xmp_methods:
                 try:
                     xmp_dict[i] = getattr(pdf_xmp, i)
                 except:
                     xmp_dict[i] = ''
             pp.pprint(xmp_dict)
```

Figure 11: PyPDF2 Function to Extract PDF's XMP Metadata.

a `ByteStringObject` if `PyPDF2` is unable to decode the string (Fenniak, 2016b). Putting these two methods together yields a custom function that can be used to extract document information dictionary metadata from PDF files (see Figure 9). The resulting `DocumentInformation` object which generated by the custom `get_doc_info()` function contains a dictionary with five key:value pairs (see Figure 10). This extracted data matches the raw metadata located in the Document Information Dictionary object located at `208 0 obj` in the file (see Figure 5).

```
In [7]: get_xmp_info("methods_of_web_philology.pdf")

{   'custom_properties': {},
    'dc_contributor': [],
    'dc_coverage': None,
    'dc_creator': [],
    'dc_date': [],
    'dc_description': {},
    'dc_format': 'application/pdf',
    'dc_identifier': None,
    'dc_language': [],
    'dc_publisher': [],
    'dc_relation': [],
    'dc_rights': {},
    'dc_source': None,
    'dc_subject': [],
    'dc_title': {},
    'dc_type': [],
    'pdf_keywords': None,
    'pdf_pdfversion': None,
    'pdf_producer': 'Adobe PDF Library 10.0.1',
    'xmp_createDate': datetime.datetime(2016, 7, 3, 20, 15, 48),
    'xmp_creatorTool': 'Adobe InDesign CS6 (Windows)',
    'xmp_metadataDate': datetime.datetime(2016, 7, 3, 20, 26, 45),
    'xmp_modifyDate': datetime.datetime(2016, 7, 3, 20, 26, 45),
    'xmpmm_documentId': 'xmp.id:72EF63A05A41E611855AE2067B83E76F',
    'xmpmm_instanceId': 'uuid:672a6a97-7e47-48ed-95f4-443d58650879'}
```

Figure 12: Example Extraction of XMP Metadata.

3.2. Extracting XMP Metadata

Using the same previously imported libraries, PyPDF2 and pprint, XMP metadata can be extracted from a given PDF file by invoking one additional method from the core PyPDF2 library. After creating a PdfFileReader object with the eponymous method, the getXmpMetadata() method can be used on this object. This will create a new XmpInformation object, which will contain any XMP metadata that the PyPDF2 module was able to extract. To access these extracted metadata, we will define a custom list of string variables which represent the exhaustive collection of methods which can be used on the newly created XmpInformation object. As outlined in the module's documentation, there are a total of twenty-five XMP metadata values which have defined methods associated (Fenniak, 2016d). By iterating over the string representation of these

methods contained in the `xmp_methods` list, and inserting the iterator into a `getattr()` function, thus calling the method from the `XmpInformation` object, the contents of this object can be easily exported into a dictionary—which can itself then be printed. We can define a custom `get_xmp_info()` function which combines all of these methods, to print the XMP metadata which `PyPDF2` is able to extract from a given PDF file (see Figure 11). This extracted data (see Figure 12) matches the raw metadata located in the Document Information Dictionary object located at `279 0 obj` in the file (see Figure 7). The twenty-five possible metadata values yielded by the custom `get_xmp_info()` function are given in the following table:

Key Name	Data Type	Value Description
custom_properties	Dictionary	Custom Metadata Properties
dc_contributor	List	Non-Authorial Contributors to the Document
dc_coverage	List	Describes the Scope or Extent of the Document
dc_creator	List	Names of Document's Authors
dc_date	List	Datetime Object of Significance to the Document
dc_description	Dictionary	Descriptions of the Document's Contents
dc_format	String	Document's MIME-Type
dc_identifier	String	Document's Unique Identifier
dc_language	List	Languages Used in the Document
dc_publisher	List	Publisher of the Document
dc_relation	List	Relationships to Other Documents
dc_rights	Dictionary	User's Rights to the Document
dc_source	String	Unique Identifier of the Document's Source
dc_subject	List	Keywords Indicating Document's Subject
dc_title	Dictionary	Document's Title
dc_type	List	Description of Document's Type
pdf_keywords	String	Additional Listing of Document's Keywords
pdf_pdfversion	String	PDF's Version
pdf_producer	String	Tool that Created PDF Document
xmp_createDate	String	Date the Document was Created
xmp_creatorTool	String	First Tool Used to Create the Document's Source
xmp_metadataDate	Datetime Object	Most Recent Change Date of Meatdata
xmp_modifyDate	Datetime Object	Most Recent Change Date of Document
xmpmm_documentId	String	Common Identifier for All Versions of the Document

xmpmm_instanceId	String	Unique Identifier for this Particular Document

4. Problems with PyPDF2

With the two custom functions defined above, `get_doc_info()` and `get_xmp_info()`, it would be a relatively simple matter to build a script that could ingest either a single PDF file or iterate through a collection of PDFs and export the metadata into some structured format like CSV, JSON, etc. Indeed, in the initial draft of this present project that is exactly what I did. However, after running several hundred PDF files through this `PyPDF2` based application, I discovered two serious problems with the module. These issues led to a complete re-thinking of this project's programming methodology—as the deficiencies inherit in the initial `PyPDF2` approach would render the resulting application useless for serious forensic investigations. Both of these problems manifest when analyzing a PDF copy of Justin Seitz's book, *Black Hat Python* (2014). So, this PDF file will serve as this section's focus of examination in demonstrating the problems PyPDF2 poses for the forensic investigator.

4.1. Inability to Extract Multiple Metadata Objects

```
In [4]: get_doc_info("bhp.pdf")

{   '/Author': 'Justin Seitz',
    '/Keywords': 'Python, Hacking, Security, Programming, Pentesting',
    '/ModDate': 'D:20141209024746Z',
    '/Subject': 'Python Programming for Hackers and Pentesters',
    '/Title': 'Black Hat Python'}
```

Figure 13: Document Information Dictionary Extracted with PyPDF2 from Black Hat Python.

When the `PyPDF2`-based `get_doc_info()` function is used to extract the document information dictionary from the *Black Hat Python* PDF document, `bhp.pdf`, it locates a single object (see Figure 13). However, manual searches for the document information dictionary references and their corresponding objects done manually through

Figure 14: Document Information Dictionary References and Objects in Black Hat Python PDF.

Vim (see Figure 14) reveal not one, but two different pairs of references (`/Info 610 0 R` and `/Info 1969 0 R`) and objects (`610 0 obj` and `1969 0 obj`). The data extracted via `PyPDF2` matches `1969 0 obj`, and the reason for this is simple: of the two document information dictionary reference-object pairs embedded in the PDF file, `610 0` represents the original object, while `1969 0` is a subsequent modification. This is by design, as the document information dictionary located at `1969 0 obj` represents the authoritative version of the document's metadata. This is the metadata object that is intended to be

Properties

General | Fonts

Title:	Black Hat Python
Location:	file:///home/nxl4/Gitlab/pdf-w...
Subject:	Python Programming for Hac...
Author:	Justin Seitz
Keywords:	Python, Hacking, Security, Pro...
Producer:	*None*
Creator:	*None*
Created:	Wed 31 Dec 1969 06:59:59 P...
Modified:	Mon 08 Dec 2014 09:47:46 P...
Format:	PDF-1.4
Number of Pages:	193
Optimized:	No
Security:	No
Paper Size:	US Letter, Portrait (8.50 × 11....

Close

Figure 15: Metadata Presented in the Properties Window of Black Hat Python as Viewed in Xreader.

identified by PDF viewers when they parse the document, as is evidenced by the fact that the metadata seen in the Properties window of this file as opened in Xreader contain all of these metadata elements culled from the `1969 0 obj` object. There is one additional element in the Properties window, Created, which maps to the `/CreationDate` keyword in the `610 0 obj` object; this is included because the `1969 0 obj` object is a modification of the `610 0 obj` object, and only contains a modification date, not a creation date. Each change that the document's creator makes to the document information dictionary results in the creation of a new reference-object pair in the file. However, it is only the most recent object that will be extracted by the `PyPDF2` module. This selection of the most recent object out of an historical sequence is not limited to the document information dictionary reference-object pairs, but also includes the XMP reference-object pairs that constitute the metadata streams category discussed above.

```
In [8]: get_xmp_info("bhp.pdf")

{   'custom_properties': '',
    'dc_contributor': '',
    'dc_coverage': '',
    'dc_creator': '',
    'dc_date': '',
    'dc_description': '',
    'dc_format': '',
    'dc_identifier': '',
    'dc_language': '',
    'dc_publisher': '',
    'dc_relation': '',
    'dc_rights': '',
    'dc_source': '',
    'dc_subject': '',
    'dc_title': '',
    'dc_type': '',
    'pdf_keywords': '',
    'pdf_pdfversion': '',
    'pdf_producer': '',
    'xmp_createDate': '',
    'xmp_creatorTool': '',
    'xmp_metadataDate': '',
    'xmp_modifyDate': '',
    'xmpmm_documentId': '',
    'xmpmm_instanceId': ''}
```

Figure 16: XMP Metadata Extracted with PyPDF2 from Black Hat Python.

This parsing technique, where the document information dictionary object that the file's creator intended to be extracted is the one that the parser extracts is fine for general use cases involving the classification and organization of files (e.g. PDF document library software). However, for digital forensics use cases, this selection of one metadata object to the exclusion of any others is an unacceptable omission. For example, in this bhp.pdf file the metadata elements that are omitted in the PyPDF2 extraction would certainly be important for a forensic investigation. The original document information dictionary object contains /Creator and /Producer elements, which identify the technical means used to create the PDF document itself. Additionally, the /Keyword and /Title elements in the original object differ significantly from the modified object—which can also be important information for an investigation. Outside of this particular example, it should become clear that there can be any number of discrepancies between the terminal object, and its

Figure 17: XMP Metadata References and Objects in Black Hat Python PDF.

preceding counterparts. And, in situations where an investigator suspects that the document's creator may have tried to conceal his identity, this historical chain of metadata objects becomes a critical priority.

4.2. Reporting False Negatives

Moving on to the PyPDF2-based get_xmp_info() function that is used to extract the XMP metadata from the *Black Hat Python* PDF document, we see that it located no

Figure 18: Preliminary Code for Low-Level Carving.

metadata elements (see Figure 16). However, when the file's binary is interrogated manually in Vim, then an XMP metadata reference-object pair is located (see Figure 17). The absence of the XMP object in the extract yielded by `PyPDF2` is explained in part by the warning that is thrown when the `get_xmp_info()` function is used on the `bhp.pdf` file:

```
PdfReadWarning: Xref table not zero-indexed. ID numbers for
objects will be corrected. [pdf.py:1736]
```

According to the comments in the `PyPDF2` source code, this warning is generated "if [the] table [is] not zero indexed, [which] could be due to [an] error from when [the] PDF was created [...] which will lead to mismatched indices later on" (Stamy 2015). The Cross-Reference table (i.e. `Xref`) is an object within the PDF file that theoretically enables applications to access any indirect object in the file without having to first read and parse the entire document (Adobe Systems Incorporated, 2001, p. 64). In the case of this file, however, the `/Metadata 3 0 R` reference which maps to the `3 0 obj` object is not mapped in the `Xref` table—which makes it functionally invisible to the `PyPDF2` module's parsing methodology. Again, while this may not be a deal-breaking issue for non-security related use-cases, this is highly problematic for anyone thinking to use the `PyPDF2` module for digital forensics work. As we saw with the prior examinations of the metadata streams (see Sections 2.5 and 3.2), these objects are incredibly rich in metadata, all of which is potentially critical for a forensic investigation. Moreover, this issue with the `PyPDF2` module being unable to handle metadata objects that are not properly cross referenced in the `Xref` table is inclusive of document information dictionary objects as well.

```
In [71]: def get_info_ref(file_path):
             with open(file_path, 'rb') as raw_file:
                 read_file = raw_file.read()
                 regex = b'[/]Info[\s0-9]*?R'
                 pattern = re.compile(regex, re.DOTALL)
                 info_ref = re.findall(pattern, read_file)
                 info_ref = de_dupe_list(info_ref)
                 if len(info_ref) == 0:
                     info_ref_exists = False
                 else:
                     info_ref_exists = True
                 return (info_ref_exists, info_ref)
```

Figure 19: Function to Extract Any Document Information Dictionary References.

5. Low-Level Python Solution

Given these fundamental and critical issues with the PyPDF2 module's ability to reliably extract all of a document's metadata, a different method is required to solve this problem with Python. In contrast with the module-based, high-level approach explored in the prior section, the proposed solution requires no modules outside of the standard library, and approaches the problem from a much lower level, carving the PDF document's binary structure directly with regular expressions (regex). Both the techniques to extract document information dictionary and XMP metadata require only two preliminary steps. The first is to import Python's regex module, and the second is to define a simple custom function, de_dupe_list(), that take a list as input and returns a new version of the list with any duplicated elements removed (see Figure 18).

5.1. Document Information Dictionary

Once the regex library has been imported and the de_dupe_list() function created, the next step will be to create a new custom function, get_info_ref(), that will replicate the manual search process, to locate any document information dictionary references, regardless of whether or not they are cross-referenced in the Xref table (see Figure 19 for the full function). This function begins by opening and reading the PDF file as a binary stream object. A regex pattern is then defined that will locate any binary strings

```
In [72]: print(get_info_ref('bhp.pdf'))

         (True, [b'/Info 610 0 R', b'/Info 1969 0 R'])
```

Figure 20: Printed Tuple Containing List of Document Information Dictionary References.

```
In [73]: def get_info_obj(file_path):
             with open(file_path, 'rb') as raw_file:
                 read_file = raw_file.read()
                 info_ref_tuple = get_info_ref(file_path)
                 info_obj_dict = {}
                 for ref in info_ref_tuple[1]:
                     info_ref = ref.decode()
                     info_ref = info_ref.replace('/Info ', '') \
                                        .replace(' R', '')
                     info_ref = str.encode(info_ref)
                     regex = b'[^0-9]' + info_ref + b'[ ]obj.*?endobj'
                     pattern = re.compile(regex, re.DOTALL)
                     info_obj = re.findall(pattern, read_file)
                     info_obj = de_dupe_list(info_obj)
                     if len(info_obj) > 0:
                         for obj in info_obj:
                             info_obj_dict[ref] = obj
                 if len(info_obj_dict) == 0:
                     info_obj_exists = False
                 else:
                     info_obj_exists = True
                 return (info_obj_exists, info_obj_dict)
```

Figure 21: Function to Extract Any Document Information Dictionary Objects.

which (1) begin with the `/Info` keyword, (2) contain an object and generation number, and (3) terminate with the `R` keyword. This regex pattern is compiled—using the `re.DOTALL` option to ensure that multi-line binary strings are captured—using the `re.compile()` method from the re module, and the compiled version is then passed through the module's `re.findall()` method to locate any instances of the pattern in the PDF's binary stream. The collection of located references is passed through the `de_dupe_list()` function to remove any duplicate elements. Conditional logic is then applied to test whether or not any references were located. The function completes by returning a tuple composed of two elements: a Boolean value indicating whether or not it successfully located any references, and a list of any located references—with the reference

```
In [75]: obj = get_info_obj('bhp.pdf')[1]
         for i in obj:
             print(i.decode())
             print(obj[i].decode(), '\n')

         /Info 610 0 R

         610 0 obj
         <<
         /Author (Justin Seitz)
         /CreationDate (D:20141209014739+00'00')
         /Creator (calibre 2.11.0 [http://calibre-ebook.com])
         /Keywords (COMPUTERS / Security / General)
         /Producer (calibre 2.11.0 [http://calibre-ebook.com])
         /Title (Black Hat Python: Python Programming for Hackers and Pentes
         ters)
         >>
         endobj

         /Info 1969 0 R

         1969 0 obj
         <</ModDate (D:20141209024746Z)
         /Title (Black Hat Python)
         /Author (Justin Seitz)
         /Subject (Python Programming for Hackers and Pentesters)
         /Keywords (Python, Hacking, Security, Programming, Pentesting)
         >>
         endobj
```

Figure 22: Printed Version of the PDF's Document Information Dictionary References and Objects.

being given as a binary string (see Figure 20). The Boolean value is added to the return value so that tests can easily be run before trying to iterate over a non-existent list.

Utilizing this custom `get_info_ref()` function, the next step is to create another custom function, `get_info_obj()`, that will take the indirect references located in the prior function and replicate the manual process, to locate each reference's respective object (see Figure 21 for the full function). The function begins by reading in the target PDF file as before. Then, the `get_info_ref()` function is called, to store the indirect references for any document information dictionaries as a variable, and a new dictionary variable is created to store any located object matching these references. The function then begins a for loop, that iterates over each reference located by the `get_info_ref()` function. Each reference value is passed through a `decode()` method, and then the `/Info` and `R` keyword

```
In [80]: obj = get_xmp_obj('bhp.pdf')[1]
         for i in obj:
             print(i.decode())
             print(obj[i].decode(), '\n')

         /Metadata 3 0 R

         3 0 obj
         << /Subtype /XML /Length 5888 /Type /Metadata /DL 5888 >>
         stream
         <?xpacket begin="" id="W5M0MpCehiHzreSzNTczkc9d"?>
         <x:xmpmeta xmlns:x="adobe:ns:meta/">
           <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns
         #">
             <rdf:Description xmlns:dc="http://purl.org/dc/elements/1.1/"
         rdf:about="">
               <dc:description>
                 <rdf:Alt>
                   <rdf:li xml:lang="x-default">&lt;p&gt;&lt;i&gt;Black Ha
         t Python&lt;/i&gt; explores the darker side of Python's capabilit
         ies, helping you test your systems and improve your security post
         ure.&lt;/p&gt;</rdf:li>
                 </rdf:Alt>
               </dc:description>
```

Figure 23: Printed Version of the PDF's XMP Metadata References and Objects.

substrings are stripped, leaving only the reference's object and generation number. This substring is then passed through a binary encoding method, and used to form the centerpiece of a regex pattern that matches binary strings containing the reference number, followed by the `obj` keyword, and terminating with the first occurrence of the `endobj` keyword. As with the prior function the regex pattern is passed through a `compile()` method, which is subsequently passed through the `findall()` method, and is finally passed through the previously defined `de_dupe_list()` function—generating a list containing any document information dictionary objects from the PDF file. Each of these list elements is paired with its matching reference, the pair of which are added to the dictionary created previously. Conditional logic is also used to create a Boolean value based on the presence of any such pairs in the dictionary. And finally, the Boolean value and the dictionary are returned by the function. By decoding each of the binary strings constituting the dictionary's key-value pairs in the tuple returned by passing the `bhp.pdf` file through this function, we can see that the `get_info_obj()` function located both of

the dictionary information dictionary objects that were seen in the manual interrogation of the PDF earlier in Vim (see Figure 22).

5.2. XMP Metadata

As we saw with the manual investigations, the general methodology for locating objects based on indirect references is the same regardless of the reference's keyword. Therefore, adapting the previously defined `get_info_ref()` and `get_info_obj()` functions to locate the respective XMP metadata references and object can easily be done by replacing the `/Info` keyword in each function with a `/Metadata` keyword. This results in a `get_xmp_ref()` function that will locate any indirect references to XMP metadata, as well as a `get_xmp_object()` function that will locate any matching XMP metadata objects in the input PDF file, regardless of the presence or integrity of an `Xref` table in the file. As the only differences between these two sets of functions are a single keyword, it would be redundant to include the XMP metadata functions in their entirety. However, see Figure 23 for an example of the XMP metadata references and objects contained in the `bhp.pdf` file being printed to the console.

6. Putting It All Together: The PDF Metadata Utility

While the explorations of the *Black Hat Python* PDF in the prior sections served to demonstrate the clear superiority of low-level binary parsing methods over high-level `PyPDF` library approaches for interrogating PDF files from a digital forensics perspective with Python, the method of demonstration (i.e. running individual functions through Jupyter Notebook) is less than ideal. To resolve these problems, I reworked the exploratory functions described above into a standalone command-line utility: `pdf-metadata` (Plaisance, 2018a). While a full examination of the utility's source code is outside the scope of this present paper, the following provides a high-level overview of its code structure and functionality. For reference, the utility's full user guide can be found in the `README.md` file within the repository (Plaisance, 2018b), and the source code in the collocated `pdf-metadata.py` file (Plaisance, 2018c).

6.1. Structural Elements

As opposed to using separate functions to parse through the PDF file's binary and extract metadata, the `pdf-metadata.py` utility makes use of an object-oriented programming framework, where the functions described in the sections above are transformed into methods within a new `BinaryPdfForensics` class. This class's namespace contains four attributes:

Attribute Name	Attribute Description
`file_path`	The path (full or truncated) of a given PDF file.
`temp_path`	The hard-coded truncated path of the temporary PDF file used for decryption and metadata extraction.
`output_path`	The path (full or truncated) of the output file.
`password`	The password used to decrypt an encrypted PDF file. This value defaults to `None` if no input is provided.

The class's essential functionality is defined by its thirteen methods:

Method Name	Method Description
`__init__`	This method initiates an instance of the class object with the provided attributes.
`pdf_magic`	This method reads the input file as a binary stream, and interrogates the first four bytes to determine if they decode to the PDF magic number (i.e. `%PDF`). This is used to determine whether or not a given file, regardless of its extension, is a PDF. It returns a tuple containing two values: 1. A Boolean value identifying whether or not the input path is a PDF file 2. A string description of the magic assessment. For PDF files, this string value will contain the version of the PDF (e.g. `%PDF-1.4`)
`get_crypt_ref`	This method reads the input file as a binary stream, and then performs a regex search to determine if the trailer object in the PDF contains a reference to an encryption object (e.g. `/Encrypt 15 R`). It returns a tuple containing two values: 1. A Boolean value identifying whether or not the file contains an `/Encrypt` reference

	2. A list of any located /Encrypt references
copy_file	This method copies an un-encrypted PDF file to a new temporary directory for metadata extraction. It begins by removing any existing temporary directories or files, and creates a new temporary directory to copy the PDF file under examination. Once created, a copy function is executed to create a copy of the file in the directory. It returns a Boolean value based on the success of the copy function.
decrypt_file	This method decrypts encrypted PDF files with the given password attribute. It begins by testing the password value. If no password is provided, then the default password is stored. A temporary directory is created to store the decrypted version of the PDF file, and any previously existing temporary directories and files are removed. The qpdf Linux command[3] is then invoked to decrypt the file, saving the decrypted file in the temporary directory. If the qpdf command fails, then the original file is copied into the temporary directory for use with subsequent methods. It returns a tuple with four elements: 1. The return code generated by the qpdf command 2. The password used in the command to decrypt the file 3. Any output (i.e. stdout) generated by the command 4. Any error messages (i.e. stderr) generated by the command
temp_clean	This method is used to remove any temporary directories that were created by the script. It returns a Boolean value based on the success or failure of the cleaning operation.
get_info_ref	This method reads the input file as a binary stream, and then performs a regex search, to determine if it contains an /Info reference. It returns a tuple containing two values: 1. A Boolean value identifying whether or not the PDF contains the /Info reference 2. A list of any located binary /Info references
get_xmp_ref	This method reads the input file as a binary stream, and then performs a regex search, to determine if it contains a /Metadata reference. It returns a tuple containing two values: 1. A Boolean value identifying whether or not the PDF contains the /Metadata reference

---

[3] For more information on qpdf, see the utility's source code and documentation on the Github repository (Berkenbilt, 2019).

	2. A list of any located binary /Metadata references
`get_info_obj`	This method reads the input file as a binary stream, and then calls the `get_info_ref` method to get any /Info references in the file. Any located /Info references are then used to locate any matching /Info objects in the file. It returns a tuple containing the following elements: 1. A Boolean value identifying whether or not an /Info object was found in the file 2. A dictionary which maps the /Info references with the corresponding string decoded /Info objects
`get_xmp_obj`	This method reads the input file as a binary stream, and then calls the `get_xmp_ref` method to get any /Info references in the file. Any located /Metadata references are then used to locate any matching /Metadata objects in the file. It returns a tuple containing the following elements: 1. A Boolean value identifying whether or not an /Metadata object was found in the file 2. A dictionary which maps the /Metadata references with the corresponding string decoded /Metadata objects
`file_stats`	This method calculates the statistics which constitute the PDF's file system metadata. It uses Python's built-in `os.stat` method from the standard library to retrieve these file system metadata. It returns a list containing five string elements: 1. The file's absolute path 2. Its human readable size 3. The most recent access time 4. The most recent modification time 5. The most recent change time
`file_hashes`	This method reads the input file as a binary stream, and then calculates file hash digests of the file for each of the hashing algorithms supported by Python's built-in `hashlib` module. It returns a list of string values for the respective digests of the file's hash for each algorithm: 1. MD5 2. SHA1 3. SHA224 4. SHA256

	5. SHA384
	6. SHA512
`gen_report`	This method generates an HTML formatted report, detailing the file system, document information dictionary, and XMP metadata contained within the inspected PDF file. It functions by invoking all of the previously defined methods within the class, capturing their return values as variables, and inserting these variables into an HTML structure, parts of which are defined as global constants, and other parts of which are variably defined based on factors such as the file's encryption status, the success or failure of decryption efforts, and the presence and numbers of any `/Info` or `/Metadata` references and objects in the PDF file.

Outside of the `BinaryPdfForensics` class, the bulk of the program's logic is contained within the `arg_parser` function that uses Python's built-in `argparse` module to accept, parse, and interpret a variety of command-line arguments for the main program. There are, additionally, a variety of helper functions which control aspects of the utility like input validation, list de-duplication, etc. These helper functions are not particularly pertinent to this paper's focus, and do not require description here. However, they are fully documented in the application source code. At its core, the `BinaryPdfForensics` class's attendant methods, and the helper functions serve to ingest a given PDF file, and conduct the same kinds of low-level metadata extraction detailed in-depth in the above sections. The extracted metadata are then collected into an HTML formatted report which contains all of the PDF file's application and file system metadata.

6.2. User Interface

By means of the `arg_parser` function described above, the utility has an argument based command-line user interface (UI). In accord with the eighth tenet of the Unix philosophy (Gancarz, 2003, pp. 91-101), this UI avoids the perils of being a *captive* user interface, and in adherence with the ninth tenet (Ibid., pp. 102-106) functions as a filter, converting data from one form into another via an argumented command.

6.2.1. Input Modes

The UI has two input modes, which are respectively defined by the following subcommands:

Subcommand	Description
`single`	This subcommand engages the *single* input mode of the utility, which enables it to ingest a single PDF file.
`batch`	This subcommand engages the *batch* input mode of the utility, which enables it to handle a CSV containing any number of properly structured input parameters.

The mode subcommand is *non-optional*; there is no default parameter. This is due to the fact that in the `arg_parser` function, the subcommands are initialized as required subparsers:

```
subparsers = parser.add_subparsers(
    title = 'modes',
    dest = 'mode',
    help = 'input mode options',
    )
subparsers.required = True
```

Following this definition, each of the respective modes is added to the now required `subparsers` object using the `add_parser` and `add_argument` methods.

6.2.2. General Arguments

There are two optional arguments for the utility that can be used without error irrespective of the chosen mode:

Argument	Description
`-h, --help`	This argument will print the help documentation to the console.
`-q, --quiet`	This argument suppresses the utility's title information from being printed to the console.

The `help` argument can be used in absence of any mode subcommand, and if used in concert with any subcommands or arguments, will override them. The `quiet` argument does require a subcommand and its respective required arguments in order to function.

6.2.3. Single Mode Arguments

There are three specific arguments within the single subcommand mode, two of which are required with the remaining argument being optional:

Argument	Description
`-i, --input`	This argument defines the path of the PDF that will be analyzed for metadata extraction. It can be given either as a relative or absolute path. This argument is *required*.
`-o, --output`	This argument indicates that output will be written to an external file. If followed by a value, it will be interpreted as the name of a new file which the utility will create (provided that there is a validly structured PDF file passed to the `input` argument). If no value is provided, the default output value, `output.html`, will be used for the new file. As with the input, either relative or absolute paths can be used to define this element. This argument is also *required*.
`-p, --password`	This argument defines a password that will be used in the decryption attempts of the PDF file input into the utility via the `input` argument. If no value is provided, the default value of a null string will be passed. Use of this option requires the prior installation of the `qpdf` utility in order to decrypt the encrypted files. This is an *optional* argument.

If no valid input parameters are provided, but single mode is indicated, the generic error splash screen with the usage instructions is provided to the user.

6.2.4. Batch Mode Arguments and Input Parameters

There is only one specific argument within the `batch` subcommand mode, which is *required*:

Argument	Description
`-f, --file`	This argument defines the CSV file that will be parsed for input parameters. It can be given either as a relative or absolute path.
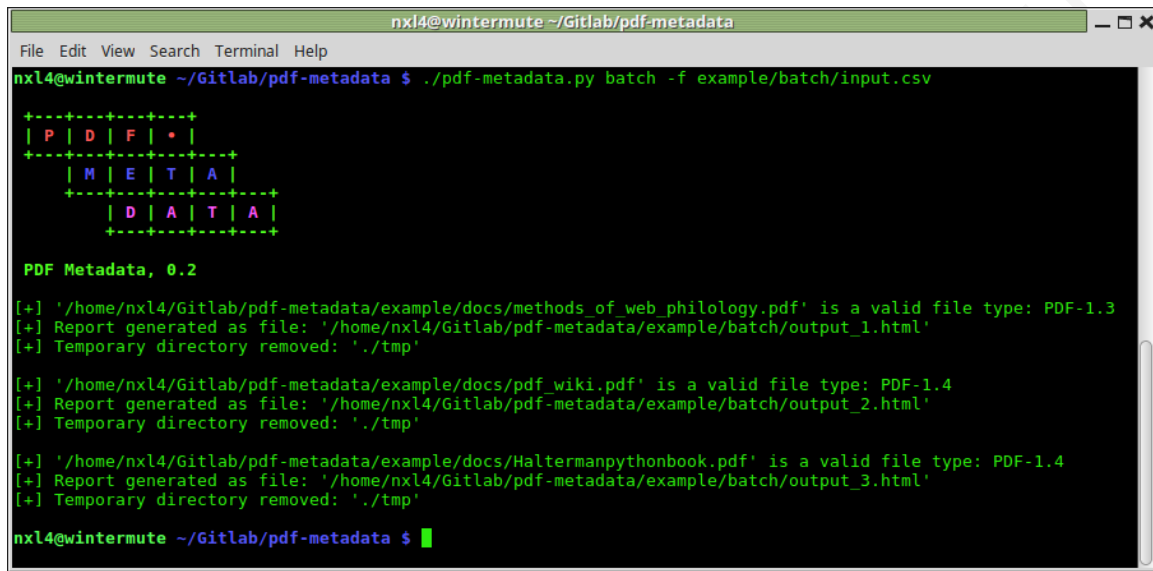
Figure 24: An example of single mode.

The CSV file that is passed as the `file` parameter in `batch` mode must adhere to the following structure in order to be properly interpreted by the utility's CSV parser and not result in an error message being thrown. First, each row in the CSV file must have *either* two *or* three columns. Any input files with rows containing column counts outside of these parameters will be unreadable by the utility. Second, the first column of the CSV file must contain the paths of the PDF files to be read by the utility. As with the `single` mode, these paths can be given either as relative or absolute. Third, the second column must contain the specific paths used to create the output file for each input PDF. These output files must be distinct, or output results may be overwritten when new PDF files are analyzed. Again, these paths can be given as relative or absolute values. Finally, the third column may optionally contain password values to be used in the decryption of the specific PDF file given in the first column of that row. As an optional value, this can be present or absent on a per row basis (i.e. row one may have a password value given in column three, but row two may not; this would be a validly structured input file).

Figure 25: An example of batch mode.

6.3. Usage

As a Linux utility, pdf-metadata can be run with the following general argument structure:

```
pdf-metadata.py [-h] [-q] {single, batch} ...
```

The general command-line structure for `single` mode is:

```
pdf-metadata.py single [-h] [-q] -i INPUT_NAME -o
OUTPUT_NAME [-p PASSWORD]
```

And, the general command-line structure for `batch` mode is:

```
pdf-metadata.py batch [-h] [-q] -f INPUT_FILE
```

For illustrations of the two modes in action please refer to Figures 24 and 25 respectively, for examples of `single` and `batch` modes processes.

6.4. Output

The output provided by the `pdf-metadata` utility for each analyzed PDF file is an HTML formatted report, which contains the following contents:

- File System Metadata
  - File Statistics
    - Absolute Path
    - Human Readable Size
    - Most Recent Access Timestamp
    - Most Recent Modification Timestamp
    - Most Recent Change Timestamp
  - File Hashes
    - MD5
    - SHA1
    - SHA224
    - SHA256
    - SHA384
    - SHA512
- Application Metadata
  - PDF Version
  - Encryption Status
  - Document Information Dictionary
    - References
    - Objects
  - XMP Metadata
    - References
    - Objects

For an illustration of the respective sections of the report generated for the *Black Hat Python* PDF examined throughout the course of this paper, please refer to Figures 26 through 31 in the Appendix.

7. Conclusion

In conclusion, this paper has served to demonstrate that the `PyPDF` and `PyPDF2` modules in Python are ill-equipped for use in digital forensics investigative contexts, owing to the fact that metadata extraction attempts using these modules can often result in a critically incomplete reporting of a PDF file's metadata. However, it was further demonstrated that a complete and accurate reporting of such can be obtained through manual investigation, and that the manual technique and processes can be successfully replicated in Python, resulting in a low-level technique operating directly on the PDF's binary structure that results in a complete extraction of any document information dictionary and XMP metadata references and objects contained therein. However, while the functions detailed herein do succeed in resolving this problem, the functions alone do not constitute a complete framework for reliably extracting PDF metadata for digital forensics investigations, the `pdf-metadata` application does.

Appendix

PDF Metadata Report

Generated by PDF Metadata, Version 0.1

Contents

- File System Metadata
  - File Statistics
  - File Hashes
- Application Metadata
  - PDF Version
  - Encryption Status
  - Document Information Dictionary
  - XMP Metadata
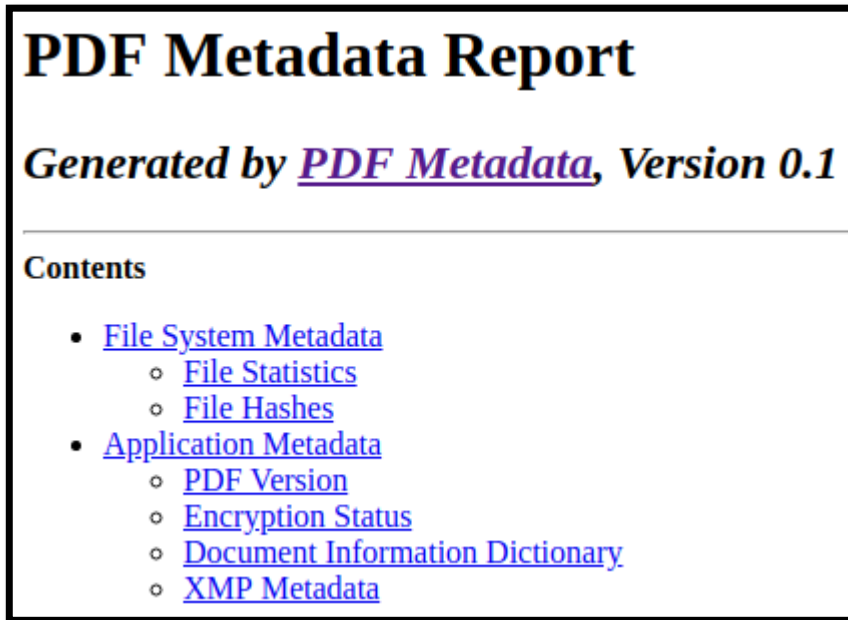
Figure 26: Report title and table of contents.

File System Metadata

File Statistics

File Path:

`/home/nxl4/Gitlab/pdf-work/bhp.pdf`

File Size:

`3.167049 MB (3167049 bytes)`

Time of Most Recent Access:

`2018-12-01 17:28:04`

Time of Most Recent Content Change:

`2018-08-11 15:05:33`

Time of Most Recent Metadata Change:

`2018-11-25 16:35:14`

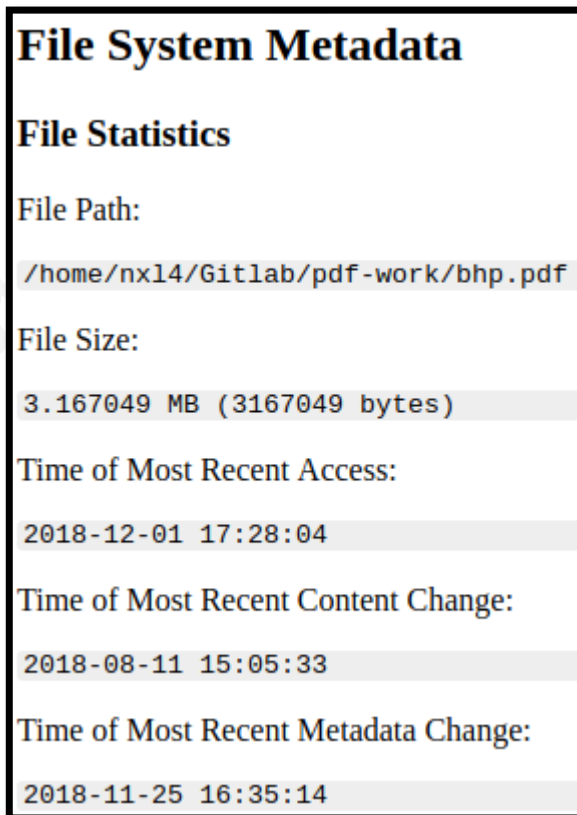*Figure 27: File statistics section of file system metadata.*

File Hashes

MD5 Hash:

b868a33824b303f128bf97f0fd1b5152

SHA1 Hash:

22457460bdd7627c4abbea00aae2fd45aa9d25e9

SHA224 Hash:

0e6ea548dc5b05babcd74d5aea39119e5d3e17a5216f9a5e800389d7

SHA256 Hash:

90b5fafc021656dc9335678e565bdc370d948b2de01098557c8ecd1859062f04

SHA384 Hash:

7ad105df4c82684b11bc717da7cacb1fdabff7bf552b67bec4cdf2bc667ad987e2d8e665b6e
ea9f51aa02f158d20b542

SHA512 Hash:

01522d29a4c7628612dceba1127a02db6f4a279eb39c63843db00954bd118e2e9d832692ad2
8336bfc01ef223ee72d34fc516b1ba46065e61d3a23732c10a40b

Figure 28: Computed file hashes section of file system metadata.

Application Metadata

PDF Version

The magic number identifies this PDF file as being version:

PDF-1.4

Encryption Status

This file is not encrypted.

Figure 29: PDF version and encryption status sections of application metadata.

Document Information Dictionary

There are a total of 2 document information directory references located in the PDF file:

```
/Info 610 0 R
```

```
/Info 1969 0 R
```

There are a total of 2 document information dictionary objects located in the PDF file:

```
610 0 obj
<<
/Author (Justin Seitz)
/CreationDate (D:20141209014739+00'00')
/Creator (calibre 2.11.0 [http://calibre-ebook.com])
/Keywords (COMPUTERS / Security / General)
/Producer (calibre 2.11.0 [http://calibre-ebook.com])
/Title (Black Hat Python: Python Programming for Hackers and Pentesters)
>>
endobj
```

```
1969 0 obj
<</ModDate (D:20141209024746Z)
/Title (Black Hat Python)
/Author (Justin Seitz)
/Subject (Python Programming for Hackers and Pentesters)
/Keywords (Python, Hacking, Security, Programming, Pentesting)
>>
endobj
```

Figure 30: Document Information Dictionary section of application metadata.

XMP Metadata

There are a total of 1 XMP metadata references located in the PDF file:

```
/Metadata 3 0 R
```

There are a total of 1 XMP metadata objects located in the PDF file:

```
3 0 obj
<< /Subtype /XML /Length 5888 /Type /Metadata /DL 5888 >>
stream
<?xpacket begin="ï»¿" id="W5M0MpCehiHzreSzNTczkc9d"?>
<x:xmpmeta xmlns:x="adobe:ns:meta/">
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description xmlns:dc="http://purl.org/dc/elements/1.1/"
rdf:about="">
      <dc:description>
        <rdf:Alt>
          <rdf:li xml:lang="x-default"><p><i>Black Hat Python</i> explores
the darker side of Pythonâ□s capabilities, helping you test your systems
and improve your security posture.</p></rdf:li>
        </rdf:Alt>
      </dc:description>
      <dc:title>
        <rdf:Alt>
          <rdf:li xml:lang="x-default">Black Hat Python: Python Programming
for Hackers and Pentesters</rdf:li>
        </rdf:Alt>
```

Figure 31: XMP Metadata section of application metadata.

Bibliography

Adobe Systems Incorporated (1999). *PostScript Language Reference* (3rd ed.). Reading: Addison-Wesley.

Adobe Systems Incorporated (2001). *PDF Reference: Adobe Portable Document Format, Version 1.4* (3rd ed.). Boston: Addison-Wesley.

Adobe Systems Incorporated (2008). *Document Management: Portable Document Format* (1st ed.). Retrieved 12 August 2018, from https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf.

Adobe Systems Incorporated (2012). *XMP Specification, Part 1: Data Model, Serialization, and Core Properties*. Retrieved 12 August 2018, from https://wwwimages2.adobe.com/content/dam/acom/en/devnet/xmp/pdfs/XMP%20SDK%20Release%20cc-2016-08/XMPSpecificationPart1.pdf.

Adobe Systems Incorporated (2014). *XMP Specification, Part 2: Additional Properties*. Retrieved 12 August 2018, from http://wwwimages.adobe.com/www.adobe.com/content/dam/acom/en/devnet/xmp/pdfs/XMP%20SDK%20Release%20cc-2014-12/XMPSpecificationPart2.pdf.

Adobe Systems Incorporated (2016). *XMP Specification, Part 3: Storage in Files*. Retrieved 12 August 2018, from https://wwwimages2.adobe.com/content/dam/acom/en/devnet/xmp/pdfs/XMP%20SDK%20Release%20cc-2016-08/XMPSpecificationPart3.pdf.

Berkenbilt, Jay (2019). *QPDF*. Retrieved 20 January 2019, from https://github.com/qpdf/qpdf.

Fenniak, Mathieu (2016a). *PyPDF2 Documentation*. Retrieved 26 November 2018, from https://pythonhosted.org/PyPDF2/index.html.

Fenniak, Mathieu (2016b). *The DocumentInformation Class*. Retrieved 26 November 2018, from https://pythonhosted.org/PyPDF2/DocumentInformation.html.

Fenniak, Mathieu (2016c). *The PdfFileReader Class*. Retrieved 26 November 2018, from
https://pythonhosted.org/PyPDF2/PdfFileReader.html.

Fenniak, Mathieu (2016d). *The XmpInformation Class*. Retrieved 26 November 2018,
from https://pythonhosted.org/PyPDF2/XmpInformation.html.

Gancarz, Mike (2003). *Linux and the Unix Philosophy*. Amsterdam: Digital Press.

Gill, Tony (2008). Metadata and the Web. In *Introduction to Metadata*, edited by Murtha
Baca, 20–37. 2nd edition. Los Angeles: The Getty Research Institute.

Gilliland, Anne J (2008). Setting the Stage. In *Introduction to Metadata*, edited by
Murtha Baca, 1–19. 2nd edition. Los Angeles: The Getty Research Institute.

International Organization for Standardization (2017). *ISO 32000-2:2017: Document
Management—Portable Document Format—Part 2: PDF 2.0*. Retrieved 26
November 2018, from https://www.iso.org/standard/63534.html.

O'Connor, T.J. (2010, April 1). *Grow Your Own Forensic Tools: A Taxonomy of Python
Libraries Helpful for Forensic Analysis* [White paper]. Retrieved 12 August 2018,
from SANS Institute: https://digital-
forensics.sans.org/community/papers/gcfa/grow-forensic-tools-taxonomy-python-
libraries-helpful-forensic-analysis_6879.

O'Connor, T.J. (2013). *Violent Python: A Cookbook for Hackers, Forensic Analysts,
Penetration Testers and Security Engineers*. Boston: Elsevier.

Pittman, Ryan D. and Dave Shaver (2010). Windows Forensic Analysis. In *Handbook of
Digital Forensics and Investigation*, edited by Eoghan Casey, 209–300. London:
Elsevier Academic Press.

Pesce, Larry (2008). *Document Metadata, the Silent Killer…* [White paper]. Retrieved 22
August 2018, from SANS Institute: https://www.sans.org/reading-
room/whitepapers/privacy/document-metadata-the-silent-killer--32974.

Plaisance, Christopher (2016). Methods of Web Philology: Computer Metadata and Web
Archiving in the Primary Source Documents of Contemporary Esotericism.

International Journal for the Study of New Religions, 7(1), 43–68.
doi:10.1558/ijsnr.v7i1.26074.

Plaisance, Christopher (2018a). *pdf-metadata*. Retrieved 30 November 2018 from
https://gitlab.com/nxl4/pdf-metadata/.

Plaisance, Christopher (2018b). *pdf-metadata/README.md*. Retrieved 20 January 2019
from https://gitlab.com/nxl4/pdf-metadata/blob/master/README.md.

Plaisance, Christopher (2018c). *pdf-metadata/pdf-metadata.py*. Retrieved 20 January
2010 from https://gitlab.com/nxl4/pdf-metadata/blob/master/pdf-metadata.py.

Sammons, John (2012). *The Basics of Digital Forensics: The Primer for Getting Started
in Digital Forensics*. Waltham: Elsevier.

Seitz, Justin (2014). *Black Hat Python: Python Programming for Hackers and
Pentesters*. San Francisco: No Starch Press.

Stamy, Matthew (2015). *PyPDF2/pdf.py*. Retrieved 30 November 2018, from
https://github.com/mstamy2/PyPDF2/blob/master/PyPDF2/pdf.py.