Dig Into the Attack Surface of PDF and Gain 100+ CVEs in 1 Year

March 2017

Ke Liu (@klotxl)

Tencent's Xuanwu Lab

Abstract

Portable Document Format (a.k.a. PDF) is one of the most widely used file formats in the world, this complex file format also exposes a large potential attack surface which is important for us to understand. During last year, by digging into the attack surface of PDF deeply and fuzzing the popular PDF readers efficiently, I discovered nearly 150 vulnerabilities independently in the world's most popular PDF readers including Adobe Acrobat and Reader, Foxit Reader, Google Chrome, Windows PDF Library, OS X Preview, and Adobe Digital Editions. More than 100 of the vulnerabilities have been fixed by vendors and assigned with CVEs.

The following section summarizes the outline of this paper:

- Introduction
- Attack Surface
- Test cases
- Fuzzing Tricks
- Results
- References

1. Introduction

PDF has become a de facto global standard for more secure and dependable information exchange since Adobe published the complete PDF specification in 1993 [1]. However, this complex file format also exposes a large potential attack surface which is important for us to understand. There are some researches about PDF security already, some of them are listed as follows:

- Nicolas Grégoire: Dumb fuzzing XSLT engines in a smart way [2]
- Zero Day Initiative: Abusing Adobe Reader's JavaScript APIs [3]
- Sebastian Apelt: Abusing the Reader's embedded XFA engine for reliable Exploitation [4]

However, each of these researches only covers a single attack surface of PDF. What's more, the basic but most frequently used PDF characteristics, such as images and fonts, are never mentioned. Since PDF is a complex format, a lot of work can be done here. By digging into the attack surface of PDF, I discovered nearly 150 vulnerabilities in various PDF readers.

My work mainly focus on Adobe Acrobat and Reader and most of the vulnerabilities were found in the products. However, this does not mean that the products are more vulnerable than other PDF readers.

2. Attack Surface

If you want to know what the attack surface is, the question you have to ask yourself first is how to find the attack surface. Here I summarized four possible methods to find the attack surface.

2.1 The Standard Documents

The ISO standard for PDF is ISO 32000-1:2008 [5] and the copy of the document can be downloaded freely from Adobe's web site [1]. This document has 756 pages and almost describes everything of the Portable Document

Format itself. However, some features are not described in detail in this document, such as JavaScript, XFA (XML Forms Architecture), FormCalc, etc. The following documents are some useful materials for reference.

- JavaScript for Acrobat API Reference [6]
- XML Forms Architecture (XFA) Specification [7]
- FormCalc User Reference [8]

Also, PDF supports embedding external files such as fonts (TrueType, Type0, Type1, Type3, etc.), Images (Jpeg2000, Jpeg, Png, Bmp, Tiff, Gif, Jbig2, etc.), XMLs (XSLT, etc.). The standard documents for these files will not be discussed in this paper.

2.2 Security Advisories

Keep an eye on security advisories is a good way to observe the status of security trends. Most importantly, we can know the flaw exists within which component. The following advisories are the ones that worth reading.

- Zero Day Initiative's Advisory [9]
- Chromium Issue Tracker [10]
- Adobe Security Bulletins and Advisories [11]

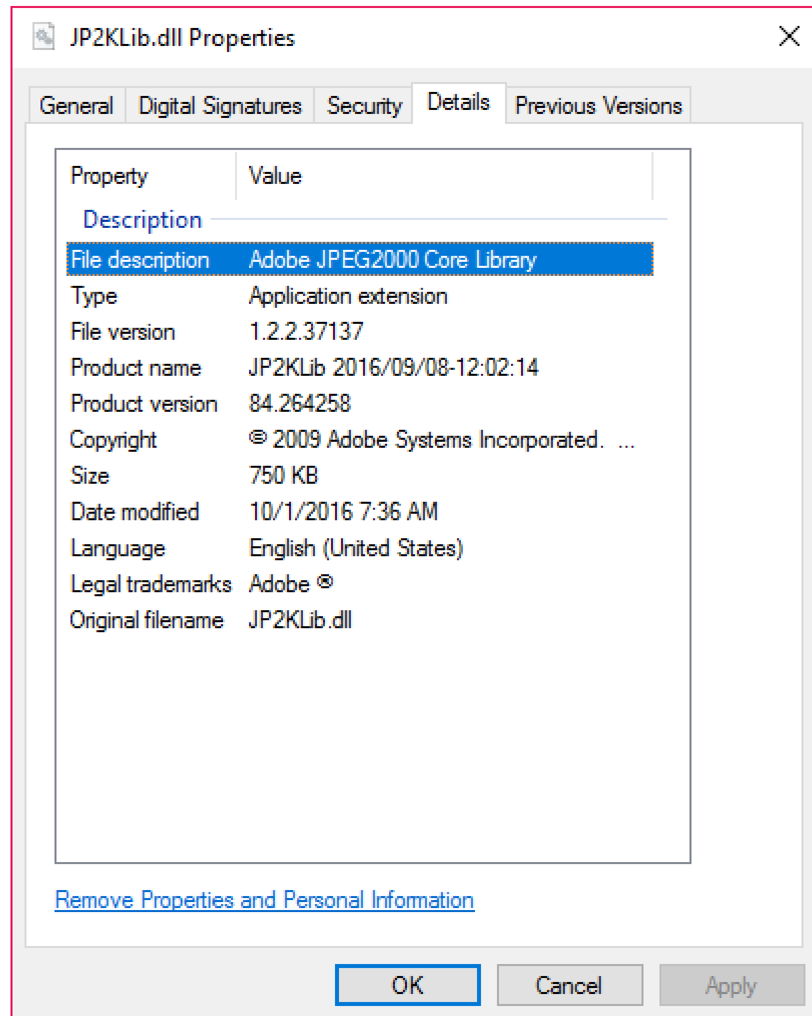
2.3 Installation Files

For closed source software, a good way to find the attack surface is to investigate the files under the installation directory, especially the executable files. To figure out the functionality of a specific executable file, we can focus on the following information.

- File name
- Properties
- Internal strings, including both ASCII and Unicode strings
- Function names, including internal symbols and export functions

- Copyrights information

For example, by analyzing the properties of a file, we can conclude that the file JP2KLib.dll in Adobe Reader's installation directory is responsible for parsing JPEG2000 images. The following figure shows the property page of the file.



The following table shows the roles of part of the files in Adobe Acrobat and Reader's installation directory.

File Name	Description
AcroRd32.dll	Core Library
ACE.dll	Color Engine
AGM.dll	Graphics Manager
AXSLE.dll	XSLT Engine
CoolType.dll	Font Engine
JP2KLib.dll	JPEG2000 Codec Library
JSByteCodeWin.bin	JavaScript Functions
AcroForm.api	Acrobat Forms Plugin (XFA)

Annots.api	Annotation Plugin
EScript.api	JavaScript Engine
ImageConversion.api	Image Convertor (Acrobat only)
Multimedia.api	Multimedia Plugin
PPKLite.api	Public-Key Security Plugin
weblink.api	WebLink Plugin

2.4 Open Source Projects

Another way to find the attack surface is to investigate similar open source projects. PDFium is a famous open source PDF rendering engine which is based on Foxit Reader's technology and maintained by Chromium's developers. In fact, we can find lots of similar code between PDFium and Foxit Reader by comparing the source code of PDFium and the disassembly code of Foxit Reader.

We can try to find the attack surface by analyzing PDFium's source code. But for PDFium, an alternative way is to analyze the libFuzzer components. Currently, PDFium has 19 official fuzzers in the "testing/libfuzzer" directory [12]. The following table shows the detailed information of these fuzzers.

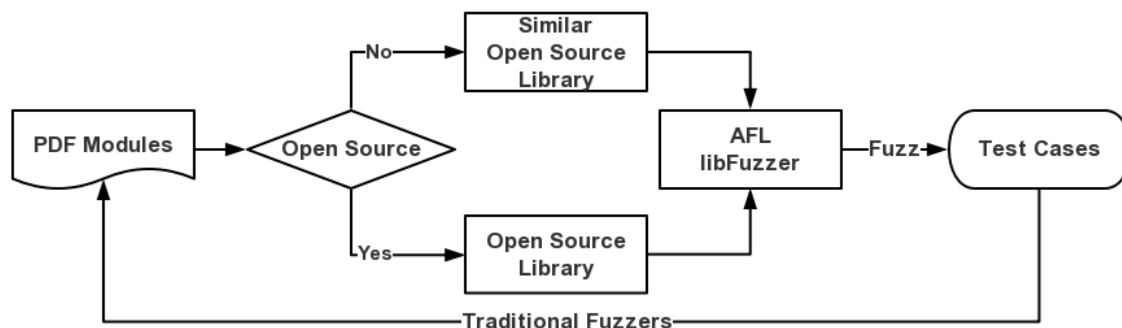
Fuzzer Name	Description
pdf_cfx_saxreader_fuzzer.cc	SAX reader fuzzer
pdf_cmap_fuzzer.cc	Font cmap fuzzer
pdf_codec_a85_fuzzer.cc	ASCII85 decode fuzzer
pdf_codec_bmp_fuzzer.cc	XFA BMP fuzzer
pdf_codec_fax_fuzzer.cc	CCITTFax fuzzer
pdf_codec_gif_fuzzer.cc	XFA GIF fuzzer
pdf_codec_icc_fuzzer.cc	ICC profile fuzzer
pdf_codec_jbig2_fuzzer.cc	JBig2 fuzzer
pdf_codec_jpeg_fuzzer.cc	JPEG fuzzer (XFA & raw JPEG)
pdf_codec_png_fuzzer.cc	XFA PNG fuzzer
pdf_codec_rle_fuzzer.cc	Run length decode fuzzer
pdf_codec_tiff_fuzzer.cc	XFA TIFF fuzzer
pdf_css_fuzzer.cc	XFA CSS fuzzer
pdf_fm2js_fuzzer.cc	Unknown
pdf_hint_table_fuzzer.cc	Hint table fuzzer
pdf_jpx_fuzzer.cc	JPEG2000 fuzzer
pdf_psengine_fuzzer.cc	PostScript fuzzer
pdf_streamparser_fuzzer.cc	Stream parser fuzzer
pdf_xml_fuzzer.cc	XML fuzzer

3. Test cases

Test case, or seed file, plays an important role in the fuzzing process. For traditional mutation based fuzzer, more test cases mean more possible code coverages which eventually mean more possible crashes. To collect more test cases, writing a crawler is acceptable in most cases, but there are some alternative ways. Here I summarized two possible methods to collect test cases.

3.1 Test cases of code coverage based fuzzers

American fuzzy lop (a.k.a AFL) and libFuzzer are two famous code coverage based fuzzers. For AFL fuzzer, a single and small test case is enough to drive the fuzzing process. For libFuzzer, it usually consumes a minimized set of test cases as the input data, but it still could work even without any initial test cases. The common feature of these two fuzzers is that they will generate lots of test cases to get higher code coverage rate. So why not reuse the test cases generated by AFL or libFuzzer? To achieve this goal, we have to fuzz an open source library, or a similar one, with AFL or libFuzzer. The following figure shows the process of this method.



3.2 Test suites of open source projects

Another way to collect test cases is to reuse open source project's test suites. Generally speaking, popular open source project also maintains a test suites repository which contains lots of valid and invalid files. Some of the test cases can crash the old version binaries directly. It's a good idea to use the test suites

as the seed files for the fuzzer. For not frequently used file formats, it's even hard to crawl some from the search engines. For example, it's hard to collect some JPEG2000 images from Google, but you can get hundreds of files from OpenJPEG [13]. The following table shows some of the available test suits.

Project	Format	Test Case Repository
PDFium	PDF	https://pdfium.googlesource.com/pdfium_tests/
		https://pdfium.googlesource.com/pdfium/testing/resources/
PDF.js	PDF	https://github.com/mozilla/pdf.js/tree/master/test/pdfs
OpenJPEG	JPEG2000	https://github.com/uclouvain/openjpeg-data
LibTIFF	TIFF	ftp://download.osgeo.org/libtiff
Google Noto Fonts	TTF	https://www.google.com/get/noto/

4. Fuzzing Tricks

Efficiency is an important metrics for fuzzers, especially when the computing resource is limited. Here I summarized two fuzzing tricks for improving efficiency.

4.1 Write PDF makers

Generally speaking, a PDF file is composed of plain texts and binary data. It's not a good idea to fuzz PDF directly if you already have a concrete target, such as images, fonts, etc. We can write PDF makers so that we'll only mutate the data that we're interested.

There are some third-party PDF makers that could convert files, such as images and fonts, to PDF files. But it's not a recommended solution because the error checking functionality in the tools may lead to lose lots of malformed test cases. In such cases, read the standard documents and write a temporary PDF maker is recommended. The technique details of PDF makers will not be discussed in this paper since it's not a tough task.

4.2 Fuzz third-party libraries

It's not strange for large software to use open source libraries. It's worth a try to fuzz third-party libraries to reveal security flaws. The following list shows the

advantages of fuzzing third-party libraries.

- Fuzz open source library with AFL or libFuzzer is more efficient
- Target software may affected by known vulnerabilities
- Zero day vulnerabilities affect all target software that use the library

The following table shows some of the open source projects that used by Adobe Acrobat and Reader.

Module	Description	Open source project
AXSLE.dll	XSLT engine	Sablotron
AcroForm.api	XFA Form	libpng, libtiff
EScript.api	JavaScript engine	Spidermonkey
ImageConversion.api	PDF converter	libpng, libtiff, libjpeg(may be libjpeg-turbo)

Last year I discovered an Out-of-Bounds write vulnerability in libtiff's PixarLogDecode function and reported to Chromium. The posts indicate that this vulnerability was also discovered by Mathias Svensson of Google [14] and Tyler Bohan of Cisco Talos [15]. The CVE identifier of this vulnerability is CVE-2016-5875. The following table shows the PDF readers that affected by this vulnerability.

Product	Affected	Remark
Adobe Acrobat Pro DC	Yes	ImageConversion Plugin (Rendering engine not affected)
Google Chrome	Yes	Chrome Canary, Dev, and Beta with XFA enabled
Foxit Reader	Yes	Rendering engine ConvertToPDF Plugin
Adobe Reader DC	No	Rendering engine not affected

For Adobe Acrobat and Reader, the rendering engine was not affected because that PixarLog compression support was not configured in AcroForm.api. For Google Chrome, the Canary, Dev, and Beta versions with XFA enabled were affected (XFA was enabled in Chrome Canary, Dev, and Beta versions shortly and disabled soon). For Foxit Reader, both the rendering engine and the ConvertToPDF plugin were affected.

4.3 Write wrappers

PDF readers or web browsers are large software, creating an instance of these products is time consuming, especially creating the instance again and again in the fuzzing process. To avoid loading unnecessary modules and initializing unnecessary data, writing wrappers is a good choice.

For open source projects, it's very easy to write a wrapper. For products that supplies APIs, such as Foxit Reader and Windows PDF Library, it's also not hard to write a wrapper. But for products that do not meet the mentioned conditions, we may need to do some reverse engineering work to write a wrapper.

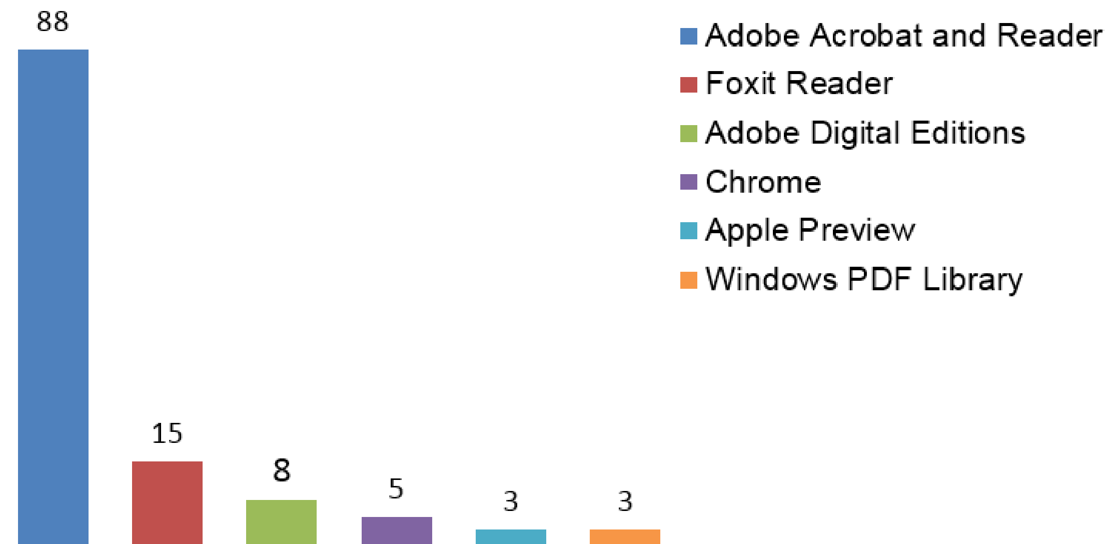
The Windows.Data.PDF.dll is responsible for rendering PDF in Edge browser and is shipped within the operating system since Windows 8.1. This library can be interacted through Windows Runtime APIs. The post [16] shows how to use C++ to write a wrapper for rendering PDF.

5. Results

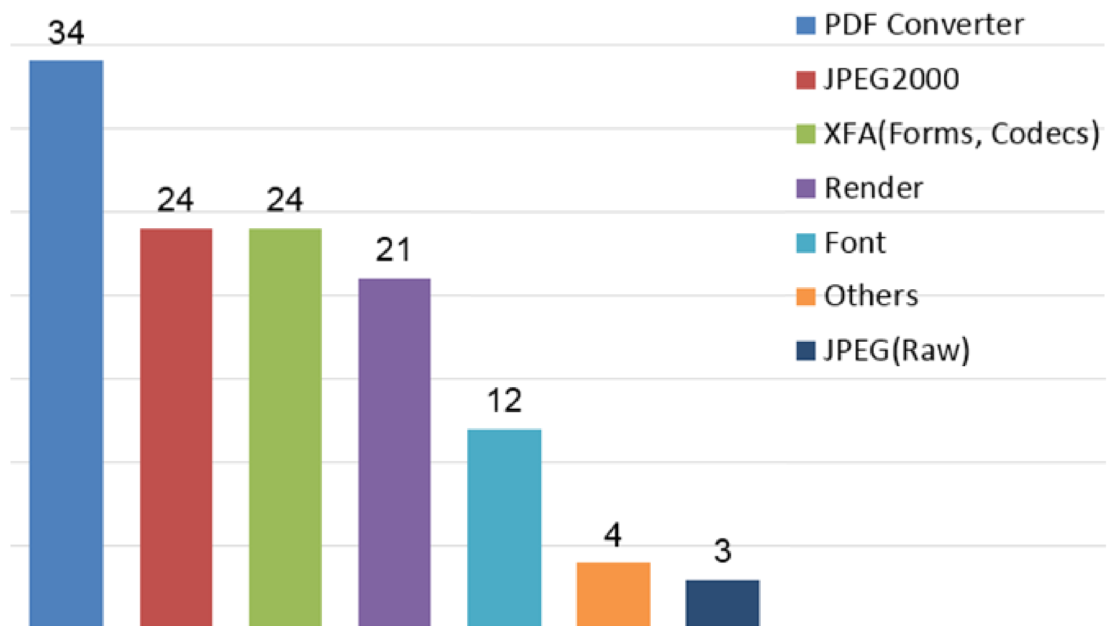
The research started since December 2015. It mainly focus on Adobe Acrobat and Reader and most of the vulnerabilities were found in the products. However, this does not mean that the products are more vulnerable than other PDF readers. During last year, 122 vulnerabilities have been patched by vendors and assigned with CVEs. It should be noted that a vulnerability will be excluded if it meets one of the following conditions.

- Vulnerabilities that do not affect stable versions of PDF readers
- Vulnerabilities that have not been fixed by vendors
- Vulnerabilities that have been reported by other researchers

The following figure shows the vulnerability distribution sorted by vendor.



The following figure shows the vulnerability distribution sorted by attack surface.



Once again, it should be noted that the vulnerability data in this paper is only for reference, it does not mean which product is more vulnerable than others.

6. References

[1]. Document management - Portable document format - Part 1: PDF 1.7, http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/PDF32000_2008.pdf

[2]. Dumb fuzzing XSLT engines in a smart way, http://www.nosuchcon.org/talks/2013/D1_04_Nicolas_Gregoire_XSLT_Fuzzing.pdf

[3]. Abusing Adobe Reader's JavaScript APIs, https://media.defcon.org/DEF%20CON%2023/DEF%20CON%2023%20presentations/DEFCON-23-Hariri-Spelman-Gorenc-Abusing-Adobe-Readers-JavaScript-APIs.pdf

[4]. Abusing the Reader's embedded XFA engine for reliable Exploitation, https://www.syscan360.org/slides/2016_SG_Sebastian_Apelt_Pwning_Adobe_Reader-Abusing_the_readers_embedded_XFA_engine_for_reliable_Exploitation.pdf

[5]. ISO 32000-1:2008, https://www.iso.org/standard/51502.html

[6]. JavaScript for Acrobat API Reference, http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/js_api_reference.pdf

[7]. XML Forms Architecture (XFA) Specification, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.364.2157&rep=rep1&type=pdf

[8]. FormCalc User Reference, http://help.adobe.com/en_US/livecycle/es/FormCalc.pdf

[9]. Zero Day Initiative's published advisories, http://www.zerodayinitiative.com/advisories/published/

[10]. Chromium issue tracker, https://bugs.chromium.org/p/chromium/issues/list?can=1&q=Type=%22Bug-Security%22

[11]. Adobe Security Bulletins and Advisories, https://helpx.adobe.com/security.html#acrobat

[12]. Official fuzzers for PDFium, https://pdfium.googlesource.com/pdfium/+/refs/heads/master/testing/libfuzzer/

[13]. OpenJPEG data, https://github.com/uclouvain/openjpeg-data

[14]. Seclists, http://seclists.org/oss-sec/2016/q2/623

[15]. LibTIFF Issues Lead To Code Execution, http://blog.talosintelligence.com/2016/10/LibTIFF-Code-Execution.html

[16]. Using WinRT API to render PDF, http://dev.activebasic.com/egtra/2015/12/24/853/