CS7038 - Malware Analysis - Wk09.1

Analysis of PDF and Office Documents

Coleman Kane
kaneca@mail.uc.edu

March 20, 2018

PDF Document Overview

We looked at PDF documents briefly during Week 2.

PDF documents are intended to be a print-representation (or "final copy") of a document prepared for digital viewing. However, the goal is that the document displayed on-screen is a 100% exact approximation of the visual document you will see if printed. It has roots in the earlier-developed *PostScript* language, but isn't a fully-compatible reimplementation.

Some interesting features in (most) PDF readers:

- JavaScript (PDFjs, ECMA) interpreter
- Forms UI support (XFA, FDF, XFDF)
- U3D/PRC 3d-model embedded support
- Inline HTML
- Numerous embedded image formats
- PDF-within-PDF
- Encoded/encrypted stream data

PDF Document Structure

PDF documents more or less follows the below structure:

%PDF-N.N	... header data ...	... unused ...
X Y obj	object data	endobj
W Z obj	object data	endobj
...	more object data	...
xref	... xref table ...	... unused ...
trailer	... trailer data ...	startxref NNNN
%%EOF		

Each entity inside of the document is located within one of the indirect objects identified above with the "X Y obj", "Z W obj", etc... declarations.

One of these objects is traditionally the “catalog”, or “root object”.

The *xref table* contains an index of the offsets for each of the indirect objects, from beginning of file.

The *trailer* contains a pointer to the *xref table* as well as a dictionary that defines the catalog, the count of objects in the cross-reference table, and other information that may be specific to the viewer.

PDF Objects

Object data is defined by beginning with the following text (where X and Y are integers):
`X Y obj`

The PDF specification defines a number of data types:

- Boolean values (representing True or False)
- Numbers
- Strings, enclosed with parentheses: `(this is a string)`
- Names, character data beginning with a slash: `/NameVal1`
- Arrays, ordered data enclosed with square brackets:
  `[(Object) (Data) (in) (a) (list)]`
- Dictionaries, name-indexed data, defined with << and >>:
  `<</Val1(This is a string) /Val2[(List) (data)]>>`
- Streams, large blobs of arbitrary data, embedded between `stream` and `endstream` keywords
- Null content

PDF Parser

The `pdf-parser.py` tool in Remnux can be helpful in navigating the PDF document structure.

- Search for data in object: `pdf-parser.py -s mytext file.pdf`
- Search for data in stream: `pdf-parser.py -searchstream=mytext file.pdf`
- List objects and their hashes: `pdf-parser.py -H file.pdf`
- Extract object: `pdf-parser.py -o 1 -d stream.dat file.pdf`
- Extract filtered object: `pdf-parser.py -f -o 1 -d stream.dat file.pdf`
- Parse, extract malformed: `pdf-parser.py -v -x malformed.dat file.pdf`
- Integrate with *yara*: `pdf-parser.py -y, -yarastrings`
- Python code generation: `pdf-parser.py -g example.pdf > example.py`

Office Software Overview

In modern office environments, a common practice is to use a number of file formats for interchange of work-related, editable content. The most common pattern is a software suite providing Spreadsheet, Slide Presentation, and Document creation.

The most common suite, by far, is **Microsoft Office**. However, there are others:

- Apache OpenOffice
- LibreOffice
- KOffice
- Apple iWork
- Hancom Office (Korean, Hangul)
- Ichitaro (japanese)

We will focus our efforts on the Microsoft suite of software, though it is notable that the space is diverse, and any one of these can be its own intrusion vector.

Microsoft Office File Formats

Generally, there are two data file formats that are of interest to MS Office document malware analysis:

- Office Open XML (OOXML) Files - Basically PKZIP archives with a specially-defined layout. Most office documents since about 2007 are dsitributed using this format (XLSX, DOCX, PPTX, etc.)
- Compound File Binary (CFB) - A binary file specification defined by Microsoft. Older Microsoft Office documents were built up from this format (DOC, XLS, PPT). Since 2007, it is still frequently used to embed Microsoft-specific binary data structures within documents and applications.

Latest CFB file specification:
`https://msdn.microsoft.com/en-us/library/dd942138.aspx`

Latest Office Open XML specifications (ISO/IEC 29500-1:2016, 29500-2:2012, 29500-3:2015, 29500-4:2008):
`https://www.iso.org/committee/45374/x/catalogue/`

The CFB file Format

We will begin with the CFB file format, which is useful in attacking Microsoft-specific vulnerabilities.

The CFB file format is a chunked data format, where the file is divided into **sectors**, and then there exist **file allocation tables** that each define an array of pointers to other file locations that map blocks in the file to their ordering within a data stream.

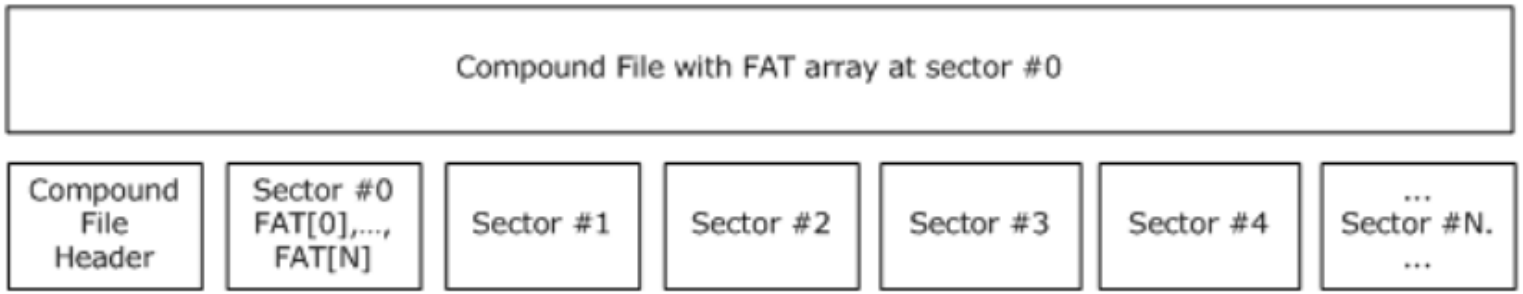
In contrast to the data from a PDF file, this organizational model creates a file structure where whole data streams (such as images, sub-documents, videos, content, embedded fonts, macros, etc...) are not guaranteed to exist contiguous within the file.

There exist a number of utilities that are useful for naviagting this structure:

- `https://www.decalage.info/python/oletools`
- `https://github.com/unixfreak0037/officeparser`
- `https://poi.apache.org/` - Java API for Office Documents

CFB Sectors

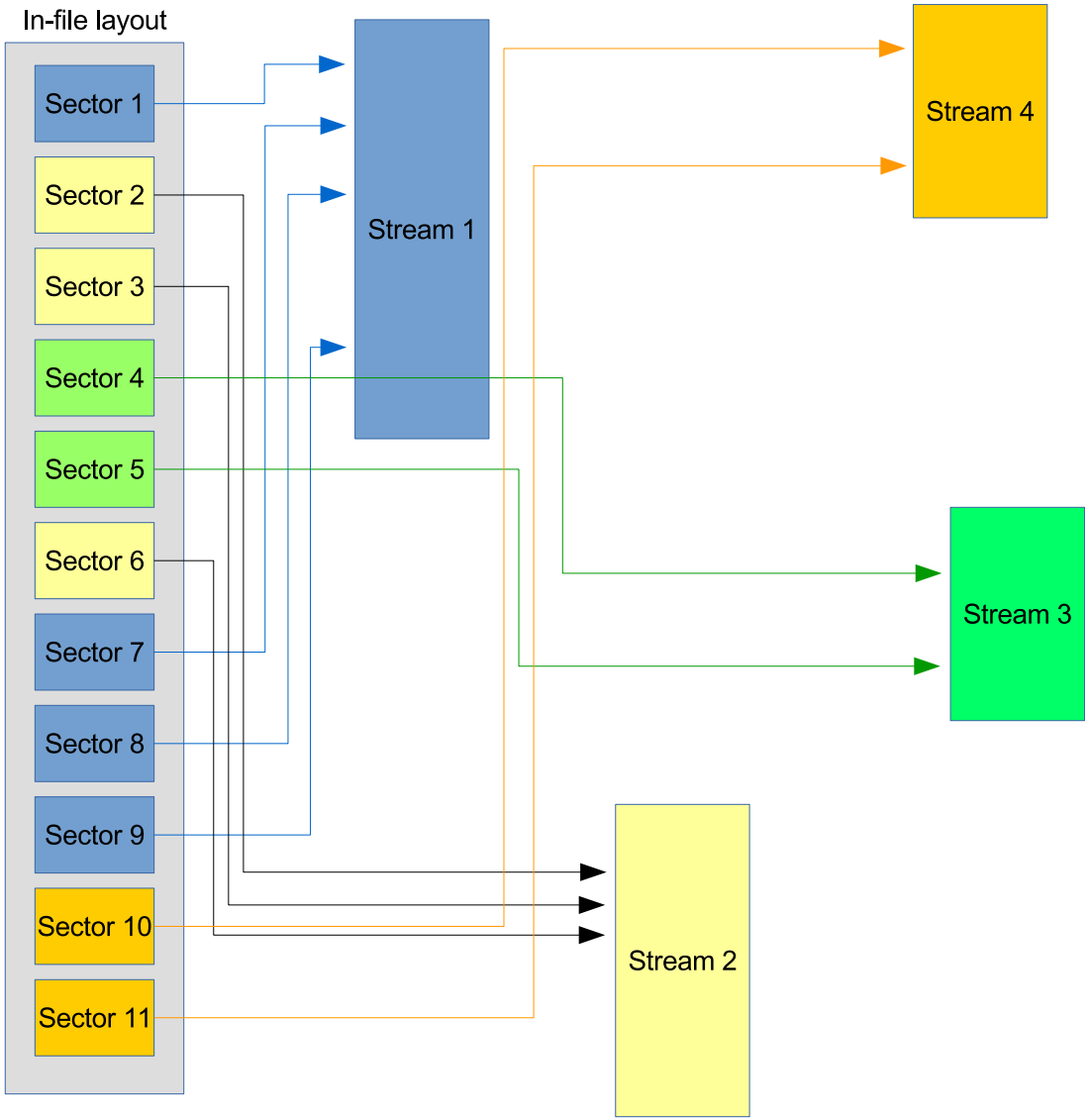
Each CFB file is divided into sectors. The first sector of the file contains the CFB header, which is where all of the information defining the top-level file layout.



This informs the user on how the rest of the file is organized, including the sector size, as well as where to find the document directory, file allocation tables, etc.

Almost exclusively, sector sizes are defined to be 512 bytes (0x200 hex), which is consistent with most common OS filesystems as well.

CFB Streams Layout



Sector layout to stream mapping

oletools helpers

All of the following tools are open-source, and have great documentation on the following site: `https://www.decalage.info/python/oletools`

- `olebrowse`: A GUI browser enabling you to navigate, view and extract streams. Very basic.
- `oledir`: Dump the stream directory of the document
- `olemap`: Dump the sector mappings (allocation) of a file
- `olemeta`: Dump metadata about the document
- `olevba`: Dump VBA macros from files

OOXML Layout

OOXML files can be analyzed with a simple `unzip` program. Some of the contents may need other tools to further analyze (like the JPEG in this one):

```
Archive:  test-doc.docx
    testing: _rels/.rels                        OK
    testing: word/document.xml                  OK
    testing: word/styles.xml                    OK
    testing: word/_rels/document.xml.rels   OK
    testing: word/settings.xml                  OK
    testing: word/media/image1.jpeg             OK
    testing: word/fontTable.xml                 OK
    testing: docProps/app.xml                   OK
    testing: docProps/core.xml                  OK
    testing: [Content_Types].xml                OK
```

Office Macros

Microsoft Office supports executable scripts embedded within documents. A common language used for this is Visual Basic for Applications (VBA). Similar to PDFjs that we discussed earlier, this langauge is a derivative of Visual Basic that has special hooks into the Office environment and the current (and linked) documents.

An example macro is available here: `https://msdn.microsoft.com/en-us/library/office/aa173542(v=office.11).aspx`

Frequently these will be used to execute arbitrary code, without relying upon exploits that intend to break parsing of the document. Some examples:

- `http://blog.fortinet.com/2017/03/08/microsoft-excel-files-increasingly-used-to-spread-malware`
- `https://blogs.sophos.com/2015/09/28/why-word-malware-is-basic/`
- `http://www.kahusecurity.com/2015/malicious-word-macro-caught-using-sneaky-trick/`