

Portable Document Format (PDF) files are a cross-platform file format that supports links, images, and fonts. The flexibility of the PDF format makes these files very useful and widely used by individuals and organizations all over the world. At the same time, this format is very appealing for cyber criminals, as they can create valid looking PDF documents that will deliver malicious code or trick users into clicking links. A [2023 report](#) found PDFs were the most commonly attached malicious file type for phishing emails.

Many attacks start from a received file, usually attached to a phishing email. Then once the victim opens the file or interacts with it by clicking on links or buttons, the next stage of the attack is executed. The purpose of the attack can be **information stealing, installing a backdoor, gaining access to the system, and more.**

One of the challenges of incident responders is to identify and classify the malicious files that were used in the attack that compromised the endpoint. PDF files make the process more time consuming because PDF files can carry malicious code that is hidden and compressed inside the streams of the file, all while these files are widely used for legitimate business both internally and externally in organizations. On top of that, alert fatigue can be a cause of missed alerts and increase the response time of new files to analyze.

In this article, we will describe the PDF format and how it can be abused to deliver malware. Then we will show how you can identify and detect a malicious PDF file using open-source and free tools. At the end we'll look at how you can automatically collect and analyze PDFs for ongoing alert triage.

Table of contents

- [What is the PDF File Format?](#)
- [How PDF Files Get Used to Deliver Malware](#)
- [Investigating Suspicious PDF Files with Open-source or Free Tools](#)
 - [Example 1 of a malicious PDF](#)
 - [Example 2 of a malicious PDF](#)
- [Scanning a High Volume of PDFs for Malware](#)

What is the PDF File Format?

The PDF format was created by Adobe in 1993, as a text-based structure that gives users a reliable way to present documents regardless of the operating system and the software they are using. Besides text, PDF files can present a wide variety of content such as images, links, video files, 3D objects, editable forms and much more. Many products include features to download or

“save as PDF”, allowing users to edit the content in another format ([like Microsoft Office files](#)) before creating the PDF.

The PDF structure is hierarchical and contains four main parts:

1. Header – Specifies the version number of the PDF.

2. The body – The document’s part that holds all of the information including text and other elements such as images, links, etc.

The body of the PDF file contains different objects which can reference each other, the objects have different types:

- Names – `/name` backslash followed by ASCII characters – setting a unique name.
- Strings – `(text)` its full syntax is a bit complex but what’s important is to know that it is enclosed in parentheses.
- Arrays – enclosed with square brackets `([...])` can contain other objects.
- Dictionaries – table of key and value pairs. The key is a name object and the value can be any other object. Enclosed within double angle brackets `(<<...>>)`
- Streams – contains embedded data structures like images (or code) which can be compressed. Streams represented by a dictionary that set the stream’s length with the key `/Length` and encoding `/Filters`.
- Indirect object – object that has a unique ID, the object starts with the keyword `obj` and ends with `endobj` other objects can reference the object using its ID. For example a reference to object with ID 3 we would look like this: `3 0 R`

3. Cross-reference table – Specifies the offset from the start of the file to each object in the file, so that the PDF reader will be able to locate them without loading the whole document (it can save time when opening big files).

4. Trailer – Specifies information about the cross-reference table so the PDF reader will be able to find the table and other objects. PDF readers start reading the file from the end, let’s look at the example below:

```
xref
0 14
0000000000 65535 f
0000000015 00000 n
0000000660 00000 n
0000000803 00000 n
0000001007 00000 n
0000001322 00000 n
0000001049 00000 n
```

```
0000001410 00000 n
0000001200 00000 n
0000001461 00000 n
0000001513 00000 n
0000001573 00000 n
0000001632 00000 n
0000001737 00000 n
trailer
<</Size 14/Root 12 0 R/Info 13 0 R/ID [<019e8b45c3227a3f8f35b7a9a09c2f70><019e8b45c3227a3
%iText-5.5.10
startxref
1901
%%EOF
```

The first line (from the bottom) is **%%EOF** above is the offset of the cross-reference table – 1901.

Above that is the trailer that specifies different settings:

- **Size** – number of entries in the cross-reference table
- **Root** – which entry in the table holds the offset for the root object. This object is the Document Catalog; it contains information about how the file will be presented and references to other objects that describe the document's content.
- **Info** – which entry in the table holds the offset for the document's information dictionary.

PDF files can be modified so additional elements (such cross-reference tables) will be appended to the end of the file. Now that we understand the format, let's see how it can be used by attackers to conceal malicious code.

The screenshot shows a presentation slide with a blue header bar. The main title is "Attack Vectors & Techniques". Below the title is a bulleted list of attack vectors:

- Phishing attacks
 - Credentials stealing
 - Installing backdoors
 - Setting persistence
- How?
 - Links to malicious sites
 - Javascript
 - Exploit PDF readers' vulnerabilities
 - Embedded streams and files within the PDF

On the right side of the slide, there is a cartoon illustration of Patrick Star from SpongeBob SquarePants, standing in a sandy area with a blue background. In the top right corner of the slide, there is a small video feed showing a person with headphones, likely the speaker.

Check out this video on our YouTube channel about analyzing PDFs, where I cover the information in this blog as well as four examples of malicious PDFs that were used in real attacks.

How PDF Files Get Used to Deliver Malware

PDF files support a wide variety of data types that can be present (and not necessarily visible). Threat actors fully control the content of the files they send to lure victims and they use the different capabilities of the PDF format for their attacks.

Many phishing attacks will contain **links**, which may appear as clickable images of buttons, coupons, fake CAPTCHA, fake play buttons, or **QR codes**. The purpose of these embedded files is to redirect the victims to sites controlled by the threat actors where they can proceed to the next stage of the attack.

PDF files natively support **JavaScript**, so attackers can create files that will **execute scripts once a file has been opened** at this stage to download additional payload or steal information.

Another way in which threat actors can use the format is to deliver malware in the **PDF streams**. Streams can contain any type of data (including scripts and binary files) and they can be compressed and encoded which makes it harder to detect embedded code inside files. The compression technique is specified with the name `/filter` (as mentioned in its part of the dictionary that describes a stream). A stream can have more than one filter.

There are many PDF readers that are being used, some are multiplatform, others based on web browsers but like any other software they have bugs and vulnerabilities. Adobe PDF Reader alone has **91 reported vulnerabilities**. Therefore threat actors can make PDF files that will exploit vulnerabilities, which will allow them to execute code and gain access to the victim's endpoint.

Investigating Suspicious PDF Files with Open-source or Free Tools

Example 1 of a malicious PDF

Let's investigate the following PDF file (MD5: a2852936a7e33787c0ab11f346631d89).

The first tool that we are going to use is **peepdf**, a **free python tool** that parses PDF files allowing us to get the types and content of each object. It will also color the object and highlight the objects that make the file suspicious, like the presence of JavaScript and embedded files.

After running the peepdf with the PDF file we get the output below. We can see that the file was updated and it has two versions. In the later version we can spot an encoded JavaScript code which makes this file suspicious so we will extract the content of the object and investigate it.

```
remnux@b262b6d26ce2:~$ peepdf files/2e26d1a3d65d7e15658033c8936c93d74cd7a1b0214c98d9a2e575fa4017d123.sample
Warning: PyV8 is not installed!!

File: 2e26d1a3d65d7e15658033c8936c93d74cd7a1b0214c98d9a2e575fa4017d123.sample
MD5: a2852936a7e33787c0ab11f346631d89
SHA1: 884a10943aff0c3c5b97ac3dae94db4e81e24b2c
SHA256: 2e26d1a3d65d7e15658033c8936c93d74cd7a1b0214c98d9a2e575fa4017d123
Size: 76918 bytes
Version: 1.7
Binary: True
Linearized: False
Encrypted: False
Updates: 1
Objects: 21
Streams: 7
URIs: 1
Comments: 0
Errors: 0

Version 0:
    Catalog: 15
    Info: 13
    Objects (15): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
    Streams (5): [7, 8, 9, 12, 14]
        Encoded (4): [7, 8, 9, 12]
    Objects with URIs (1): [5]
    Suspicious elements:
        /OpenAction (1): [15]
        /Names (1): [15]

Version 1:
    Catalog: 15
    Info: 13
    Objects (6): [13, 14, 15, 16, 17, 18]
    Streams (2): [14, 18]
        Encoded (1): [18]
    Objects with JS code (1): [18]
    Suspicious elements:
        /OpenAction (1): [15]
        /Names (2): [15, 16]
        /JavaScript (2): [15, 17]
        /JS (1): [17]
```

Output of peepdf

To extract the data from the suspicious object number 18 we will use [another open-source tool](#) called [pdf-parser](#) created by Didier Stevens. This free python tool allows us to inspect and extract different objects. The command below will extract the JavaScript saved in the object into a file called `extract_js`:

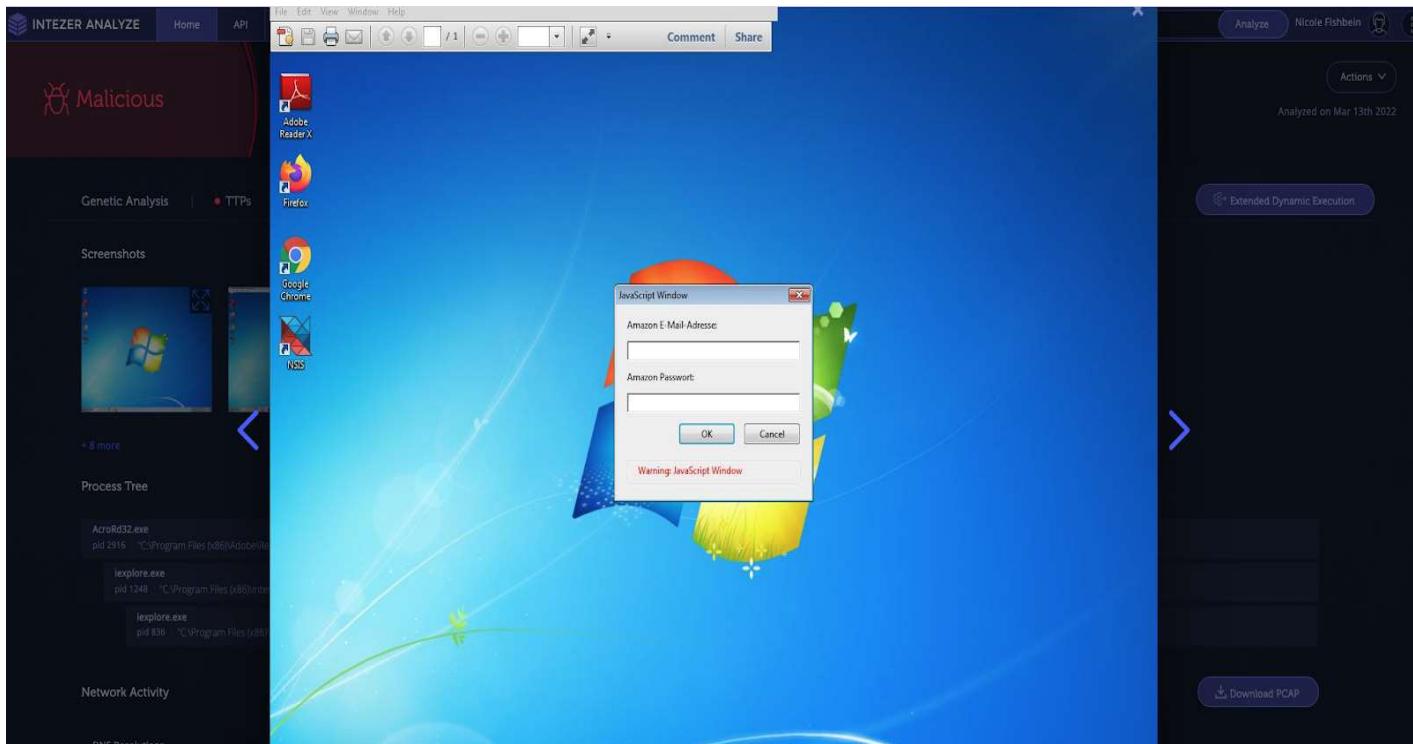
```
pdf-parser.py -f -o 18 -d extract_js files/example1.pdf
```

Upon inspecting the JavaScript code we can determine that the PDF file will open a window that will ask the victim to login into his Amazon account... but the data will be actually be submitted to a malicious site [http://sellercentral\[.\]amazon.de.56U8GTHDGT4U7YWEWE84GTYS.abecklink.com](http://sellercentral[.]amazon.de.56U8GTHDGT4U7YWEWE84GTYS.abecklink.com) for credential stealing.

We could also upload the file to a [free Intezer account](#) to get fast, deep answers about whether the file is malicious and what it does. (If you don't have an account, you can [sign up for one here now](#) or learn more [about what you can do with a free account here](#).) In the screenshot below is the execution of the PDF file in Intezer, which lets us inspect the behavior of the file.

This PDF was sent as part of a [phishing scam](#) that targeted German speaking victims: the PDF was attached to an email regarding a "tax invoice" asking the victims to open the attached

document and login to their Amazon account.



The PDF file as it can be seen in the [behavior](#) tab of Intezer's analysis.

Example 2 of a malicious PDF

Let's take a look at another PDF file (MD5: 1ba5c7ecab62609e4f1d44192cef850e). Once again we could start by running peepdf on the file to understand if the file might be malicious.

```

File: 7a810869705eed61ae1afa60c0f4cf202fe4137fa7ffb164d523d9d380279ee6.sample
MD5: 1ba5c7ecab62609e4f1d44192cef850e
SHA1: f637574b0166e6db975993e8744a8da5f345ea7b
SHA256: 7a810869705eed61ae1afa60c0f4cf202fe4137fa7ffb164d523d9d380279ee6
Size: 4618779 bytes
Version: 1.1
Binary: True
Linearized: False
Encrypted: False
Updates: 0
Objects: 9
Streams: 2
URIs: 0
Comments: 0
Errors: 0

Version 0:
Catalog: 1
Info: No
Objects (9): [1, 2, 3, 4, 5, 6, 7, 8, 9]
Streams (2): [5, 8]
Encoded (1): [8]
Objects with JS code (1): [9]
Suspicious elements:
/OpenAction (1): [1]
/Names (1): [1]
/JS (1): [9]
/JavaScript (1): [9]
/EmbeddedFiles: [1]
/EmbeddedFile: [8]

```

Peepdf output

As we see in the output below, the file contains a JavaScript object (object number 9) and one embedded file (object number 8). Let's inspect the content of object 8 by running pdf-parser as we

did in the previous example:

```
remnux@6866a79d9f93:~$ pdf-parser.py -o 8 -f -d drop_file2 exmple2.pdf
obj 8 0
Type: /EmbeddedFile
Referencing:
Contains stream

<<
/Length 2004304
/Filter /FlateDecode
/Type /EmbeddedFile
>>
```

The content of the embedded file is saved to drop_file2, we see that the stream containing the data is The content of the embedded file is saved to drop_file2. We can see that the stream containing the data is decoded with FlateDecode therefore we use the -f flag which tells pdf-parser to pass the data through filters. Next we'll check what is the type of the embedded data, so let's use the file command on the extracted file:

```
remnux@6866a79d9f93:~$ file drop_file2
drop_file2: Rich Text Format data, unknown version
```

At this point, we need to investigate an RTF file. We'll use rtfobj from oletools to inspect the file:

```
remnux@6866a79d9f93:~$ rtfobj drop_file2
rtfobj 0.60 on Python 3.8.10 - http://decalage.info/python/oletools
THIS IS WORK IN PROGRESS - Check updates regularly!
Please report any issue at https://github.com/decalage2/oletools/issues
```

```
=====
File: 'drop_file2' - size: 4967656 bytes
+-----+
id index   IOLE Object
+-----+
0 |00000E2Eh |format_id: 2 (Embedded)
|   |class name: b'EqUatiON.3'
|   |data size: 993012
|   |MD5 = '570c67219afede2ec0a6b5106274921c'
|   |Possibly an exploit for the Equation Editor vulnerability
|   |(VU#421280, CVE-2017-11882)
+-----+
remnux@6866a79d9f93:~$
```

rtfobj output for the extracted RTF file

In my previous blog about [analyzing malicious Office files](#), you can find more information about the tools and techniques for files like RTF. From the output of rtfobj we learn that the extracted RTF file exploited [CVE-2017-11882](#) – a known vulnerability in Microsoft Equation Editor that enables code execution. Since Microsoft patched this vulnerability, this particular attack would only be effective if a user failed to keep their software updated. Now, we can either continue investigating the shellcode located in the RTF file or upload the original PDF to Intezer to will automatically execute

all the steps we just did and on top of that provide a classification of the threat executed by the PDF.

The results of the analysis can be seen in the screenshot below:

Behavior tab in the PDF analysis report from Intezer

In the process tree we see the execution chain of the PDF followed by word process to open the RTF file and then the call to Equation Editor that is being exploited by the RTF. The malware is classified as Raccoon Stealer. We can see in the Genetic Analysis tab that the sample shares code with Azorult and that's because Raccoon Stealer is considered as its successor.

Genetic Analysis tab of the PDF file in Intezer.

Scanning a High Volume of PDFs for Malware

PDF files are very common and useful for all types of organizations but the flexibility of the PDF format makes it also very attractive for threat actors who use it to carry out different sorts of attacks. In this blog we presented several open-source and free tools that can be used for static analysis of a single PDF.

But manual analysis of PDFs isn't scalable for organizations when there are hundreds of files (or even more) that need to be investigated. In this case we can use a platform like Intezer that will automate the initial investigation of the files, allowing us to focus our efforts only on cases where the manual analysis is absolutely mandatory.