Portable Document Format (PDF) Security Analysis and Malware Threats

Alexandre Blonce - **Eric Filiol**[1] - *Laurent Frayssignes*
Army Signals Academy - Virology and Cryptology Laboratory

About Author(s)

Eric Filiol is Head Scientist Officer of the Virology and Cryptology Laboratory at the French Army Signals Academy. Contact Details: c/o Ecole Supérieure et d'Application des Transmissions, Laboratoire de virologie et de cryptologie , B.P. 18, 35998 Rennes, France, phone +33-2-99843609, fax +33-2-99843609,
e-mail: efiliol@esat.terre.defense.gouv.fr

Alexandre Blonce and Laurent Frayssignes are from the French Navy as IT-Security Officers and stayed at the Virology and Cryptology Laboratory in Rennes for this study.

Keywords
Malware, Portable Document Format, Document malware, security analysis, proof-of-concept, OUTLOOK.PDF*orm,* W32/YOURDE *virus.*

[1]Corresponding author

Portable Document Format (PDF) Security Analysis and Malware Threats

Abstract

Adobe Portable Document Format has become the most widespread and used document description format throughout the world. It is also a true programming language of its own, strongly dedicated to document creation and manipulation which has accumulated a lot of powerful programming features from version to version. Until now, no real, exploratory security analysis of the PDF and of its programming power with respect to malware attacks has been conducted. Only a very few cases of attacks are known, which exploit vulnerabilities in the management of external programming languages (Javacript, VBS). This paper presents an in-depth security analysis of the PDF programming features and capabilities, independently from any vulnerability. The aim is to exhaustively explore and evaluate the risk attached to PDF language-based malware which could successfully subvert some of PDF primitives in order to conduct malware based attacks. Along with a dedicated PDF document analysis and manipulation tool we have designed, this paper presents two proof-of-concepts on an algorithmic point of view, which clearly demonstrate the existence of such a risk. We also suggest some security measures at the users'level to reduce this risk.

Introduction

The widespread use of any hardware or software makes necessary their security analysis, especially with respect to the malware hazard. Applications embed more and more powerful execution features and capabilities that may enable and favour the design, writing and spread of new malware. Those features are generally motivated by the commercial need to provide more interoperability with existing applications, to make sofware easier to install, to configure and to use. But the (too) rapid development of products by the software industry makes such security analysis very difficult to conduct in time; in most cases, as long as no problem occurs, no such true risk assessment is made on a technical, auditing basis, particularly with the potential attacker's approach in mind.

The case of PDF documents (*Portable Document Format*) is probably symptomatic to that situation. The worldwide use of that multi-platform document format, due to its total portability and interoperability, whatever the platform we may consider, makes any potential security problem a very critical issue. But aside its extraordinary features that make this document format so popular, PDF document is more than a powerful document format. It is also a complete, programming language of its own, dedicated to document creation and manipulation, with relatively strong execution features. The question is to determine whether some of those features could be subverted or perverted by an attacker to design PDF document dedicated malware and thus create a new, worldwide threat.

In this paper, we address the problem of the real security level with respect to PDF documents, at the PDF code level. Up to now, no true, deep study has been conducted about the security of PDF language. Only two security problems regarding application vulnerabilities are known and surprisingly they did not suggest any further security analysis. The vulnerabilities have been patched

and that is all. The algorithmic analysis of the PDF language philosophy and of its programming capabilities has never been conducted, at least publicly.

We have conducted such a study and analysed the core execution capabilities of the PDF language that could be subverted and misused by attackers. The results is that the level of risk is far higher than expected. The main conclusion is that the extraordinary power of the PDF language, which provides flexibility, interoperability and easy-to-useness, may be a critical weakness as well. In order to validate and proove those security results, we have designed two proof-of-concepts – among many possible other ones – that have been successfully tested in operational conditions. They have clearly demonstrated that PDF could be efficiently used to attacks users through simple PDF documents, while simply using common PDF readers only. As a consequence, we suggest to limit some of the features allowed for PDF documents when working in critical environments where security is a priority.

The paper is organized as follows. In the first section, we first present the PDF language on a functional basis, in particular in terms of the technical evolutions of the different version of this language. Next, the second section is devoted to the technical analysis of the PDF decription format language and we present the structure of any PDF document. Since no suitable tool to analyze and manipulate PDF documents was available, we have designed our own tool, called *PDF StructAzer* and we present it here. The third section deals with the PDF language security analysis. We first expose the few existing PDF-based malware threats and then we explore and classify PDF language primitives in order to identify those which represent a potential risk with respect to malware writing. Finally we address the few existing security mechanisms of PDF manipulation software. The fourth section presents two proof-of-concepts that have been designed in order to evaluate the level of risk in terms of actual attacks. Finally, we conclude by presenting some security measures to take in order to potentially prevent those attacks.

Disclaimer.- The purpose of this paper is to present security analysis results regarding a critical risk which any IT experts and computer specialist must be aware of. Proof-of-concepts have been essential to validate our study since in many cases it is the only existing scientific approach to prove and convince people and particularly decision-makers. The codes of proof-of-concept will or course not be disclosed in any way and only an algorithmic description will be presented in this paper. The aim is to prevent any misuse of those critical data.

Introduction to the PDF World

The philosophy of the PDF language is to enable users to conveniently exchange and manipulate electronic documents in a reliable way, independently from any particular platform. This language inherits from the Adobe vector description language while offering more structured document than the latter, in particular by introducing objects, streams and a nested architecture. Lastly, the PDF language enables file execution contexts for an increasing interactivity and accessibility of documents. The main consequence is that **PDF documents are indeed not inert data**.

PDF exhibits a wide range of features and advantages: it is an open, evolutive description format language which is considered as reliable and secure. As a consequence, it is considered in practice as a standard by most countries (industry and gouvernmental administration).

The PDF Document Model

A PDF document can be defined as a collection of objects which describe how one or more pages must be displayed. This collection of objects can also consider additional interactive components and application data at a higher level. To manage these elements, PDF relies on the *Adobe Imaging model* inherited from the Postscript language. This general model enables to describe text, images... as an abstract model instead in terms of pixels.

Objects and components are managed through page content streams which contain operators and operands. At a higher level, the page description is enabled by means of a language which complies to the Imaging model. Displaying a PDF files is a two-step process:

- the application first generate a document description in page description language, which is independent from the hardware;
- a program, which controls the chosen interface to render the document, will interpret the description produced in the previous step.

These two steps can be operated independently both from time and space aspects, thus offering a powerful capability to exchange, store, print and display documents.

PDF Functionalities

The number of features and functionalities are so huge that it would be impossible to describe them all here. Let us just mention the most significant ones, especially in our analysis context. For more details, the reader can refer to (Adobe Systems Inc., 2004).

PDF files are binary files encoded by default in bytes. Binary format instead of text format provides a far higher portability and prevent any loss of data. Moreover, to reduce the size of files, PDF supports a number of compression standards:

- JPEG and JPEG2000 for (color and/or black and white) image compression,
- CCITT-2 and 3, JBIG2 for monochromatic images,
- LZW compression for text, graphics and images.

A PDF file must be seen as a flat representation of a data structure describing a collection of objects. Each object may refer to any other one in an arbitrary way. This means that the order of appearance of the objects whithin the file does not matter. In other words, access to any of the objects can be done randomly. For that purpose, any PDF file embeds a cross reference table which enable to access directly and randomly to any of these objects. This table is located at the end of the file. The use of such a table is essential to reduce the access time to any document object, whatever may be the size of this document (this can be compared to the RAM model).

As far as security is concerned, PDF language and PDF management applications can use:

- data encryption (RC4 and AES algorithms),

- data integrity (through hash functions),
- digital signature, including biometric-based signature.

An important but critical feature of PDF comes from the fact that a document can be modified or updated in an incremental way. This means that modifications can be simply stored within the document leaving the original one, intact. Thus the saving time is proportional to the modification size and not to the document size; thus a great amount of time can be saved. But since we also can come back to original data by cancelling modifications (by just removing the part refering to the modification), any illegitimate user can recover some confidential information hidden whithin a PDF file (see Figure 1).

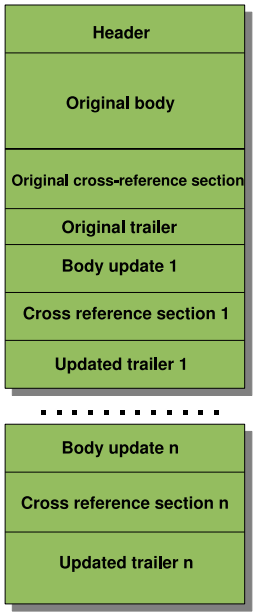

Figure 1: Structure of an updated PDF file

All this functionalities and some more programming oriented aspects, which will be presented later on, show that PDF language exhibits features that true programming languages possess: execution capabilities, various actions on the application environment. The noticeable difference is that PDF does not manage logical functions.

Technical Analysis of PDF Language

In this section, we are now going to explain how the internal structure of PDF file and how this structure is managed by the PDF language. This is essential to fully understand all these internals

mechanisms and which PDF languages primitives are involved to understand how a malware could subvert them. We just summarize here the main aspects while details can be found in (Adobe Systems Inc., 2004).

Structure of PDF Files

Any PDF file contains a least different sections, in the following order:

- the file header; this probably the most simple section. It is made of a single line which specifies the PDF language version (which range from 1.0 to 1.6);
- the file body, which generally contains the most part of the PDF code. This section is made of a list of objects which describes how the final document will look;
- the cross reference table; this table contains all the data required to the PDF management software (e.g. a reader) in order to access directly any document object without having to read throughout the file to find this object. For that purpose, each line in the table refers to a given object (more precisely an offset, in bytes, from the beginning of the document). The table always begins with the label `xref` (for *cross ref*). Then, sub-sections (one per file update; an original, unmodified file contains a single sub-section) are listed one after the another. Every sub-section begins with sub-section header made of two numbers on the same line. The first number identifies the first object in the current sub-section while the second number gives the number of objects in the current sub-section. Every line in a sub-section contains 20 bytes (including the end of line character). Here is an example of a cross reference table which contains a single sub-section with 14 objects:

```
xref
0 14
0000000000 65535 f
0000000009 00000 n
0000000074 00000 n
0000000120 00000 n
0000000183 00000 n
0000000365 00000 n
0000008910 00000 n
0000009092 00000 n
0000011135 00000 n
0000000000 00005 f
0000011349 00000 n
0000012474 00000 n
0000013599 00000 n
0000013789 00000 n
```

Lines ending with an 'n' refer to objects in use while those ending with an 'f' indicate that the object is free (it has been removed) and that its number can be used by another (future) object. We thus have two cases:

  - either the object is used; in this case, the first group of ten digits describes the offset from the beginning of the file to the object (in bytes). This group of digits is most of the time padded with 0 at its beginning to fit the line 20-byte size. The next five digits are either `00000 n` (the object is an original one and is not a reused object) or `xxxxx n`, where `xxxxx` is the generation number (the object is not an original one but a reused one) padded with 0 at its beginning;
  - or the object has been freed; in this alternative case, the first 10 digits are `0000000000`. The next five digits are used to memorize the generation number to give to a future new object (with a maximum of 65535; the object cannot then be reused).

- the trailer. Any PDF software management application always begins to read from the end of the file where this last section is located. The trailer contains different essential data, which are from the top to the bottom of the trailer:
  - the number of objects contained in the file (field `/Size`),
  - the ID of the file root document (field `/Root`),
  - the offset (in bytes) of the cross reference table (the line just above the `%%EOF` line).

Each of those sections contains the different objects which compose the document itself. In order to illustrate the internal structure of a PDF file, Table 1 presents an example of such a file.

PDF as a Programming Language

PDF is an object-oriented page description language. The *body* section contains all the objects used to represent the PDF document. These objects can belong to eight different classes:

- Boolean values,
- integers or float values,
- character strings (in ASCII or hexadecimal representation),
- labels, names,
- arrays,
- dictionaries which are arrays of object pairs (each pair is composed of a key and of a value attached to this key),
- streams,
- functions (printing optimization, graphic calculation ...),

%PDF 1.4	Header	5 0 obj	|
1 0 obj	<−	<< /Length 35 >>	|
<< /Type /Catalog	|	stream	|
/Outlines 2 0 R	|	...Page − markingoperators...	|
/Pages 3 0 R	|	endstream	|
>>	|	endobj	|
endobj	|	6 0 obj	|
2 0 obj	|	[/PDF]	|
<< /Type Outlines	|	endobj	<−
/Count 0	|	xref	Cross
>>	|	0 7	Reference
endobj	|	0000000000 65535 f	Table
3 0 obj	|	0000000009 00000 n	|
<< /Type /Pages	Body	0000000074 00000 n	|
/Kids [4 0 R]	|	0000000120 00000 n	|
/Count 1	|	0000000179 00000 n	|
>>	|	0000000300 00000 n	|
endobj	|	0000000384 00000 n	<−
4 0 obj	|	trailer	Trailer
<< /Type /Page	|	<< /Size 7	|
/Parent 3 0 R	|	/Root 1 0 R	|
/MediaBox [0 0 612 792]	|	>>	|
/Contents 5 0 R	|	startxref	|
/Resources << /ProcSet 6 0 R >>	|	408	|
>>	|	%%EOF	<−
Endobj	|		

Table 1: Example of a PDF code (a single subsection with seven objects, the first is freed and cannot be reallocated)

- the NULL object.

More complex structures can be build up with those objects. One very important feature is that objects or structures of objects can address, refer or call to resources (files, documents...) that are external to the file in which they currently are. The high level modularity provided by objects enables any PDF generating application to create its own objects structure and to store them inside a PDF file. Any other PDF application, which does not recognize those structures, will simply ignore them.

Finally, it is essential to mention that PDF language does not use control structure (*if, for, while...* statements). This is the main difference with other classic programming languages.

PDF StructAzer Tool

There exist many PDF file manipulation software (*Adobe Acrobat, PDF Creator...*) but all these software work at the object level only. Unfortunately, it is necessary to work at a lower level – the PDF code level – when analyzing PDF programming features and capabilities. This is the reason why we have developed our own PDF file analysis tool, called *PDF StructAzer* (*PDG Structure Analyzer* for short).

This software, whose front-end is depicted in Figure 2, enables to directly create a PDF file by really "programming" it. In other words, we can directly create such a file by writing it as a PDF program and then directly saving (interpreting) it as a PDF file. The main functions of this tool

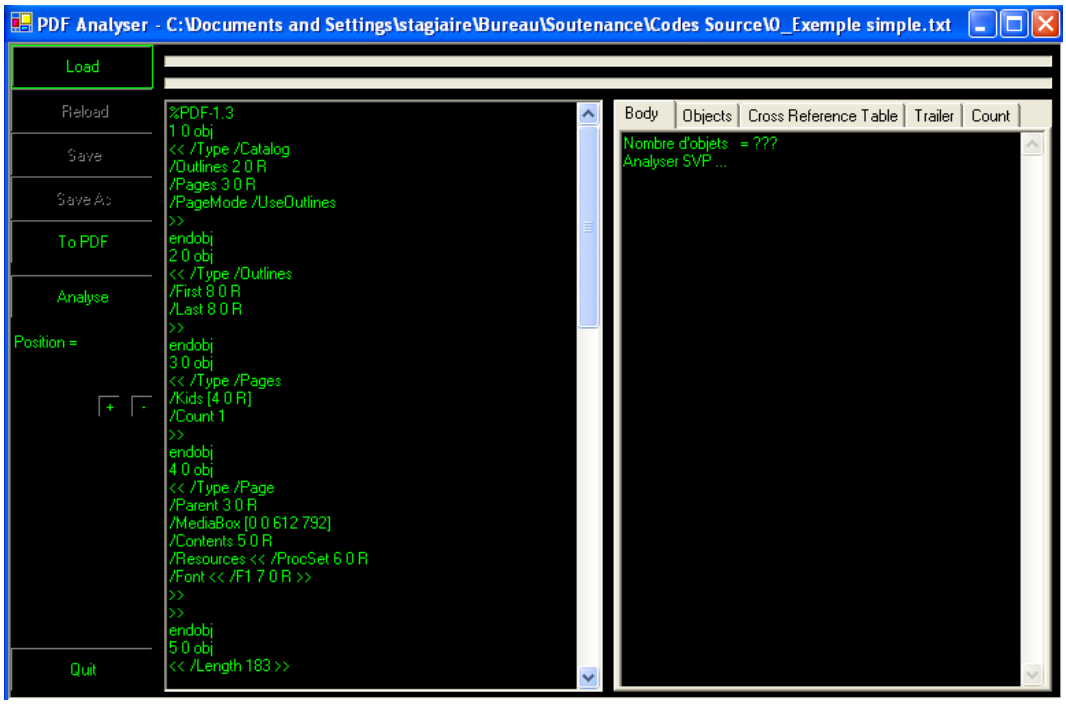

Figure 2: Front-end of the PDF StructAzer Tool

are:

- analysis of the internal structure of PDF files (identifying and then picking it out),
- calculation on the cross reference table to accurately locate and manipulate every existing object,
- basic PDF programming.

The reader will find in (Blonce - Filiol - Frayssignes, 2007) numerous examples how to use this tool to create/manipule PDF files. The PDF StructAzer Tool will be made available under Gnu Public Licence very soon.

Security Analysis of PDF Language

Known PDF-based Threats

PDF has always been considered as immune against malware since it is seen more as a description language and less – or simply not – as a manipulation one. Despite a few existing threats which have been identified since 2001, this perception has never really changed. Still now, antivirus software do not really analyze PDF documents.

At least, three threats have been identified up to now:

- in 2001, a proof-of-concept called "*Outlook_PDFWorm* or *Peachy* (Trend Micro, 2000) has been disclosed. This VBScript code spread through PDF documents sent as Microsoft Outlook email attachments. Once the document is opened, instructions to play a game are displayed as well as a link to which the user is invited to click on. The latter action is running the malicious code embedded into the PDF document. However the attack is possible when using the commercial version of Adobe Acrobat (5 or higher) only and not just the Adobe Acrobat reader;

- in 2003, the *W32.Yourde* virus appeared and exploited a vulnerability in Acrobat 5.0.5 (Adobe, 2003). Due to this vulnerability in the JavaScript parsing engine, a malicious PDF document can instruct Acrobat to write code into the user's Plug-ins folder. Any file in this folder that is developed to the Acrobat plug-in specification will automatically install and run when a user launches Acrobat. The *W32.Yourde* virus just dropped a "*Death.api*", containing the virus replication code, into the folder `C:\ProgramFiles\Adobe\Acrobat5.0\Plug_ins` and a "*Evil.fdf* JavaScript file (on the `C:` volume), which operates as a virus launcher. This vulnerability, which again affected the full version of Acrobat 5, does not affect other Acrobat versions;

- a conceptual weakness has been disclosed in 2000 (Cert, 2000) and exploited in 2003 (O. Shezaf, 2003) regarding the ability of a Adobe reader to run malicious scripts on a victim's computer when (nearly any) browser activate a link like

  `http://host/file.pdf#anyname=javascript:your_code_here`

  and uses Acrobat in embedded mode. This attacks exploits the XSS (*Cross Site Scripting*) attack technique. More recently, in december 2006 (J.-M. Laurio, 2007) another critical XSS attack through PDF documents has been disclosed.

The main observation we can made is that most of the attacks are possible when using the full version of Adobe Acrobat only, and that security analysis of PDF document has always been restricted to existing attacks, and to vulnerabilities. No study has considered the security of the PDF language itself, up to now.

Exploration and Classification of Potentially Dangerous PDF Functions

From version to version, Adobe Systems Inc. has developed the PDF language towards more interactivity with the operating system, with other files and with networks. For that purpose, actions

can be launched either automatically or manually. But the PDF language – quite like a postulate – is still considered as a passive, secure file format since its most known features are those dedicated to document description more than to object manipulation. However, the PDF language contains new programming-like characteristics that can be subverted and misused by an attacker. What is probably the most interesting point is that each PDF characteristics is generally not dangerous enough alone, with respect to the PDF malware risk: only suitable, clever combined uses of PDF languages functions can produce dangerous PDF malware threats.

During our exploratory analysis of the PDF primitives, we have identified two major kinds of actions that can be realised through those primitives:

- the *OpenAction* class, whose elements are operating actions whenever a PDF file is opened. In this case, the PDF directive `/OpenAction 9 0 R` will be used within the relevant object;
- the *Action* class, whose elements are operating actions on user's own action (e.g. activating a hypertext link). But it is possible to fool and trap the innocent user by using invisible forms (which are activated once the mouse cursor enters the invisible field), hypertext link...

Since PDF version 1.2, generating actions has become easier with the concept of "*Trigger Events*" which makes possible to create large structures and trees of actions in which each action is link to other ones. Some of those functions are particularly sensitive with respect to misuse by malware writers. Let us summarize the most critical ones (among others) along with the possible misuse:

- the *GoTo* function which performs and manages deplacement within the active document, towards a specified destination. A valid destination can be a predetermined page or an object (graphics, hypertext link, annotation...). Here is an example of possible use:

```
....
/A << /Type /Action
      /S /GoTo
      /D [2 0 R/FitR - 4 399 199 533]
   >>
....
```

  The destination is the object number 2 and the page zoom is tuned up. This command is potentially not too dangerous unless used with other critical functions. It can be used as a first step in a multi-step attack, in order to induce some user's action (e.g. to point the cursor to a file annotation, or an active, invisible or not, area, which executes some code when activated.

- the *GoToR* function which generalizes the *GoTo* function to resources that are outside the active document, in another PDF file. This external resource is then opened in place of the active document. This could be used to launch some external malicious code (e.g. logic bomb). The main risk lies in the fact that the command is not included in the external resources, thus making detection quite impossible. In the context of *k* ary malware (Filiol, 2007), this could be dramatically misused.

- the *GoToE* is a special case of the *GoToR* function and enables to access any other PDF file which embedded or put as an appendix to the active PDF file. Embedded files can themselves contain another embedded files, thus piling up different files.

```
....
<< /Type /Action
   /S /GoToE
   /D (Chapter 1)
   /F (someFile.pdf)
   ....
>>
```

  This can be misused to include some malicious code in a third or fourth level embedded file.

- the *launch* function can launch an application, open or print a document. This function accepts optional arguments with respect to Windows, Mac or Linux systems, to manage OS-specific actions/applications. In the following code:

```
....
<< /Type /OpenAction
   /S /Launch
   /F (/c/SecretFiles/password.doc)
   /O (print)
>>
....
```

  the misuse of the *Launch* function enables to eavesdrop the administrator's password file (a critical error) through the network printer, whenever the malicious PDF document is opened. The number of possible misuses for this command is quite infinite (run malicious code, eavesdrop data, steal data, subvert legitimate application...). This is why the *Launch* function is probably the most critical one.

- the *URI* function which enables to access remote resources by mean of an *Universal Resource Indicator* (hypertext) link.

```
....
<< /Type /OpenAction
   /S /URI
   /URI (http://www.some_phishing_site.com)
>>
....
```

This example clearly shows that it is possible to access any external object, on any Intranet or on the Internet; thus making the risk exploding, unlashing the different dangers that are present in Internet, for example.

- with the *SubmitForm* function we can send predefined labels and data of interactive form fields towards a given URL (e.g. a web server address) as shown in the following chunk of PDF code:

```
....
<<
   ....
   /S /SubmitForm
   /F << /FS
         /URL
         /F (ftp://www.rogue_website.com/song.mp3)
      >>
>>
....
```

  In this example, form data are sent to a website and hidden into a innocent-looking file.

- the *ImportData* function enables to import data into the active PDF file (under the *Forms Data Format* (FDF)). This function thus can be used to steal data from the computer on which a malicious PDF file is opened. Moreover, this function can be misused to conduct Cross Scripting attaks.

- the *JavaScript* function enables to execute JavaScript file, through the *JavaScript* PDF application module, provided that its use complies with the definition of the PDF application library. This function is critical since it is possible to bypass some software security protection before Adobe PDF version 8.0, and of course to execute malicious scripts. As an example, let us consider the following script which just display a message box.

```
....
9 0 obj
        << /Type /OpenAction
           /S /JavaScript
           /JS 10 0 R
        >>
endobj
10 0 obj
....
app.alert({cMsg:''Hello world'',cTitle:''Hello world box''});
....
endobj
```

It is worth mentioning that in most of the cases a malware will use and combine more than one such critical function.

PDF Security Mechanisms

Adobe Systems Inc has implemented some security mechanisms in its Applications (*Adobe reader* and *Adobe Acrobat*) in order to alert the user in case of some malicious or potentially attempts. These alert measures are most of the time just message box alerts, asking the user to confirm or cancel a given action. Unfortunately, it is possible to bypass these security mechanisms. In order to more deeply forecast the scope and impact of a potential PDF malware, we have analyzed all those security mechanisms up to the latest Adobe Reade 8. Most of the aspects we present hereafter apply to any former version. Without loss of generality, we have conducted this study for the French localized version under Windows.

Application-level security: alert message boxes

In the `C:\ProgramFiles\Adobe\reader8.0\Reader` directory, two configuration files are involved in the management of alert message boxes: `RdLang32.FRA` and `AcroRd32.dll`. The main weakness comes from the fact that:

- there is no integrity checking for those files thus allowing modify the alert message in order to fool the user. We could do such modification without triggering any alert;
- those files are not in read-only access. Thus a malware can modify them very easily, in particular in multi-step attacks.

Operating system-level security: configuration file and registry keys

This security is enforced directly at the operating system level. This means that any weakly configured system will ease PDF-based attacks. Our analysis has clearly shown that this was the most critical part in the PDF environment security.

In order to not disclose technical details that could benefit to bad guys, we will just summarize our results and give some of the most relevant registry keys along with the SUITABLE value they SHOULD take, for an efficient security policy, with respect to PDF malware attacks. This configuration was sufficient to prevent most of our proof-of-concept attacks presented in the next section but it is essential to keep in mind that any malware can modify any registry key. Consequently, frequent control should be an essential part of any security policy.

- Access to Internet is managed through the registry key givern hereafter.

```
HKU\S-1-5-21-1202660629-706699826-854245398-1003\Software\Adobe\
Acrobat Reader\8.0\TrustManager\cDefaultLaunchURLPerms\
```

To block access to any website, the *iURLPerms* subkey must be set to `0x00000001`. However, it is possible to manage a single website through the *tHostPerms* subkey. As a few examples, we have:

- to authorize access to `www.google.com` only, just set the key as follows:

```
HKU\S-1-5-21-1202660629-706699826-854245398-1003\Software\Adobe\
Acrobat Reader\8.0\TrustManager\cDefaultLaunchURLPerms\tHostPerms:
"version:1|www.google.fr:2"
```

- while on the contrary, to deny access to `www.google.com` only, just set the key as follows:

```
HKU\S-1-5-21-1202660629-706699826-854245398-1003\Software\Adobe\
Acrobat Reader\8.0\TrustManager\cDefaultLaunchURLPerms\tHostPerms:
"version:1|www.google.fr:3"
```

- Since *Adobe Reader* version 8, full screen display must be confirmed by the user. Indeed it can be misused to simulate a GUI or an Internet website. However some values for the following registry key (with the suitable value for security enforcement)

```
HKU\S-1-5-21-1202660629-706699826-854245398-1003\Software\Adobe\
Acrobat Reader\8.0\FullScreen\iShowDocumentWarning: 0x00000001
```

  can cancel the display of any confirmation box and thus launch the full screen display without user's control (and awareness).

- the use of *JavaScript* resources directly from a PDF file can be subverted in order to operate actions while bypassing application security local mechanisms. Indeed, JavaScrip maganement in PDF applications is done directly at the registry base level. The registry key

```
HKU\S-1-5-21-1202660629-706699826-854245398-1003\Software\Adobe\
Acrobat Reader\8.0\JSPrefs\
```

  is the most critical key with respect to *JavaScript* security within PDF application. The relevant subkeys with the suitable value for security are

  - `JSPrefs\bEnableJS` (value *0x00000000*) to forbid *Acrobat JavaScript*,
  - `JSPrefs\bEnableMenuItems` (value *0x00000000*) to restrict JavaScript execution priviliges,

  - `JSPrefs\bEnableGlobalSecurity` (value *0x00000000*) to activate the global security strategy for PDF objects.

- *Adobe* applications restrict by default the opening of appended (embedded or attached) documents. But an attacker can modify the following key

```
HKU\S-1-5-21-1202660629-706699826-854245398-1003\Software\Adobe\
Acrobat Reader\8.0\Attachments\cUserLaunchAttachmentPerms\
```

  in order to autorize it without control and user's awareness. Two subkeys are thus involved:

  - `cAttachmentTypeToPermList` (no value to forbid access to attached file with a known extension),
  - `iUnlistedAttachmentTypePerm` (*0x00000001* to forbid access to attached file with an unknown extension).

Assessing the Risk: two Proof-of-concepts

Assessing practically a risk that has been identified theoretically only cannot be done without proof-of-concepts. Convincing decision-makers, modifying a security policy in place are huge, not to say intractable, challenges. Suspicion, intellectual blindness, doubts about the operational scope are all common fences that any IT researcher had to face up at least once in his life. The only scientific approach is to use proof-of-concept validation. Of course, this must be done in a legal and secure way. In particular, this is the only way to identify possible side-effets that would make any theoretical weakness . . . just a theoretical one, with no real impact on the overall security. On the contrary, successful operational validation can enable to quickly react and prevent the weakness to be exploited by attackers, and incitate the software editor to modify his product.

In this section, we are now going to present two successful proof-of-concepts – among many others. In order to not disclosed technical data that could be misused – in particular the source code of course but also the operational context that makes the attack really successful –, we will just present them shortly, from an algorithmic point of view only. Video demos will be presented during the conference. It is essential to mention that antivirus software we have tested, did not raise any alert.

PDF-based Phishing Attack

This attack basically exploits the strong capabilities of PDF language to accurately and faithfully describe a given document or data. In this respect the full screen mode enables to display to the user a PDF document that perfectly mimick any website. For our attack we have designed a malicious PDF file which fakes the website of a French bank.

At the PDF code level, this malicious file has been prepared as follows:

- The usual *login* and *password* fields have been replaced by simple PDF form fields. However, in order to fool the user and mimick the true behaviour of a login page, the password is not displayed when typeset by the user (a sequence of star symbols is just displayed instead).
- The connection button has been replaced by an interactive widget which in reality launches the email client (see the steps of the attacks farther).

Then the attack is performed through the following steps:

1. the user opens the PDF file which displays the fake website. Every unsuspicious user will then connect typesetting his login/password to access his bank account;
2. when connecting, the PDF file transparently launches the email client and displays a fake email seemingly from the bank. This email proposes to send a security certificate to the bank for security control purpose;
3. if the user accepts to send this email, in fact the email transmits a FDF file which contains the user's login/password data, under an encoded, innocent-looking form (in case the PDF code would be analyzed);
4. once this email has made this FDF file evade, the malicious PDF file send an URL request to transparently redirect the user towards the legitimate bank website.

This simple scenario shows how the confidence in PDF format could be perverted by an attacker in order to mount a phishing attacks.

PDF-based Two-step Attack

In this attack, the attacker's aim is to fool a privileged user (e.g. a system or network administrator) to make him execute an executable code attached to the PDF file. This latter code will then target both the *Adobe Reader* application and the Windows operating system. This very simple scenario could be generalized to any k-ary code-based attack (Filiol, 2007). In our case, we have just use a ternary code which is made up of

- the attack vector which is composed of a malicious PDF file and an attached executable file F_1;
- the payload as another executable file F_2, hidden within the PDF file (which can be in the general case any separate file previously introduced into the target system). In our case, the payload simply displays a "Hello World" message box but more destructive payloads could be implemented in PDF language (refer to the next section).

The attack has then been conducted through the following steps:

1. a malicious, trapped PDF is sent to the privileged user as an attachment of a spoofed email. The email uses some social engineering tricks in order to incitate the user to activate the F_1 executable;

2. once the executable F_1 is launched, it

   (a) modifies some configuration files of *Adobe Reader* in a permanent way in order to change the most critical security parameters and manipulate message text of *Adobe* warning/alert boxes;

   (b) replicates into any non infected PDF files that are found in the target system;

   (c) finally launches the payload F_2 either automatic or upon a user's action.

   File F_1 moreover takes care of managing its action in order to prevent any conflict between the unmodified version of *Acrobat Reader* and the modified one.

This very simple yet powerful attack scheme can be generalized to produce far more sophisticated attacks.

Some Other Possible Attacks

The sophisticated features and characteristics of PDF language primitives enable to design other powerful attacks. We will just mention some of them that have been addressed in our study and have proved to be potentially very critical:

- theft of data by picking them up and hidden them into a malicious PDF file,
- eavesdropping/wiretapping of sensitive data,
- information warfare against people,
- malicious actions against the operating system and/or the file system,
- ...

Future Work and Conclusion

In this paper, we have addressed the security of PDF language primitives with respect to malware attacks. This study has clearly demonstrated that there exist a real risk that must be envisaged very seriously by IT experts. It is likely that our study did not identify all existing risks. However our analysis and results, which have been validated by a few proof-of-concept codes, strongly suggest that PDF security should be envisaged in a more accurate way in operating system management, security policies ans user awareness.

In particular, our study urges for some security measures that should be taken to limit the risk of PDF malware. Some of them could be easily implemented at the antivirus software level directly:

- integrity and access rights should be strengthened in order to forbid the modification of `AcroRd32.dll` and `RdLang32.xxx` configuration files (Adobe Systems Inc's level). More generally, any configuration file of a PDF management application should be suitably protected and administrated (read-only access mode);

- regularly monitor the registry base in order to check that PDF application related key are suitably configured; we are currently developing a free tool to perform security check on demand,
- PDF file must not be open while logged as root or as a privileged user;
- monitor any suspect or unusual aspect/behaviour of PDF management applications,
- preferably use PDF with no (too much) active/critical content, unless strictly necessary;
- systematically use digital signature to exchange PDF document.

We now intend to explore further the risk attached to PDF language, in particular firstly with respect to the Unices environments – in which the security at the operating system level "seems" to be less permissive – and second with respect to *Adobe Reader 8* whose features and primitives have been significantly increased for more document accessibility. PDF evolution towards more and more ergonomy – a new version is published every two years – asks for a permanent and deep security analysis of the PDF language power.

References

Adobe Inc (2003). Adobe Acrobat 5.0.5 Security, Accessibility and Forms Patch. http://www.adobe.com/support/downloads/detail.jsp?ftpID=2121

Adobe Systems Inc. (2004). PDF Reference Version 1.6. Fifth Edition. http://www.adobe.com/support/

Blonce, A., Filiol, E. and Frayssignes, L. (2007). Evaluation SSI des risques inhérents au format PDF et réalisation de codes "proof of concept". Virology and Cryptology Lab Technical Report.

CERT (2000). http://www.cert.org/advisories/CA-2000-02.html

Filiol, A. (2007). Formalisation and Implementation Aspects of *K*-ary (malicious) Codes. EICAR 2007 Special Issue, V. Broucek ed., Journal in Computer Virology, 3 (2).

Laurio, J.-M. (2007). Universal XSS with PDF Files: highly dangerous. http://lists.virus.org/full-disclosure-0701/msg00095.html

Shezaf, O. (2003). The Universal XSS PDF Vulnerability. http://www.owasp.org/images/4/4b/OWASP_IL_The_Universal_XSS_PDF_Vulnerability.pdf

Trend Micro (2000). Peachy Virus Analysis. http://www.trendmicro.com/vinfo/virusencyclo/default5.asp?VName=VBS_PEACHYPDF.A