RUHR-UNIVERSITÄT BOCHUM

hgi Lehrstuhl für Netz- und Datensicherheit

Vulnerability Report

Insecure Features of PDF Documents

J. Müller, D. Noss, C. Mainka, V. Mladenov, J. Schwenk

Abstract

PDF is the de facto standard for document exchange. It is common to open PDF files from potentially untrusted sources such as email attachments or downloaded from the Internet. In this report, analyze the capabilities of malicious PDF documents. We abuse legitimate features of the PDF standard itself instead of focusing on implementation bugs. Our attacks are categorized into five classes: (1) Denial-of-Service attacks affecting the host on which the document is processed. (2) Invasion of privacy attacks which track the document usage. (3) Information disclosure attacks leaking personal data out of the victim's computer. (4) Data manipulating on the victim's system. (5) Code execution on the victim's machine. An evaluation of 28 popular PDF processing applications shows that 26 of them are vulnerable at least one attack. Finally, we propose a methodology to systematically protect against attacks based on PDF features.

Contents

1 Background **3**

1.1 Powerful Document Features . . . 3

1.2 Basic Blocks . . . 3

1.3 PDF Forms . . . 4

1.4 Actions & JavaScript . . . 4

2 Attacker Model **4**

2.1 Actions of the Victim . . . 5

2.2 Attacker's Capabilities . . . 5

2.3 Winning Condition . . . 5

3 Attacks **6**

3.1 Denial of Service . . . 6

3.2 Invasion of Privacy . . . 7

3.3 Information Disclosure . . . 9

3.4 Data Manipulation . . . 11

3.5 Code Execution . . . 12

4 Evaluation **13**

4.1 Denial of Service . . . 13

4.2 Invasion of Privacy . . . 15

4.3 Information Disclosure . . . 17

4.4 Data Manipulation . . . 18

4.5 Code Execution . . . 19

5 Exploits **20**

5.1 Directory Structure . . . 20

6 Countermeasures **21**

6.1 Towards an Unambiguous Specification . . . 22

6.2 Resource Limitation and Sandboxing . . . 22

6.3 Implementing Privacy by Default . . . 22

6.4 Removing or Restricting JavaScript . . . 22

6.5 Identification of Dangerous Paths . . . 23

7 Conclusion **24**

References **25**

A Appendix **27**

1 Background

This section briefly introduces the PDF document structure. For reasons of clarity, we only describe the building blocks relevant for the attacks of this report.

1.1 Powerful Document Features

PDF is arguably the most widely used data format for office document exchange. Introduced in 1993 by Adobe Systems, the Portable Document Format (PDF) was designed to provide a consistent representation of documents, independent of the platform. It supports numerous advanced features, ranging from cryptography to calculation logic [28], 3D animations [35], JavaScript [2] up to form fields [37]. PDF document can be updated or annotated without losing previous revisions [38] and define specific actions [36], for example, to display a certain page once the document is opened. On top of this, PDF is enriched with different data formats which can be embedded into documents, such as XML [4], or Flash [3]. Each of the formats has its strengths, but allowing their inclusion also enables their weaknesses and concerns. In this report we analyze the security of native PDF functions.

1.2 Basic Blocks

A PDF document consists of four basic sections:

1. A header defining the PDF document version (1.1 to 2.0).
2. A body containing the content, a bundle of PDF objects.
3. An index table with references to each object in the body.
4. A trailer defining the root element of the document and a reference to the index table.

The most important section is the body which contains the objects – the actual content of the document. An object can, for example, define a headline, a textblock, or an image.

```
obj 10 0
<< /Length 10 >>                                         % stream length
stream                                             % start of the stream
  Content                        % content (e.g., text, image, font, file)
endstream                                            % end of the stream
endobj
```

Listing 1: The PDF object `10 0` includes a 10-byte content stream.

Every object is enclosed by the delimiters `obj` and `endobj` and has has an identifier. In Listing 1, the object's identifier is `10` in generation number `0`. Content can be provided as a string, or – as shown in Listing 1 – as a stream enclosed by `stream` and `endstream`. It can be prefaced with additional information, such as encoding or length. Streams can optionally be compressed. Many documents use `FlatDecode` for this purpose, meaning that the zlib *Deflate* algorithm is used.

1.3 PDF Forms

With PDF version 1.2, Adobe introduced *AcroForms* in 1996. Similarly to HTML forms, AcroForms allow to define input fields, checkboxes and buttons. The user-input can either be stored directly into the document (using incremental updates) or be submitted to a dedicated server. In the latter case, AcroForms use the *Forms Data Format* (FDF), which is based on raw PDF objects, for transmitting the data.

1.4 Actions & JavaScript

The PDF specification defines multiple *Actions* for various purposes. These actions can be used, for example, to navigate to a certain page in the document (*GoTo* action). Actions are commonly combined with form elements (e.g., to play a sound once a button is clicked, or to show/hide form fields), however, they can also be triggered automatically based on various events such as opening, printing or closing the document.

A special action in PDF is the execution of JavaScript. Adobe defined a basic set of functions [2], but PDF viewer applications often choose to implement a subset of Adobe's standard as well as to extent their feature set with proprietary functions. JavaScript provides a huge flexibility for documents, for example, complex input validation of forms or changing their values depending on specific conditions (e.g., locale).

2 Attacker Model

In this section, we describe the attacker model, including the attacker's capabilities and the winning condition.

2.1 Actions of the Victim

The *victim* is an individual who retrieves and opens a malicious PDF document from an attacker controlled source.

This is a realistic attack scenario, because even sophisticated users download and open PDF files from untrusted sources such as email attachments or the Internet, for example, invoices or academic papers are usually shared as PDF files. PDF documents are generally considered as relatively "safe" compared to other file formats such as Microsoft Office files, which are known to contain potentially dangerous macros [17].

To open the PDF document, the victim uses a pre-installed application which processes the file in order to display its content. Different applications may process the file, or interpret features of the PDF standard, differently, thereby enabling or disabling the various attack vectors described in this report.

2.2 Attacker's Capabilities

The *attacker* can create a new PDF file or modify an existing document which we denote as the *malicious document*. We do not require the malicious document to be compliant to the PDF specification, although the attacker targets basic functionality and features of the PDF standard. The attacker has full control over the document structure and its content. While the attacker can easily craft a malicious document which *looks* benign once opened and interpreted by the PDF application (i.e., similar to a document that the victim would expect), this is not assumed to be necessary, because all attacks are automatically triggered once the file is opened. The only interaction of the victim is to open the malicious document on the victim's computer.

For two privacy-related weaknesses – *Evitable metadata* and *Revision recovery* – the victim is the one creating the document and the goal of the attacker is to obtain potentially sensitive information from this file, such as revealing metadata or older revisions of the document.

2.3 Winning Condition

An attack is classified as *successful* if its winning condition is fulfilled. The winning condition – the goal of the attacker – is dependent on the attack class and documented in the corresponding section. For example, in the DoS attack class, the winning condition is reached if the PDF processing application can be forced to consume all available resources in terms of CPU or memory. In the information disclosure class of attacks, the winning condition is fulfilled if the attacker manages to obtain sensitive data, such local files from the victim's disk.

3 Attacks

In this section we summarize well-known attacks and propose new approaches. At the beginning of each attack category we introduce the goals of the attack and their applicability.

Methodology

To identify attack vectors, we systematically surveyed which potentially dangerous features exist in the PDF specification. We created a list of all possible attacks which can be carried out by abusing these features, and classified them accordingly. If an attack had previously been discovered, instead of simply being documented as a “feature” in the standard, we refer to the according literature once introducing the attack.

To generate our test suite of malicious PDF documents, we chose a semi-automated approach: we hand-crafted the payloads to test for a certain weakness and wrote a set of helper tools in Python, in order to generate an exhaustive set of attack variants as well as a valid PDF structure for each test case. Our efforts resulted in 213 unique PDF files, which we manually opened in 28 PDF applications to observe the result. This process can be automated by launching each test for each PDF viewer in a batch and logging the program’s behaviour.

3.1 Denial of Service

The goal of this class of attacks is to craft a document which enforces PDF interpreters to consume all available resources (i.e., computing time or memory) or causes them to crash. Note that, while the impact of DoS is limited for end-users, it can lead to severe business impairment if the document is processed on a server, for example, to generate preview thumbnails of PDF files uploaded to cloud storage.

Infinite loop Inducing an endless loop causes the program execution to get stuck. The PDF standard allows various elements of the document structure to reference to themselves, or to other elements of the same type. This can lead to cycles, if not explicitly handled by the implementation. For example, a *Pages* object may reference to other Pages, which is a known problem of the specification, already discovered in CVE-2007-0104. We systematically studied the PDF standard for further constructs that allow for reference cycles, recursion, or other kinds of loops, and found the following novel variants:

- *Action loop.* PDF actions allow to specify a *Next* action to be performed, thereby resulting in "action cycles".
- *ObjStm loop.* Object streams may extend other object streams which allows to craft a document with cycles.
- *Outline loop.* PDF documents may contain an outline. Its entries, however, can refer to themselves or to each other.
- *Calculations.* PDF defines "Type 4" calculator functions, for example, to transform colors. Processing hard-to-solve mathematical formulas may lead to high demands of CPU.
- *JavaScript.* Finally, in case the PDF application processes scripts within documents, infinite loops can be induced.

Deflate bomb Data amplification attacks based on malicious zip archives are well known (see [8, 14, 29]). The first publicly documented DoS attack using a "zip bomb" was conducted in 1996 against a Fidonet BBS administrator [12]. However, not only zip files but also stream objects within PDF documents can be compressed using various algorithms such as *Deflate* [13] to reduce the overall file size. The idea that it may theoretically be possible to build a deflation bomb for PDF was recently noted by [15]. To the best of our knowledge we are the first to actually implement such "PDF bombs".

3.2 Invasion of Privacy

In this section, we discuss privacy aspects of PDF. Our first attack, URL invocation, tracks the usage of a document by silently invoking a connection to the attacker's server once the file is opened. The other two attacks, evitable metadata and revision recovery, deal with the amount of information an attacker can learn from a document *created by* the victim.

URL invocation Tracking pixels in HTML emails are well documented[1], but the existence of similar technologies for PDF files is largely unknown to the general public. However, PDF documents that silently "phone home" should be considered as privacy-invasive. They can be used, for example, to deanonymize reviewers, journalists or activists behind a shared mailbox. The goal of this attack is to open a backchannel to an attacker controlled server once a PDF document is opened by the victim. Besides learning *when* the file was opened and by *whom* (i.e., by which IP address), the attacker may learn additional information such as the victim's

[1]Recently, [31] found backchannels in 40 out of 48 tested email clients.

PDF viewer application and operating system, derived from the *User-Agent* HTTP header. The possibility of malicious URI resolving in PDF documents has been introduced by [19] who gave an evaluation for *URI* and *SubmitForm* actions in Acrobat Reader. We extend their analysis to *all* standard PDF features that allow to open a URL, such as *ImportData*, *Launch* and *GoToR* as well as JavaScript, and to a comprehensive set of viewers.

Evitable metadata In 2005, the former US president Bush gave a speech on the war in Iraq and published a strategy document on the White House website. The metadata of the PDF document revealed a Duke University political scientist as the original author of the document [21]. Afterwards, the NSA published best practices addressing risks involved with hidden data and metadata in PDF files [5]. This example shows that there are valid use-cases where the author of a document prefers to remain anonymous. The issue of unwanted metadata in various file formats is well known and has been discussed in [7, 30]. Even though metadata is a feature of the PDF standard, from a privacy perspective creator software should avoid to include excessive metadata by default and instead let users opt-in. Although many PDF documents are created with non-PDF software (e.g., LaTeX, office suites, or system printers), all professional PDF editors offer the creation of PDF files as well. They are especially used when designing complex PDF documents that, for example, include forms and JavaScript. During the creation process, these editors generate special PDF metadata objects, which can contain sensitive information (e.g., usernames). We are the first to systematically evaluate which metadata is generated by PDF editors, and we analyze the metadata contained in a representative sample of PDF documents publicly available on the Internet.

Revision recovery The PDF standard allows editing applications to modify existing documents while only appending to the file and leaving the original data intact. Whenever new content is added to the document, it is not simply inserted into the existing body section. Instead, a new body section is appended at the end of the PDF file containing new objects.[2] This feature is called "incremental updates". It enables authors, for example, to undo changes. However, it also enables third parties to restore previous versions of the document, which may not be desired in the context of privacy and document security. Especially when sensitive content is explicitly redacted/blackened in a document to be published, this can be dangerous: Instead of deleting the underlying text object, PDF editors may simply overlay a black rectangle, allowing for easy "unredaction". Examples of poorly redacted documents revealing classified information have been published by the Washington Post [16], the Pentagon [25], Facebook [26], and many others. This is a well-known problem and has been researched for PDF documents generated by various office

[2]A new *XRef* index table and a new trailer must also be appended.

suites in [18]. However, modern PDF editors have an explicit “redact” function, which has not yet been comprehensively evaluated. Therefore we systematically analyze how document modification and text redaction is implemented in PDF editors.

3.3 Information Disclosure

The goal of this class of attacks is to leak PDF document form data, local files, or NTLM credentials to the attacker.

Form data leakage Documents can contain forms to be filled out by the user – a feature introduced with PDF version 1.2 in 1996 and used on a daily basis for routine offices tasks such as travel authorization or vacation requests. Depending on the nature of the form, user input can certainly contain sensitive information (e.g., financial or medical records). Therefore, the question arises if an attacker can access and leak such information. The idea of this attack is as follows: The victim downloads a form – a PDF document which contains form fields – from an attacker controlled source and fills it out on screen, for example, in order to print it. Note that there are legitimate cases where a form is obtained from a third party, while the user input should not be revealed to this party. For example, European SEPA remittance slips can be downloaded from all over the web[3] – even though they have to be manually signed to be accepted by a local bank. The form is manipulated by the attacker in such a way that it silently, without the user noticing, sends input data to the attacker's server. To the best of our knowledge, we are the first to implement such an attack, which can be carried out using the PDF *SubmitForm* action, or by reading and exfiltrating the form values using standard JavaScript functions.

Local file leakage The PDF standard defines various methods to embed external files into a document or otherwise access files on the host's file system, as documented below.

- *External streams.* Documents can contain stream objects (e.g., images) to be included from external files on disk.
- *Reference XObjects.* This features allows to a document to import content from another (external) PDF document.
- *Open Prepress Interface.* Before printing a document, local files can be defined as low-resolution placeholders.

[3]E.g., `https://www.ibancalculator.com/fileadmin/EU-Ueberweisung.pdf`

- *Forms Data Format (FDF).* Interactive form data can be stored in and auto-imported from external FDF files.
- *JavaScript functions.* The Adobe JavaScript reference enables documents to read data from or import local files.

If a malicious document managed to 1. read files from the victim's disk and 2. send them back to the attacker[4] such behaviour would arguably be critical. However, standard PDF functions can be chained together to achieve exactly this. For example, form values can be references to stream objects and every stream, on its part, can reference to an external file. Moreover, forms can be crafted to auto-submit themselves using various actions as documented in Figure 1 in section 6. Furthermore, standard JavaScript functions may lead to reading local files and leaking their contents. We give a systematic overview on this new chaining technique in terms of a directed graph containing all chains detected during our evaluation, and are the first to implement these attacks.

Credential theft In 1997, Aaron Spangler posted a vulnerability in Windows NT on the Bugtraq mailing list [34]: Any client program can trigger a connection to a rogue SMB server. If the server requests authentication, Windows will automatically try to log in with a hash of the user's credentials. Such captured NTLM hashes allow for efficient offline cracking[5] and can be re-used by applying pass-the-hash or relay attacks [20, 27] to authenticate under the user's identity. This design flaw in the Windows operating system is not solved until today[6]. Back in 1997, Spangler used a remote image to trick web browsers into making a connection to and thereby authenticate to the attacker's host. In April 2018, [33] showed that a similar attacks can be performed with malicious PDF files. They found that the target of *GoToR* and *GoToE* actions can be set to `\\\\attacker.com\\dummyfile`[7], thereby leaking credentials in the form of NTLM hashes. The issue was fixed quickly by Adobe and Foxit. We systematically evaluate this attack for other PDF viewers, and describe novel variants of this attack, for example, by using various other techniques to access a network share such as by including it as external content stream or by testing different PDF actions.

[4]Note that exfiltration does not necessarily have to occur via the network: For example, if a cloud storage service generates thumbnail images from uploaded PDF documents, the backchannel can be the rendered image itself. If a reviewer adds comments to a malicious PDF document, local files may unintentionally be included when saving, exporting or printing the document.

[5]For NTLMv2, it is estimated that cracking eight character passwords of any complexity takes around 2,5 hrs on a modern GPU [11]. Previous versions (NTLMv1, LM) are trivial to crack and can be considered as broken [24].

[6]Microsoft introduced the possibility to define "NTLM blocking" in the Windows security policy, but is has to be actively enabled by administrators. Furthermore, some ISPs block port 445, however this cannot be relied on.

[7]Note that the \ character must be escaped in PDF strings, leading to \\.

3.4 Data Manipulation

This attack class deals with the capabilities of malicious documents to write to local files on the host's file system, to silently modify form data, or to show a different content based on the application that is used to open the document.

Form modification The idea of this attack is as follows: Similar to "form data leakage" as described above, the victim obtains a harmlessly looking PDF document from an attacker controlled source, for example, a remittance slip or a tax form. The goal of the attacker is to dynamically and without knowledge of the victim change form field data. Therefore, the document is crafted in such a way that it "modifies itself", and manipulates certain form fields immediately before it is printed or saved. This could be, for example, the recipient of a wire transfer or the declarations regarding taxable income. Technically, this can be achieved using an *ImportData* action which imports form data from an external source or an embedded file, or with JavaScript included in the document. This technique can be used by an attacker to either get the victim into trouble (e.g., tax fraud suspicion) or to gain financial advantages (e.g., by adding herself as recipient of a tax refund). To the best of our knowledge, this attack has not been implemented before.

File write access As previously described, the PDF standard enables documents to submit form data to external webservers. However, technically the webserver's URL is defined using a PDF *File Specification*. This ambiguity in the standard may be interpreted by implementations in such a way that they enable documents to submit PDF form data to a local file, thereby writing to this file. Furthermore, there are various JavaScript functions which allow to write to local files on disk. If successful, this feature can be used to overwrite arbitrary files on the victim's file system and thereby purge their content. Furthermore, write access to local files may even be escalated to code execution if the attacker has write access to certain startup scripts (e.g., *autoexec.bat* under Windows, *.bashrc* under macOS and Linux). JavaScript based attacks to write to local files have previously been shown, for example, in CVE-2018-14280 and CVE-2018-14281 for Foxit Reader. We evaluate write access for a broad range of standard PDF and JavaScript functions. To the best of our knowledge, we are the first to propose the attack variant based on PDF forms that automatically submit data to a local file.

Content masking The goal of this attack is to craft a document that renders differently, depending on the applied PDF interpreter. This can be used, for example, to show different content to different reviewers, to trick content filters (AI-based machines as well as human content moderators), plagiarism detection software, or search engines, which index a different text than the one shown to users when opening the document. Content masking attacks have been shown in the past by [22, 6] who use polyglots, for example, PDF files that are also a valid JPEG images, if opened by image processing software. Recently, [23] presented “PDF mirage”, which applies font encoding to present a different *displayed* content to humans than to text exfiltration software. We propose a new approach which targets edge cases in the PDF specification, leading to different parts of the document actually being *processed* by different implementations. To achieve this, we systematically studied the PDF standard for ambiguities at the syntax and structure level as documented below.

- *Stream confusion.* It is unclear how content streams are parsed if their *Length* value does not match the offset of the *endstream* marker, or if syntax errors are introduced.
- *Object confusion.* An object can overlay another object. The second object may not be processed if it has a duplicate object number, if it is not listed in the *XRef* table, or if other structural syntax errors are introduced.
- *Document confusion.* A PDF file can contain yet another document (e.g., as embedded file), multiple *XRef* tables, etc., resulting in ambiguities on the structural level.
- *PDF confusion.* Objects before the PDF header or after an *EOF* marker may be processed by implementations, introducing ambiguities in the outer document structure.

There are numerous variants of the four test classes mentioned above, resulting in an overall of 94 hand-crafted edge cases which we evaluate in section 4. Note that some of those edge cases have already been discussed in [39, 1, 10] in the context of what can go wrong when parsing a PDF document.

3.5 Code Execution

The goal of this attack is to execute attacker controlled code. This can be achieved by silently launching an executable file, embedded within the document, to infect the host with malware.

Launch action The PDF specification defines the *Launch* action, which allows documents to launch arbitrary applications. The file to be launched can either be specified by a local path, a URL, or a file embedded within the PDF document itself. The standard does not provide any security considerations regarding this obviously dangerous feature, it even specifies how to pass command line parameters to the launched application. Therefore, it can be said that PDF offers “command execution by design" – if the standard is straightforwardly implemented. An example of a malicious document which contains an embedded executable file (*evil.exe*) that is launched once the document is opened (*OpenAction*) is depicted in Listing 2.

```
1 0 obj
<< /Type /Catalog /Names <<
  /EmbeddedFiles << /Names [(evil.exe) 2 0 R] >> >>
  /OpenAction << /S /Launch /F (evil.exe) >>
>>
endobj

2 0 obj
<< /Type /EmbeddedFile /Length 1337 >>
stream
[executable code]
endstream
```

Listing 2: Example PDF document to launch an embedded executable file.

This danger of *Launch* actions is well-known (see, e.g., [9], and modern PDF viewers should warn to user before executing potentially malicious files – or stop supporting this insecure feature at all. We extend the analysis of [9] to a comprehensive set of 28 modern PDF implementations.

4 Evaluation

To evaluate the attacks introduced in section 3, we tested them on 28 popular PDF processing applications that were assembled from public software directories for the major platforms (Windows, Linux, macOS, and Web).[8] If a "viewer" and an "editor" version was available, we tested both. All applications were tested in the default settings, neither relaxing nor hardening their security policies. Evaluation results are depicted in Table 1.

4.1 Denial of Service

In the following, we discuss the results for DoS attacks. Because of the large number of test cases, a fully detailed evaluation is given in Table 3 in the appendix. We classify an application as vulnerable if it either hangs (usually consuming vast amounts of CPU or memory), or if the program crashes. A controlled program termination (i.e., raising an exception before closing) is not considered as a vulnerability.

[8]Note that some PDF applications are available for multiple platforms. In such cases we limited our tests to the platform with the highest market share.

Attack Category			DoS		Invasion of Privacy			Disclosure			Manipulation			RCE
Application	Version		Infinite loop	Deflate bomb	URL invocation	Evitable metadata	Revision recovery	Form data leakage	Local file leakage	Credential theft	Form modification	File write access	Content masking	Code execution
Acrobat Reader DC	(2019.008.20081)	Windows	●	●	●	–	–	○	○	○	●	○	○	○
Foxit Reader	(9.2.0.9297)	Windows	●	●	○	–	–	○	○	●	○	○	○	○
PDF-XChange Viewer	(2.5.322.9)	Windows	●	●	●	–	–	●	●	●	●	○	●	○
Perfect PDF Reader	(8.0.3.5)	Windows	●	●	●	–	–	●	○	●	○	○	○	○
PDF Studio Viewer	(2018.1.0)	Windows	○	●	●	–	–	●	◐	○	○	○	○	●
Nitro Reader	(5.5.9.2)	Windows	●	●	●	–	–	●	○	●	○	○	●	●
Acrobat Pro DC	(2017.011.30127)	Windows	●	●	●	○	◐	○	○	○	●	○	○	○
Foxit PhantomPDF	(9.5.0.20723)	Windows	●	●	○	●	○	○	○	●	○	○	○	○
PDF-XChange Editor	(7.0.326.1)	Windows	●	●	●	○	○	●	●	●	●	●	○	○
Perfect PDF Premium	(10.0.0.1)	Windows	●	●	●	●	○	●	○	●	○	○	○	○
PDF Studio Pro	(12.0.7)	Windows	○	●	●	○	○	●	◐	○	○	○	○	●
Nitro Pro	(12.2.0.228)	Windows	●	○	●	●	○	●	○	●	○	○	●	●
Nuance Power PDF	(3.0.0.17)	Windows	●	●	●	●	○	○	●	●	○	○	●	○
iSkysoft PDF Editor	(6.4.2.3521)	Windows	●	○	○	●	◐	○	○	○	○	○	○	○
Master PDF Editor	(5.1.36)	Windows	●	○	●	○	○	●	◐	●	○	●	○	○
Soda PDF Desktop	(11.0.16.2797)	Windows	●	●	●	●	○	○	○	●	○	○	●	○
PDF Architect	(7.0.23.3193)	Windows	●	●	●	●	○	○	○	●	○	○	●	○
PDFelement	(6.8.0.3523)	Windows	●	○	○	●	◐	○	○	○	○	○	○	○
Preview	(10.0.944.4)	Mac	●	●	○	–	–	○	○	○	○	○	○	○
Skim	(1.4.37)	Mac	●	●	○	–	–	○	○	○	○	○	○	○
Evince	(3.2.11)	Linux	○	●	○	–	–	○	○	○	○	○	○	◐
Okular	(0.26.1)	Linux	●	●	○	–	–	○	○	○	○	○	○	◐
MuPDF	(1.14.0)	Linux	●	○	○	–	–	○	○	○	○	○	○	◐
Chrome	(70.0.3538.67)	Web	◐	●	●	–	–	●	○	○	○	○	○	○
Firefox	(66.0.2)	Web	◐	●	●	–	–	○	○	○	○	○	○	○
Safari	(11.0.3)	Web	○	○	○	–	–	○	○	○	○	○	○	○
Opera	(57.0.3098.106)	Web	◐	◐	●	–	–	●	○	○	○	○	○	○
Edge	(42.17134.1.0)	Web	○	○	○	–	–	○	○	○	○	○	○	○

● Application vulnerable   ◐ Vulnerability limited   ○ Not vulnerable   – No editing

Table 1: Out of 28 tested PDF applications, 26 are vulnerable to at least one attack.

Infinite loop Each of the tested applications running natively on Windows, macOS, or Linux, except three viewers, was vulnerable to at least one attack variant and could be tricked into an endless loop. It is noteworthy, that CVE-2007-0104 still works in 6 applications until today. Our novel attack variants, such as *GoTo* loops (9 vulnerable), Action loops (9 vulnerable), Outline loops (9 vulnerable) and

JavaScript (13 vulnerable) cause endless loops in various PDF interpreters. The impact is either a crash of the program, or the application becoming completely unresponsive, often combined with a high consumption of CPU time. Browser based PDF viewers instead perform much better: We observed that for Chrome, Firefox, and Opera only the current tab gets stuck in an endless loop and becomes unresponsive, which is why we classified the vulnerability as "limited" here. We assume this is because modern browsers sandbox each tab and enforce resource limits, thereby restricting the impact of, for example, a malicious or runaway website.

Deflate bomb To evaluate the impact of compression bombs, we crafted a valid PDF file containing a long string of 10 GB of repeated characters, "`AAA`...", within a compressed content stream. To display this string to the user, a PDF processing application must first decompress it using the *Deflate* algorithm – with an amplification factor of 1023:1.[9] The attack resulted in memory exhaustion in 20 applications, of which three applications crashed after a short period of time. In various cases, the operating system slowed down noticeable or became completely unresponsive. In contrast to attacks based on infinite loops, even browsers such as Chrome and Firefox were fully affected, while in Opera only the current tab became unresponsive. The remaining seven PDF applications did refuse to decompress the whole stream, but instead aborted decompression after a decent amount of time – probably after a watchdog limit was reached. It is noteworthy that we did not even have to actually *open* the malicious document on Windows and Linux to cause DoS to the operating system. Both, Windows File Explorer and Gnome Nautilus file manager try to preview the document if the containing directory is opened, and thereby process its content resulting in resource exhaustion. MacOS (Finder) was not vulnerable, because it stopped thumbnail generation, probably after a resource limit was hit. Even though not tested for ethical reasons, applications processing PDF files on the server-side are likely to be affected too. For example, Evince and Okular, which are both vulnerable, are based on Poppler[10], a popular PDF library used by various cloud storage providers to generate preview images of uploaded PDF documents.

4.2 Invasion of Privacy

URL invocation To evaluate if malicious documents can enforce PDF applications to trigger a connection to an attacker controlled server, we combined various PDF

[9]This is the maximum compression ratio that can be achieved with *Deflate*. However, the PDF file size can be drastically reduced by applying multiple *Deflate* filters to the stream, resulting in a compression ratio of 1,8470,265:1. Therefore, a 10 MB file on disk is decompressed to 10 GB in memory.

[10]See `https://poppler.freedesktop.org/`.

features with techniques to automatically call them once the document was opened. The results for auto-triggered PDF actions resulting in URL invocation are as follows: *URI* action (9 vulnerable), *GoToR* (1 vulnerable), *Launch* (6 vulnerable), and *SubmitForm* (11 vulnerable). For 7 applications, we could use standard JavaScript functions to invoke a connection. In one viewer we could set a URL as the external content stream of an image, which was loaded from the attacker's server. In two viewers we were able to inject a subset of XHTML, leading to HTML tags being being processed which triggered a remote connection. Altogether, 17 PDF applications could be tricked into (silently) invoking a connection to our server, once a malicious document was opened by the user. It can be concluded that it is relatively easy to craft a PDF document which reports back to the author (or a third party) when the document is opened, in a majority of the tested applications.

Evitable metadata PDF allows additional data such as the author's name, creation date and creator software to be embedded in documents. To identify which information is included by modern applications, we created a simply "Hello World" document with each tested PDF editor and spotted the metadata in the saved file, which can either be found in the *document information dictionary* or within a *metadata stream*. The results are as follows.

- All editors stored the date of creation and modification.
- All editors stored the creator software (version number).
- Eight editors stored the author's name, derived from the name of the currently (at creation time) logged in user.

We classify a PDF editor as "vulnerable", if it silently stores the author's name (i.e., the username) in the default settings. This was the case for eight out of twelve PDF applications with document creation/editing functionality.

We also performed a large-scale evaluation, of 294.586 PDF files downloaded from the Internet[11] of which 173.112 (58%) contained an author name. Of course, we cannot make any statement if this information was included on purpose or by accident. The single largest creator software of documents that contain an author name was Microsoft Office with 64.167 files.

[11]We obtained the dataset by crawling the Cisco Umbrella 1 Million list of domains, see `https://s3-us-west-1.amazonaws.com/umbrella-static/index.html`.

Revision recovery To test if sensitive information can be recovered from a document redacted by a PDF editor, we used two PDF files – one containing selectable text, the other containing a scanned document (i.e., an image)[12]. We applied the PDF editor's "redact" function to draw a black rectangle over parts of the document as well as the "delete" function to remove the text or image. In all tested PDF editors, the "redaction" feature was found to be secure, because the actual content of the text or image object was modified, thereby overwriting potentially sensitive content in the file. However, we determined potential security issues in Acrobat Pro and and two other PDF editors, whereby we deleted the content (text or image). The removed content is not displayed anymore, but it is still contained in the file and can be extracted. We do classify this vulnerability as "limited" in our evaluation, because the "delete" function is not explicitly promoted as a secure feature, even though users may misinterpret it as such. To conclude, redaction tools in PDF viewers can be considered as well-developed these days. The only identified risk is caused by removing sensitive information without using the *redact* feature of the PDF editors. This approach does not provide the same security level and should be avoided.

4.3 Information Disclosure

Form data leakage To test if form data can be leaked silently, without the user knowing, we modified the standard U.S. individual tax return form 1040[13] to send all user input to our webserver once the document is either printed or closed. This can be done by combining the *DP* ("did print") and *PC* ("page closed") events with a *SubmitForm* action or JavaScript. We classify the attack as successful if a PDF application passes filled-in form data without the user being made aware of it (i.e., no warning message or confirmation dialog displayed). Nine applications are vulnerable to this attack, using forms that auto-submit themselves. For two additional applications, we were able to use JavaScript to access form data and silently exfiltrate it to our server. Nine applications did ask the user before sending the data, which we consider as sane behaviour. Another eight PDF interpreters (e.g., on macOS and Linux) did not support the feature of submitting PDF form data at all.

Local file leakage Even though part of the standard, only two applications support the feature of external streams. For both applications we were able to craft a document which embeds arbitrary files on disk into the document and silently leaks them to an external server using both, auto-submitting forms and JavaScript.

[12]We used the scan of a document from WWI, describing cipher techniques, which was recently declassified by the CIA and can be downloaded from: `https://www.cia.gov/library/readingroom/docs/Secret-writing-document-one.pdf`.

[13]Available for download from `https://www.irs.gov/pub/irs-pdf/f1040.pdf`.

Exfiltration happens in the background once the document is opened, without the user noticing and without any visible changes to the document. For another three applications, we were able to include and automatically leak the contents of FDF files and XML-based XFDF files (using the *ImportData* action or the *ImportFDF* JavaScript function). We classify this vulnerability as limited, because it restricted by file type – yet it should be clear that such behaviour is not desired either. Note that this attack is different from "form data leakage" as mentioned before, because – while FDF/XFDF files usually contain PDF form data – this attack results in the contents of *external* FDF/XFDF files from disk being leaked, which may be completely unrelated to the form data of the currently opened (malicious) document. For one viewer, in addition, we were able to use standard JavaScript functions to access arbitrary files and the leak them.

Credential theft We used Responder[14] as a rogue authentication server to obtain the client's NTLM hashes. We were able to leak the hashes of NTLM credentials to our server without the user noticing or being asked for confirmation to open a connection to the rogue network share on 12 out of the 18 Windows based PDF viewers. Note that, by design, only applications running on Windows are affected. We used a mixture of techniques to accomplish this goal: external streams, standard PDF actions, as well as JavaScript. Various of the vulnerable readers where affected by to multiple test cases. It is interesting to note that, while Foxit fixed this issue in 2018 for PhantomPDF/Reader, we could identity bypasses using four different techniques. This is because – apparently – accessing a share invocation via *GotoR* actions (as documented in the original exploit) was prohibited, however, using other action types, such as auto-printing a file on network share, we were again able to enforce NTLM hashes being leaked. Of course, it is up to the configuration of the victim's setup (i.e., password strength, security policy, and Windows version) if efficient cracking or relay attacks are actually feasible.

4.4 Data Manipulation

Form modification To test the feasibility of crafting PDF documents that silently manipulate their own form data, we – once again – modified the U.S. tax return form 1040. We added an *ImportData* action that changes the refund account number to the attacker's account number once the document is printed.[15] We used the *WP* ("will print") event for this purpose. Unfortunately, from an attacker's point of view, none of the tested applications supports importing form data from an *embedded file* within the document itself – or from an external URL. By using standard PDF

[14]See `https://github.com/SpiderLabs/Responder`.

[15]It must be noted that, in practice, this attack does no only have a technical component. It will only work if the attacker's bank accepts the deposit, see `https://www.irs.gov/faqs/irs-procedures/refund-inquiries/refund-inquiries-18`

JavaScript functions (`getAnnots()[i].contents`), we were however able to modify PDF form data in 4 applications. JavaScript also allowed us to temporarily store the original user data and undo our manipulation immediately after the document had been printed, using the *DP* ("did print") event, and to enforce that these modifications are only performed until a certain date, thereby making it more difficult to reproduce the manipulation.

File write access Only three applications allowed to submit form data to a local file. While two applications explicitly ask the user before writing to disk, one PDF editor silently writes to or overwrites arbitrary files with attacker controlled content by auto-submitting the form data to a PDF *File Specification.* We also tested six standard PDF JavaScript functions to write to disk. The `extractPages()` function allowed us to write data to arbitrary locations on disk in one application. The other applications did not support writing files with JavaScript at all, asked the user for confirmation, or showed a "Save as" dialog, instead of automatically writing the file to a given location.

Content masking We define an application as vulnerable if we can create a document that displays certain text in this, and only in this, application, while a completely different text is displayed in all other tested PDF viewers – with the exception of two applications utilizing the same underlying PDF interpreter (e.g., Evince/Okular are both based on Poppler). Furthermore, if a vendor produces a "viewer" and an "editor" version of an application, both may also display the same text. Of our 94 hand-crafted edge cases, 63 rendered differently when opened in different applications. Full details are given in Table 4 in the appendix. For three PDF interpreter engines (six applications), we found a case where certain text was displayed *only* in this interpreter. For other PDF interpreters, we could not find edge cases resulting in a *unique* appearance (i.e., no other interpreter displaying the same text), therefore we did not classify them as vulnerable. It must however be noted that test cases can potentially be chained together, which may result in getting more applications to render unique content. This challenge is considered as future work. Another interesting use of this technique would be fingerprinting PDF interpreters applied in web applications to process or preview documents, based on the rendered result when uploading a PDF file.

4.5 Code Execution

Launch action In theory, by chaining PDF standard features, an attacker can easily get code execution *by design.* We combined a *LaunchAction* with an *OpenAction* to achieve this goal and launch an executable file. Surprisingly, this worked out of the box on 4 applications. The *.exe* file was launched without any confirmation dialog being displayed to the user. However, it must be said that these 4 applications originated from only 2 different vendors. Each of them probably use the same code base for both, their *viewer* and *editor* product version. The other tested applications asked the user for confirmation (8 viewers) before executing the file, denied to launch executable files (Acrobat Reader/Pro)[16], or did not support the *LaunchAction* at all in the default settings (11 viewers). The remaining 3 Linux based viewers (Evince, Okular, MuPDF) use *xdg-open*[17] to handle the file to be launched, thereby delegating the security decision to a third-party application. On our Debian GNU/Linux test system, this resulted in code execution with minimal user interaction: by referencing an *.exe* from a *Link* annotation, the file was executed with `/usr/bin/mono`, an emulator for .NET executables, if the user clicked *somewhere* into the document.[18] We do classify these vulnerabilities as "limited" because – even though no confirmation dialog is presented to the user – the exploit is not *fully* automated. One additional viewer – which we initially tested – was vulnerable too, however the latest version had removed support for the *Launch* action. Finally, it must be said that, even if a confirmation dialog is presented, targeted attackers may apply social engineering techniques to trick the victim into launching the file.

Because the *Launch* action can be considered as a dangerous feature, we conducted a large-scale evaluation of 294.586 PDF documents downloaded from the Internet, in order to research if there are any legitimate use cases at all. Of those documents, only 532 files (0.18%) contained a *Launch* action. While none of the files was classified as malicious according to the VirusTotal database[19], we conclude that the *Launch* action is rarely used in the wild and its support should be removed by PDF implementations as well as the standard.

5 Exploits

In this section, we briefly describe the proof-of-concept exploits provided with this report and the used file naming convention.

5.1 Directory Structure

The general naming convention for exploit files is `{nn}-[category]-{nn}-[attack].pdf` where *category* is one of the five defined attack categories and *attack* is the (sub)attack

[16]Note that Adobe products use a blacklist of potentially "dangerous" file extensions. However, various bypasses have been identified in the past [32].

[17]See `https://www.freedesktop.org/wiki/Software/xdg-utils/`.

[18]Readers of may ask themselves: How often did I click in this document to jump to a certain section? Would we anticipate this can lead to code execution?

[19]See `https://www.virustotal.com/`.

variants described in section 3 and evaluated in in section 4. This results in the following proof-of-concept files:

- `01-dos-01-infinite-loop.pdf`
- `01-dos-02-deflate-bomb.pdf`

- `02-privacy-01-url-invocation.pdf`
- `02-privacy-02-evitable-metadata.pdf`
- `02-privacy-03-revision-recovery.pdf`

- `03-disclosure-01-form-data-leakage.pdf`
- `03-disclosure-02-local-file-leakage.pdf`
- `03-disclosure-03-credential-theft.pdf`

- `04-manipulation-01-form-modification.pdf`
- `04-manipulation-02-file-write-access.pd`
- `04-manipulation-03-content-masking.pdf`

- `05-rce-01-code-execution.pdf`

Note that not every PDF application is vulnerable to each attack. Furthermore, the actual exploits differ for each PDF viewer, therefore every application has its own directory.

6 Countermeasures

In this section we discuss short-term mitigations as well as more generic, in-depth countermeasures to be considered by implementations and future versions of the PDF standard.

6.1 Towards an Unambiguous Specification

To counter infinite loops, constructs that can lead to cycles or recursion, such as self-referencing objects, must be prohibited in implementations (e.g., by remembering their path) and ambiguous formulations should be removed from the standard. A clearly stated specification would also help to prevent content masking attacks. In practice, this is not an easy task as it would require a formal model of the PDF standard, in order to proof that the model is cycle free and that a certain document can only be processed in one single way. Furthermore, it must be noted that an unambiguous PDF specification would only protect the document structure, not embedded data formats such as XML, JavaScript, Flash, etc.

6.2 Resource Limitation and Sandboxing

To counter compression bombs, [29] propose to halt decompression once the size of the decompressed data exceeds an upper limit. This strategy should be applied by PDF processing applications. It must however be noted that a single document can contain thousands of streams to be processed in a row. In general, a good approach is limiting the resources to be consumed by a single document, by sandboxing it – similar to a tab in a modern web browser, thereby preventing malicious documents to affect the whole application or even the operating system.

6.3 Implementing Privacy by Default

PDF editors should not include excessive metadata such as usernames in the default settings. Furthermore, all editing functions (redaction, modification, and deletion of elements) should be performed on the actual object to prevent a third party from recovering previous versions of the document. Such best practices regarding metadata and text redaction should not only be applied by PDF editors, but by all applications that allow to export content to PDF (e.g., office suites).

6.4 Removing or Restricting JavaScript

JavaScript support in PDF applications is extremely varied. The absence of a sound test suite to accompany the standard makes it difficult for developers to create compliant and robust implementations. In addition the great disparity between PDF viewers regarding their feature support complicates the effective utilization of JavaScript by authors of PDF documents. While we could observer some viewers to borrow a stable JavaScript engine from other projects, such as SpiderMonkey or V8, multiple viewers provide very unstable homebrewn solutions which can be crashed with ease. Unrelated to the used engine, many viewers implement obscure

JavaScript API functions without providing public documentation. Neither their purpose nor resistance to exploitation is clear.

Given that PDF is supposed to be a format for portable documents, the need to embed a full programming language is debatable. Many legitimate use cases of JavaScript in PDF, such as input validation of form fields, can be covered without a programming language, as established and proven in HTML5[20]. Any scenario exceeding the non-programmatic features of PDF should be considered to be implemented as a web application instead of a PDF document, given that JavaScript in modern web browsers is well researched and robustly implemented.

6.5 Identification of Dangerous Paths

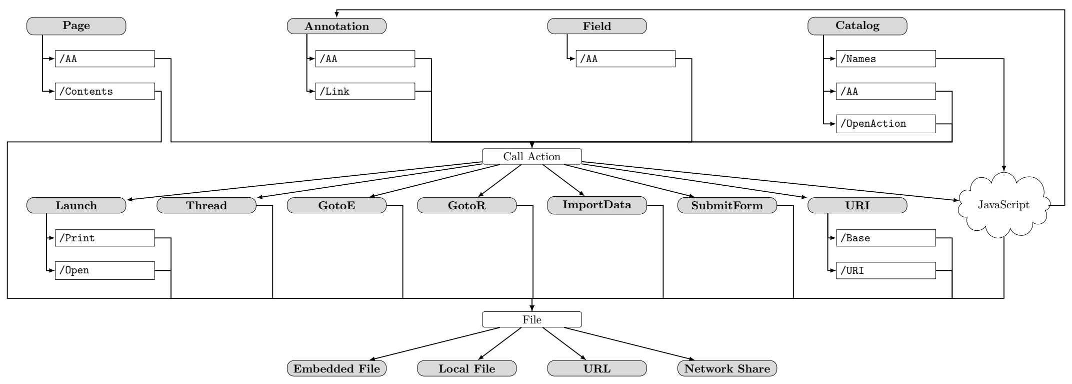

Figure 1: **Dangerous paths identified by studying the PDF specification.** There are different special PDF objects (*Catalog*, *Page*, ...) defined that allow to call various actions (*Launch*, *Thread*, ...) which can access a file handle. The handles can be used to read data (e.g., local files) and to leak it (e.g., via URL).

Most of the attacks presented in this report surprisingly have the same root cause. To read data from the victim's computer as well as to exfiltrate information, and even to execute embedded files, the attacker needs access to a file handle (i.e., a PDF *File Specification*). We systematically analyzed the PDF standard for methods to access such a handle. The results are depicted in Figure 1. In the upper row, we identified PDF objects which allow to call different actions (*Page*, *Annotations*, ...). For calling them, most objects offers multiple alternatives. The *Catalog* object, for example, defines the *OpenAction* or additional actions (*AA*) keys. Each key can be used to launch any number of PDF actions, which are depicted in the lower part (*Launch*, *Thread*, ...). Some actions provide multiple techniques to access a file handle. For example, the action *URI* can use the keys *Base* or *URI* for this purpose. Once access

[20]See `https://html.spec.whatwg.org/multipage/input.html#input-impl-notes`.

is provided, the file handle can be used to read data from the victim (embedded file, local file) or to exfiltrate data (URL, network share).

To build exploits, actions can be chained together. For example, an attacker can craft a document which first imports data from a local file using the import actions and the sends the content to the attacker's server using the *SubmitForm* action. Note that any action can be define an arbitrary number of *Next* actions to be performed, thereby allowing actions to call each other.

In addition to native PDF features, JavaScript can be used within documents, opening a new area for attacks. For example, with JavaScript, new annotations can be created, which can have actions, once again leading to a file handle.

Our attacks took a path from the top the file handle. If the path was not blocked or required user consent, the attack was successful. Many viewer applications blocked particular paths, but failed to block all of them. Two positive examples for blocking dangerous paths are Safari and Edge. These application blocked all but one path: *Annotation⇒Link⇒URI⇒URL*. In addition, this path required user interaction by actively clicking on the link. This example illustrates that how a secure PDF application should work. We would like to see more applications, restricting the dangerous paths systematically (e.g., by removing them completely or by asking the user for consent) in order to prevent data exfiltration attacks.

Launch	*Thread*	*GoToE*	*GoToR*	*SubmitForm*	*ImportData*	*URI*
532 (0.18%)	4416 (1.49%)	0 (0.00%)	693 (0.23%)	64 (0.02%)	0 (0.00%)	46612 15.82%)

Table 2: PDF actions in 294,586 publicly available documents.

As part of this report, we conducted a large-scale evaluation of 294,586 publicly available PDF documents. The results on how many documents contain a certain action is depicted in Table 2. As one can seem the only action-based PDF feature that is widely in practice is the *URI* action, which can be restricted to a *Link Annotation*. Insecure features instead are rarely used in real-world PDF documents. Therefore, it can be concluded that PDF viewers should drop support for potentially dangerous features such as the *Launch* action or at least not enable them in the default settings.

7 Conclusion

PDF is more than a simple document format. Each standard compatible PDF viewer must support a large set of additional features. While PDF exploitation caused by implementation bugs, such as buffer overflow based code execution, has been a long-standing research area with many important results, this report shows that even native PDF features can lead to minor and major security vulnerabilities exploited by malicious PDF documents.

References

[1] caradoc: A pragmatic approach to pdf parsing and validation.

[2] Adobe Systems. Acrobat JavaScript Scripting Guide, 2005.

[3] Adobe Systems. Adobe Supplement to the ISO 32000, BaseVersion: 1.7, ExtensionLevel: 3, 2008.

[4] Adobe Systems. XMP Specification Part 1, 2012.

[5] National Security Agency. Hidden Data and Metadata in Adobe PDF Files: Publication Risks and Countermeasures, 2008.

[6] A. Albertini. This PDF is a JPEG; or, This Proof of Concept is a Picture of Cats. *PoC 11 GTFO 0x03*, 2014.

[7] C. Alonso, E. Rando, F. Oca, and A. Guzmán. Disclosing Private Information from Metadata, Hidden Info and Lost Data, 2008.

[8] P. Bieringer. Decompression Bomb Vulnerabilities, 2001.

[9] A. Blonce, E. Filiol, and L. Frayssignes. Portable Document Format Security Analysis and Malware Threats. *BlackHat Europe*, 2008.

[10] C. Carmony, X. Hu, H. Yin, V. Bhaskar, and M. Zhang. Extract me if you can: Abusing pdf parsers in malware detectors. In *NDSS*. The Internet Society, 2016.

[11] T. Claburn. Use an 8-char Windows NTLM password?, February 2019.

[12] A. Denied. DFS Issue 55, 1996.

[13] P. Deutsch. DEFLATE Compressed Data Format Specification, 1996.

[14] E. Ellingsen. ZIP File Quine, 2005.

[15] D. Fifield. A better zip bomb (website), 2019.

[16] K. Foss. Washington Post's scanned-to-PDF Sniper Letter More Revealing Than Intended, 2002.

[17] J. Gajek. Macro malware: Dissecting a malicious word document. *Network Security*, 2017(5):8–13, 2017.

[18] S. Garfinkel. Leaking Sensitive Information in Complex Document Files - -and How to Prevent It. *IEEE Security & Privacy*, 12(1):20–27, 2013.

[19] V. Hamon. Malicious uri resolving in pdf documents. *Journal of Computer Virology and Hacking Techniques*, 9(2):65–76, 2013.

[20] Chris Hummel. Why Crack When You Can Pass The Hash. *SANS Institute InfoSec Reading Room*, 21, 2009.

[21] B. Krebs. Document security 101, 2005.

[22] J. Magazinius, B. Rios, and A. Sabelfeld. Polyglots: Crossing Origins by Crossing Formats. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, pages 753–764. ACM, 2013.

[23] I. Markwood, D. Shen, Y. Liu, and Z. Lu. PDF Mirage: Content Masking Attack Against Information-Based Online Services. In *26th USENIX Security Symposium (USENIX Security 17), (Vancouver, BC)*, pages 833–847, 2017.

[24] M. Marlinspike. Divide and conquer: Cracking ms-chapv2 with a 100% success rate. *CloudCracker [online]*, 29, 2012.

[25] K. McCarthy. That classified us military report's secrets in full, 2005.

[26] A. Nusca. Facebook settlement revealed via poor pdf redaction, 2009.

[27] N Ochoa. Pass-the-hash toolkit-docs & info, 2008.

[28] T. Parker. How to do (not so simple) form calculations, July 2006.

[29] G. Pellegrino, D. Balzarotti, S. Winter, and N. Suri. In the Compression Hornet's Nest: A Security Study of Data Compression in Network Services. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 801–816, 2015.

[30] C. Pesce. Document metadata, the silent killer...

[31] D. Poddebniak, C. Dresen, J. Müller, F. Ising, S. Schinzel, S. Friedberger, J. Somorovsky, and J. Schwenk. Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 549–566, Baltimore, MD, 2018. USENIX Association.

[32] F. Raynal, G. Delugré, and D. Aumaitre. Malicious Origami in PDF. *Journal in Computer Virology*, 6(4):289–315, 2010.

[33] Check Point Research. Ntlm credentials theft via pdf files, April 2018.

[34] Aaron Spangler. Winnt/win95 automatic authentication vulnerability (ie bug #4), March 1997.

[35] Adobe Systems. Displaying 3d models in pdfs, June 2017.

[36] Adobe Systems. Applying actions and scripts to pdfs, April 2019.

[37] Adobe Systems. How to fill in pdf forms, April 2019.

[38] Adobe Systems. Starting a pdf review, April 2019.

[39] J. Wolf. Omg wtf pdf. In *27th Chaos Communication Congress (27C3)*, 2010.

A Appendix

		Pages loop				GoTo loop				Action loop			Calc		Outline		ObjStm	JavaScript			Deflate
	Application	A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	C3	D1	D3	E2	E3	F1	G1	G2	G3	DB
Windows	Acrobat Reader DC	○	○	○	○	○	○	○	💣	○	○	🕘	○	○	○	○	🕘	🕘	💣	🕘	🕘
	Foxit Reader	○	○	🕘	○	○	💣	💣	💣	○	💣	💣	○	○	○	○	○	🕘	○	🕘	🕘
	PDF-XChange Viewer	💣	💣	💣	💣	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	🕘
	Perfect PDF Reader	○	○	💣	💣	○	○	○	○	○	○	○	○	○	○	🕘	○	○	○	○	🕘
	PDF Studio Viewer	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	🕘
	Nitro Reader	○	○	○	○	💣	○	○	○	○	○	○	○	○	💣	💣	○	🕘	🕘	🕘	○
	Acrobat Pro DC	○	○	○	○	○	○	○	💣	○	○	○	○	○	○	○	🕘	🕘	💣	🕘	🕘
	Foxit PhantomPDF	○	○	🕘	○	💣	💣	💣	💣	○	💣	💣	○	○	○	○	○	🕘	○	🕘	🕘
	PDF-XChange Editor	○	○	○	○	○	🕘	🕘	○	○	💣	💣	○	○	○	○	○	🕘	○	🕘	🕘
	Perfect PDF Premium	○	○	💣	💣	○	○	○	○	○	○	○	○	🕘	○	🕘	○	○	○	○	🕘
	PDF Studio Pro	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	🕘
	Nitro Pro	○	○	○	○	💣	○	○	○	○	○	○	○	○	💣	💣	○	🕘	🕘	🕘	○
	Nuance Power PDF	○	○	💣	💣	💣	○	○	💣	💣	💣	💣	💣	○	💣	💣	○	🕘	○	🕘	💣
	iSkysoft PDF Editor	○	○	○	○	○	○	○	○	○	○	○	○	○	🕘	🕘	○	○	○	○	○
	Master PDF Editor	○	○	○	○	💣	○	○	○	○	○	○	○	○	○	○	○	🕘	○	🕘	○
	Soda PDF Desktop	○	○	○	○	○	○	○	○	○	💣	💣	○	○	💣	💣	○	🕘	💣	💣	🕘
	PDF Architect	○	○	○	○	○	○	○	○	○	💣	💣	○	○	💣	💣	○	🕘	💣	💣	🕘
	PDFelement	○	○	○	○	○	○	○	○	○	○	○	○	○	💣	🕘	○	○	○	○	○
Mac	Preview	○	○	○	○	○	○	○	○	💣	💣	💣	○	○	○	○	○	○	○	○	🕘
	Skim	○	○	○	○	○	○	○	○	💣	💣	💣	○	○	○	○	○	○	○	○	🕘
Linux	Evince	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	💣
	Okular	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	🕘	○	🕘	💣
	MuPDF	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	🕘	🕘	🕘	○
Web	Chrome	○	○	○	○	(🕘)	○	○	○	○	○	(🕘)	○	○	○	○	○	(🕘)	○	○	🕘
	Firefox	○	○	○	(🕘)	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	🕘
	Safari	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
	Opera	○	○	○	○	(🕘)	○	○	○	○	○	(🕘)	○	○	○	○	○	(🕘)	○	○	(🕘)
	Edge	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○

💣 Application crashes 🕘 Applications hangs (🕘) Only current tab hangs ○ Application not vulnerable

Table 3: Detailed results for the DoS class of attacks.

Application	A1	A3	A4	A5	B1	C1	C2	C3	C4	C6	C7	C8	CX	D1	D2	D4	E3	E4	F1	F3	G1	G3	H2	H3	H5	H6
Acrobat Reader/Pro	2	2	1	1	1	–	1	1	–	1	1	–	1	2	2	2	2	1	2	2	–	–	–	–	–	–
Foxit Reader	2	2	2	1	2	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	1	1	1	1	1	1
Foxit PhantomPDF	2	2	2	1	2	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	1	1	1	1	1	1
PDF-XChange Viewer	2	2	2	2	2	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	1	1	2	1	2	1
PDF-XChange Editor	2	2	2	1	2	2	1	1	2	1	1	2	1	2	2	2	2	–	2	2	2	2	2	1	2	1
Nitro Reader/Pro	2	2	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	1	2	2	1	1	–	1	–	1
Nuance Power PDF	2	2	1	1	1	1	1	1	1	1	1	1	1	2	2	2	1	1	1	2	1	1	–	2	–	2
Soda PDF Desktop	2	2	2	1	2	2	1	1	2	1	1	2	1	2	2	2	2	–	1	1	1	1	1	1	1	1
PDF Architect	2	2	2	1	2	2	1	1	2	1	1	2	1	2	2	2	2	–	1	1	1	1	1	1	1	1
Poppler (Evince/Okular)	2	2	2	1	2	1	–	–	1	–	–	1	–	2	2	2	2	1	2	2	–	–	–	–	–	–
Chrome	2	2	2	1	2	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	1	1	1	1	1	1
Firefox	2	2	2	1	2	1	1	1	1	1	1	1	1	2	2	2	2	2	1	2	–	–	1	–	1	–
Opera	2	2	2	1	2	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	1	1	1	1	1	1
Perfect PDF Reader	1	1	2	–	1	1	1	2	1	1	2	1	1	1	1	1	1	1	2	2	1	2	1	1	1	1
Perfect PDF Premium	1	1	2	1	2	1	1	2	1	1	2	1	1	1	1	1	2	1	2	2	1	2	1	1	1	1
PDF Studio Viewer/Pro	1	2	2	1	2	2	–	–	2	–	–	2	–	2	2	2	2	–	2	2	1	1	1	1	1	1
iSkysoft PDF Editor	1	1	2	1	2	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	1	1	1	1	1	1
Master PDF Editor	1	1	2	1	2	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	1	1	1	1	1	1
PDFelement	1	1	2	1	2	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	1	1	1	1	1	1
Preview	1	2	2	1	2	1	–	–	1	–	–	1	–	2	2	2	2	1	2	2	–	–	–	–	–	–
Skim	1	2	2	1	2	1	–	–	1	–	–	1	–	2	2	2	2	1	2	2	–	–	–	–	–	–
MuPDF	1	2	2	1	2	2	1	1	2	1	1	2	1	2	2	2	2	2	2	2	1	1	2	1	2	1
Safari	1	2	2	1	2	1	–	–	1	–	–	1	–	2	2	2	2	1	2	2	–	–	–	–	–	–
Edge	1	1	2	–	2	–	–	–	–	–	–	–	–	2	2	2	2	2	2	2	2	2	–	–	–	–

Application	I3	J1	K1	K4	K5	K6	K7	K8	M3	M4	N1	N2	N3	N4	N5	P1	P3	P4	P6	P7	P8	P9	PX	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	QX
Acrobat Reader/Pro	2	–	–	1	–	2	2	2	1	2	2	2	1	1	1	–	–	–	2	2	2	–	–	–	–	–	–	2	2	–	–	–	–
Foxit Reader	1	1	1	2	1	2	2	2	1	2	2	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
Foxit PhantomPDF	1	1	1	2	1	2	2	2	1	2	2	2	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
PDF-XChange Viewer	2	1	1	2	1	2	2	2	2	2	2	1	2	2	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
PDF-XChange Editor	2	1	2	2	1	2	2	2	1	2	2	1	2	2	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Nitro Reader/Pro	2	1	2	1	1	2	2	2	1	1	2	1	2	2	2	1	2	2	2	2	2	–	–	–	–	2	2	2	2	2	2	2	2
Nuance Power PDF	2	2	1	1	1	1	1	1	1	1	2	1	2	2	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Soda PDF Desktop	2	1	1	1	1	1	1	1	2	2	1	1	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
PDF Architect	2	1	1	1	1	1	1	1	2	2	1	1	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Poppler (Evince/Okular)	2	–	2	1	2	2	2	2	2	2	2	1	2	2	2	2	–	–	2	2	2	2	2	2	2	–	–	2	2	–	–	–	–
Chrome	1	1	1	2	1	2	2	2	1	2	2	1	1	2	2	1	1	1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
Firefox	–	–	–	1	–	2	2	2	–	–	1	1	2	2	2	–	–	–	–	2	2	2	2	2	2	–	–	2	2	–	–	–	–
Opera	1	1	1	2	1	2	2	2	1	2	2	1	1	2	2	1	1	1	2	2	2	2	2	2	2	2	2	2	2	–	–	–	–
Perfect PDF Reader	2	1	1	1	1	2	2	2	–	–	2	1	2	2	2	1	2	–	–	2	–	2	–	2	–	2	–	–	–	2	–	–	–
Perfect PDF Premium	2	1	1	1	1	2	2	2	–	–	2	1	2	2	2	1	–	–	–	2	–	2	–	2	–	–	–	2	2	–	–	–	–
PDF Studio Viewer/Pro	–	1	1	1	1	1	1	1	2	2	2	1	2	2	2	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
iSkysoft PDF Editor	1	1	1	2	1	2	2	2	1	2	2	1	1	2	2	1	1	1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
Master PDF Editor	1	1	1	2	1	2	2	2	1	2	2	1	1	2	2	1	1	1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
PDFelement	1	1	1	2	1	2	2	2	1	2	2	1	1	2	2	1	1	1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
Preview	2	–	–	1	–	2	2	2	2	2	2	1	2	2	2	–	–	–	2	2	2	2	2	2	2	–	–	2	2	–	–	–	–
Skim	2	–	–	1	–	2	2	2	2	2	2	1	2	2	2	–	–	–	2	2	2	2	2	2	2	–	–	2	2	–	–	–	–
MuPDF	2	1	2	1	1	2	2	2	1	1	2	1	2	2	2	1	1	1	2	2	2	2	2	1	1	1	1	2	2	1	1	1	1
Safari	2	–	–	1	–	2	2	2	2	2	2	1	2	2	2	–	–	–	2	2	2	2	2	2	2	–	–	2	2	–	–	–	–
Edge	–	–	2	2	2	2	2	2	1	1	2	1	2	2	2	2	2	–	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

1 First text being displayed 2 Second text being displayed – No text being displayed

Table 4: Detailed results for the content masking class of attacks.